ArgoUML User Manual

A tutorial and reference description

Alejandro Ramirez
Philippe Vanpeperstraete
Andreas Rueckert
Kunle Odutola
Jeremy Bennett
Linus Tolke
Michiel van der Wulp

ArgoUML User Manual: A tutorial and reference description

by Alejandro Ramirez, Philippe Vanpeperstraete, Andreas Rueckert, Kunle Odutola, Jeremy Bennett, Linus Tolke, and Michiel van der Wulp

Abstract

This version of the manual is intended to describe the version 0.24 of ArgoUML.

Table of Contents

Preface

Software design is a cognitively challenging task. Designers must manually enter designs, but the primary difficulty is decision-making rather than data-entry. If designers improved their decision-making capabilities, it would result in better designs.

Current CASE tools provide automation and graphical user interfaces that reduce the manual work of entering a design and transforming a design into code. They aid designers in decision-making mainly by providing visualization of design diagrams and simple syntactic checks. Also many CASE tools provide substantial benefits in the area of version control and concurrent design mechanisms. One area of design support that has been not been well supported is analysis of design decisions.

Current CASE tools are usable in that they provide a GUI that allows designers to access all the features provided by the tool. And they support the design process in that they allow the designer to enter diagrams in the style of popular design methodologies. But they typically do not provide process support to guide the designer through the design task. Instead, designers typically start with a blank page and must remember to cover every aspect of the design.

ArgoUML is a domain-oriented design environment that provides cognitive support of object-oriented design. ArgoUML provides some of the same automation features of a commercial CASE tool, but it focuses on features that support the cognitive needs of designers. These cognitive needs are described by three cognitive theories:

1. reflection-in-action;
2. opportunistic design; and
3. comprehension and problem solving.

ArgoUML is based directly on the UML 1.4 specification. The core model repository is an implementation of the Java Metadata Interface (JMI) which directly supports MOF and uses the machine readable version of the UML 1.4 specification provided by the OMG.

Furthermore, it is our goal to provide comprehensive support for OCL (the Object Constraint Language) and XMI (the XML Model Interchange format).

ArgoUML was originally developed by a small group of people as a research project. ArgoUML has many features that make it special, but it does not implement all the features that commercial CASE tools provide.

The current version (0.24) of ArgoUML implements all the diagram types of the UML 1.4 standard [http://www.omg.org/cgi-bin/doc?formal/01-09-67] (versions of ArgoUML prior to 0.20 implemented the UML 1.3 standard [http://www.omg.org/cgi-bin/doc?formal/00-03-01]). It is written in Java and runs on every computer which provides a Java 2 platform of Java 1.4 or newer. It uses the open file formats XMI [http://www.omg.org/cgi-bin/doc?formal/02-01-01] (XML Metadata Interchange format) (for model information) and PGML [http://www.w3.org/TR/1998/NOTE-PGML] (Precision Graphics Markup Language) (for graph information) for storage. When ArgoUML implements UML 2.0, PGML will be replaced by the UML Diagram Interchange specification.

This manual is the cumulative work of several people and has been evolving over several years. Connected to the release 0.10 of ArgoUML, Jeremy Bennett, wrote a lot of the new material that was added to the earlier versions by Alejandro Ramirez, Philippe Vanpeperstraete and Andreas Rueckert. He also ad-

ded things from some of the other documents namely the developers cookbook by Markus Klink and Linus Tolke, the Quick Guide by Kunle Odutola, and the FAQ by Dennis Daniels. Connected to the release 0.14 changes were made by Linus Tolke, and by Michiel van der Wulp. These changes were mostly to adopt the manual to the new functions and appearance of ArgoUML version 0.14, and introduction of the index. The users and developers that have contributed by providing valuable input, such as review comments or observations while reading and using this manual are too many to name.

ArgoUML is available for free and can be used in commercial settings. For terms of use, see the license agreement presented when you download ArgoUML. We are providing the source code for ArgoUML for you to review, customize to your needs, and improve. Over time, we hope that ArgoUML will evolve into a powerful and useful tool for everyone to use.

This User Manual is aimed at the working designer, who wishes to make use of ArgoUML. The manual is presently written assuming familiarity with UML, but eventually it will support those new to UML.

The manual is written in DocBook/XML and available as both HTML and PDF.

The ArgoUML project welcomes those who want to get more involved. Look at the project website [http://argouml.tigris.org/] to find out more.

Tell us what you think about this User Manual! Your comments will help us improve things. See Section 1.3.3, “User Feedback” .

Chapter 1. Introduction

1.1. Origins and Overview of ArgoUML

1.1.1. Object Oriented Analysis and Design

Over the past decade, Object Oriented Analysis and Design (OOA&D) has become *the* dominant software development paradigm. With it has come a major shift in the thought processes of all involved in the software development life cycle.

Programming language support for objects began with Simula 67, but it was the emergence in the 1980's of hybrid languages, such as C++, Ada and Object Pascal that allowed OOA&D to take off. These languages provided support for both OO and procedural programming. Object Oriented *programming* became mainstream.

An OO system is designed and implemented as a *simulation* of the real world using software artifacts. This premise is as powerful as it is simple. By using an OO approach to *design* a system can be designed and tested (or more correctly simulated) without having to actually build the system first.

It is the development during the 1990's of tools to support Object Oriented *analysis* and *design* that moved this approach into the mainstream. When coupled with the ability to design systems at a very high level, a tool based OOA&D approach has enabled the implementation of more complex systems than previously possible.

The final driver that has propelled OOA&D has been its suitability for modeling graphical user interfaces. The popularity of object based and object oriented graphical languages such as Visual Basic and Java reflect the effectiveness of this approach.

1.1.2. The Development of ArgoUML

During the 1980's a number of OOA&D process methodologies and notations were developed by different research teams. It became clear there were many common themes and, during the 1990's, a unified approach for OOA&D notation was developed under the auspices of the Object Management Group [http://www.omg.org]. This standard became known as the Unified Modeling Language (UML), and is now the standard language for communicating OO concepts.

ArgoUML was conceived as a tool and environment for use in the analysis and design of object-oriented software systems. In this sense it is similar to many of the commercial CASE tools that are sold as tools for modeling software systems. ArgoUML has a number of very important distinctions from many of these tools.

1. It is free.
2. ArgoUML draws on research in cognitive psychology to provide novel features that increase productivity by supporting the cognitive needs of object-oriented software designers and architects.
3. ArgoUML supports open standards extensively - UML, XMI, SVG, OCL and others.
4. ArgoUML is a 100% pure Java application. This allows ArgoUML to run on all platforms for which a reliable port of the Java2 platform is available.
5. ArgoUML is an open source project. The availability of the source ensures that a new generation of software designers and researchers now have a proven framework from which they can drive the development and evolution of CASE tool technologies.

UML is the most prevalent OO modeling language and Java is one of the most productive OO development platforms. Jason Robbins and the rest of his research team at the University of California, Irvine leveraged these benefits in creating ArgoUML. The result is a solid development tool and environment for OO systems design. Further, it provides a test bed for the evolution of object oriented CASE tools development and research.

A first release of ArgoUML was available in 1998 and more than 100,000 downloads by mid-2001 show the impact that this project has made, being popular in educational and commercial fields.

1.1.3. Finding Out More About the ArgoUML Project

1.1.3.1. How ArgoUML is Developed

Jason Elliot Robbins founded the Argo Project and provided early project leadership. While Jason remains active in the project, he has handed off project leadership. The project continues to move forward strongly. There are more than 300 members on the developer mailing list (see http://argouml.tigris.org/servlets/ProjectMailingListList [http://argouml.tigris.org/servlets/ProjectMailingListList]), with a couple of dozen of those forming the core development group.

The developer mailing list is the place where all the discussion on the latest tasks takes place, and developers discuss the directions the project should take. Although controversial at times, these discussions are always kept nice and friendly (no flame-wars and such), so newbies should not hesitate and participate in them. You'll always get a warm welcome there.

If you want to learn how the project is run and how to contribute to it, go the the ArgoUML Web Site Developer Zone [http://argouml.tigris.org/dev.html] and read through the documentation there. The Developers' Cookbook was written specifically for this purpose.

1.1.3.2. More on Infrastructure

Besides the developer mailing list, there's also a mailing for users (see The ArgoUML Mailing List List [http://argouml.tigris.org/servlets/ProjectMailingListList]), where we can discuss problems from a user perspective. Developers also read this list, so highly qualified help will generally be provided.

Before posting to this list, you should take a look at the user FAQ [http://argouml.tigris.org/faqs/users.html] maintained by Ewan R. Grantham.

More information on ArgoUML and other UML related topics is also available on the ArgoUML website [http://argouml.tigris.org], maintained by Linus Tolke.

1.2. Scope of This User Manual

1.2.1. Target Audience

The current release of this document is aimed at experienced users of UML in OOA&D (perhaps with other tools) who wish to transfer to ArgoUML.

Future releases will support designers who know OOA&D, and wish to adopt UML notation within their development process.

A long term goal is to support i) those who are learning design and wish to start with an OOA&D process that uses UML notation, and ii) people interested in modularized code design with a GUI.

1.2.2. Scope

The intention is that this document will provide a comprehensive guide, enabling designers to use ArgoUML to its full extent. It is in two parts.

- A tutorial manual, showing how to work with ArgoUML
- A complete reference manual, recording everything you can do with ArgoUML.

Version 0.22 of the document achieved the second of these.

In this guide there are some things you will not find, because they are covered elsewhere.

- Descriptions of how ArgoUML works on the inside.
- How to improve ArgoUML with new features and functions.
- A trouble shooting guide.
- A summary quick reference to using ArgoUML.

These are covered in the Developers Cookbook [http://argouml-stats.tigris.org/documentation/defaulthtml/cookbook/], the FAQ [http://argouml.tigris.org/faqs/users.html], and the Quick Guide [http://argouml-stats.tigris.org/documentation/quick-guide-0.22/].

1.3. Overview of the User Manual

1.3.1. Tutorial Manual Structure

Chapter 2, *Introduction (being written)* provides an overview of UML based OOA&D, including a guide to getting ArgoUML up and running.

Chapter 4, *Requirements Capture* through Chapter 7, *Code Generation, Reverse Engineering, and Round Trip Engineering* then step through each part of the design process from initial requirements capture through to final project build and deployment.

As each UML concept is encountered, its use is explained. Its use within ArgoUML is then described. Finally a case study is used to give examples of the concepts in use.

1.3.2. Reference Manual Structure

Chapter 8, *Introduction* is an overview of the user interface and provides a summary of the support for the various UML diagram types in ArgoUML. Chapter 10, *The Menu bar* and Chapter 11, *The Explorer* describe the menu bar, and each of the sub-windows of the user interface, known as *Panes*.

Chapter 15, *The Critics* gives details of all the cognitive critics within the system. Eventually ArgoUML will link directly to this manual when giving advice on critics.

Chapter 16, *Top Level Model Element Reference* is an overview of the model elements (i.e. the UML entities that can be placed on diagrams) within ArgoUML. The following chapters (Chapter 17, *Use Case Diagram Model Element Reference* through Chapter 24, *Built In DataTypes, Classes, Interfaces and Stereotypes*) describe, the model elements that can be created through each ArgoUML diagram, and their

properties, as well as some standard model elements provided with the system.

A complete Glossary is provided. Appendix A, *Supplementary Material for the Case Study* provides material to supplement the case study used throughout the document. Appendix B, *UML resources* and Appendix C, *UML Conforming CASE Tools* identify background information on UML and UML CASE tools. Appendix F, *Open Publication License* is a copy of the GNU Free Documentation License.

A future ambition is to provide a comprehensive index.

1.3.3. User Feedback

Please tell us what you think about this User Manual. Your comments will help us make improvements. Email your thoughts to the ArgoUML Users Mailing List [mailto:users@argouml.tigris.org]. In case you would like to add to the missing chapters you should contact the ArgoUML Developer Mailing List [mailto:dev@argouml.tigris.org] to check whether anyone else is working on this part. You can subscribe to either of the mailing lists via the ArgoUML web site [http://argouml.tigris.org].

1.4. Assumptions

This release of the manual assumes the reader is very familiar with UML already. This is reflected in the sparseness of the description of UML concepts in the tutorial.

The case study is described, but not yet fully realized throughout the tutorial. This will be achieved in future releases of the manual.

Part 1. Tutorial

Chapter 2. Introduction (being written)

This tutorial will be taking you through a tour of the use of ArgoUML to model a system.

First you will become familiar with the feel of the product and then we will go through an analysis and development process for a test case. Not every nook and cranny of the product will be demonstrated. That degree of detail is given in the reference materials to be found in subsequent parts of this document.

The state of the model at the end of key sections will be available in .zargo files. These are available so that you can play with various aspects not specifically covered in this tutorial and then restore yourself back to the proper state of the model in your work area. These .zargo files will be identified at the end of the sections whose work they represent.

An ATM (automated teller machine) project has been chosen as a case study to demonstrate the various aspects of modeling that ArgoUML offers. In subsequent sections we are going to develop the ATM example into a complete description in UML. The tutorial, however, will only walk you through part of it.

At this point you should create a directory to contain your project. Name the directory anything you feel is consistent with the rest of your file system. You should name the contents and any subdirectories as directed for reasons that will become apparent.

The case study will be an ATM system. Your company is FlyByNight Industries. You are going to play two roles. That of the Project Manager and that of the Designer Analyst.

We are not going to build a physical ATM, of course. The product that we will build as a case study will be an ATM simulator to be used for testing the design of a physical ATM.

How your company arranges its work into projects is usually determined as much by politics as anything else and is, therefore, out of the scope of this document. We will go into how you structure the project itself once one has been defined.

Chapter 3. UML Based OOA&D

In this chapter, we look at how UML as a notation is used within OOA&D.

3.1. Background to UML

Object orientation as a concept has been around since the 1960's, and as a design concept since 1972. However it was in the 1980's that it started to develop as a credible alternative to a *functional approach* in analysis and design. We can identify a number of drivers.

1. The emergence of mainstream OO programming languages like SmallTalk and particularly C++. C++ was a pragmatic OO language derived from C, widely used because of its association with Unix.

2. The development of powerful workstations, and with them the emergence into the mainstream of windowing operating user environments. Graphical User Interfaces (GUI) have an inherent object structure.

3. A number of very public major project failures, suggesting that current approaches were not satisfactory.

A number of researchers proposed OOA&D processes, and with them notations. Those that achieved some success include Coad-Yourdon, Booch, Rumbaugh OMT, OOSE/Jacobson, Shlaer-Mellor, ROOM (for real-time design) and the hybrid Jackson Structured Development.

During the early 1990's it became clear that these approaches had many good ideas, often very similar. A major stumbling block was the diversity of notation, meaning engineers tended to be familiar with one OOA&D methodology, rather than the approach in general.

UML was conceived as a common notation, that would be in the interests of all involved. The original standard was driven by Rational Software (www.rational.com [http://www.rational.com], in which three of the key researchers in the field (Booch, Jacobson and Rumbaugh were involved). They produced documents describing UML v0.9 and v0.91 during 1996. The effort was taken industry wide through the Object Management Group (OMG), already well known for the CORBA standard. A first proposal, 1.0 was published in early 1997, with an improved version 1.1 approved that autumn.

ArgoUML is based on UML v1.4, which was adopted by OMG in March 2000. The current official version is UML v1.5 dated March 2003, soon to be replaced by a major revision, UML v2.0, which is in the final stages of standardization and is expected to be complete in 2006.

3.2. UML Based Processes for OOA&D

It is important to understand that UML is a notation for OOA&D. It does not prescribe any particular process. Whatever process is adopted, it must take the system being constructed through a number of phases.

1. Requirements Capture. This is where we identify the requirements for the system, using the language of the *problem domain*. In other words we describe the problem in the “customer's” terms.

2. Analysis. We take the requirements and start to recast them in the language of a putative solution—the *solution domain*. At this stage, although thinking in terms of a solution, we ensure we

keep things at a high level, away from concrete details of a specific solution—what is known as *abstraction*.

3. Design. We take the specification from the Analysis phase and construct the solution in full detail. We are moving from *abstraction* of the problem to its *realization* in concrete terms.

4. Build Phase. We take the actual design and write it in a real programming language. This includes not just the programming, but the testing that the program meets the requirements (*verification*), testing that the program actually solves the customer's problem (*validation*) and writing all user documentation.

3.2.1. Types of Process

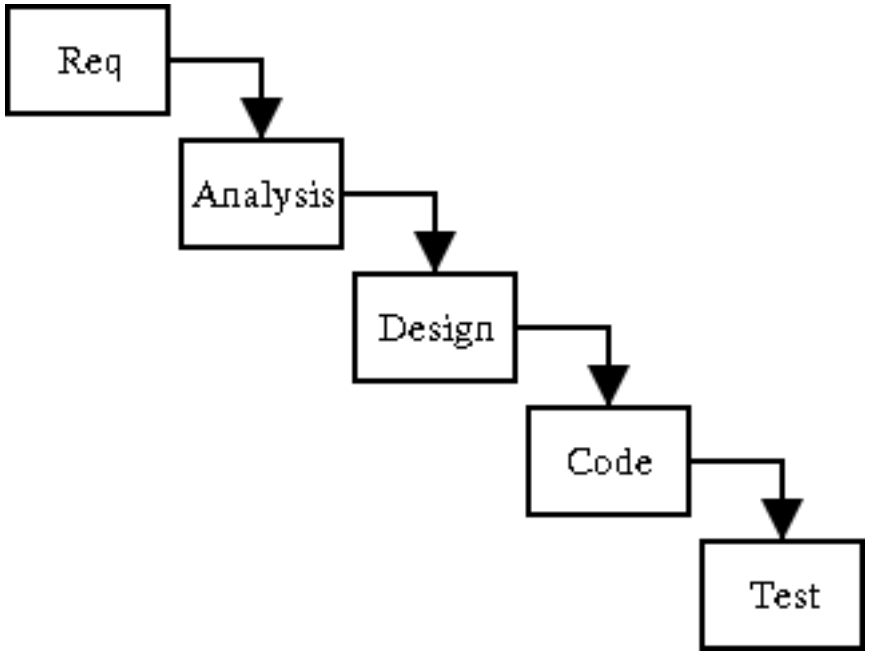
In this section we look at the two main types of process in use for software engineering. There are others, but they are less widely used.

In recent years there has also been a move to reduce the effort required in developing software. This has led to the development of a number of lightweight variants of processes (often known as *agile computing* or *extreme programming*) that are suited to very small teams of engineers.

3.2.1.1. The Waterfall Process

In this process, each stage of the process—requirements, analysis, design and build (code and test) is completed before the next one starts. This is illustrated in Figure 3.1, “The Waterfall Process”.

Figure 3.1. The Waterfall Process



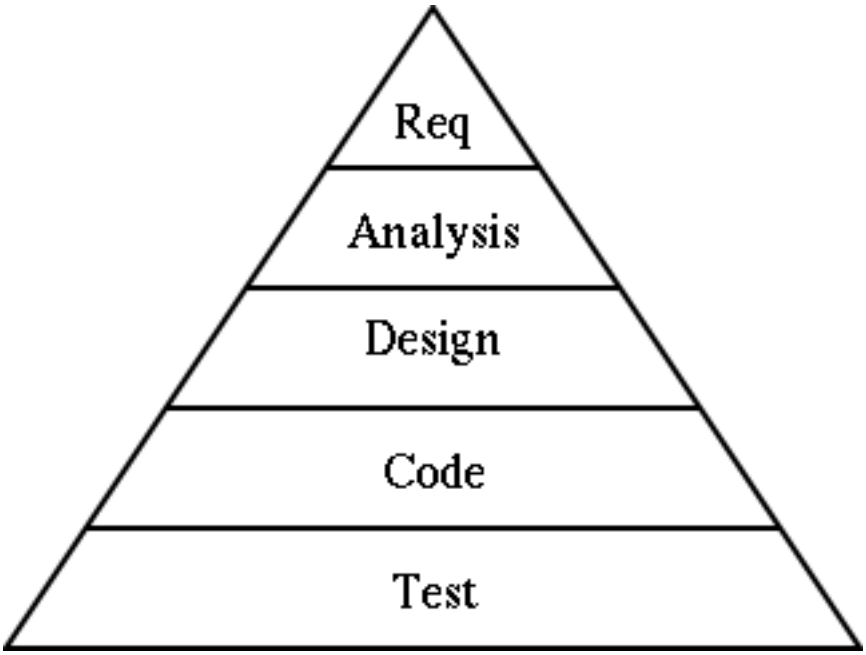
This is a very satisfactory process where requirements are well designed and not expected to change, for example automating a well proven manual system.

The weaknesses of this approach show with less well defined problems. Invariably some of the uncertainties in the requirements will not be clarified until well into the analysis and design, or even code phases, requiring backtracking to redo work.

The worst aspect of this, is that working code does not become available until near the end of the project, and very often it is only at this stage that problems with the original requirements (for example with the user interface) become apparent.

This is exacerbated, by each successive stage requiring more effort, than the previous, so that the costs of late problem discovery are hugely expensive. This is illustrated by the pyramid in Figure 3.2, “Effort Involved in the Steps of the Waterfall Process”.

Figure 3.2. Effort Involved in the Steps of the Waterfall Process



The waterfall process is still probably the dominant design process. However because of its limitations it is increasingly replaced by *iterative* processes, particularly for projects where the requirements are not well defined.

3.2.1.2. Iterative Development Processes

In recent years a new approach has been used, which aims to get at least part of the code up and running as quickly as possible, to bring discovery of problems forward in the development cycle.

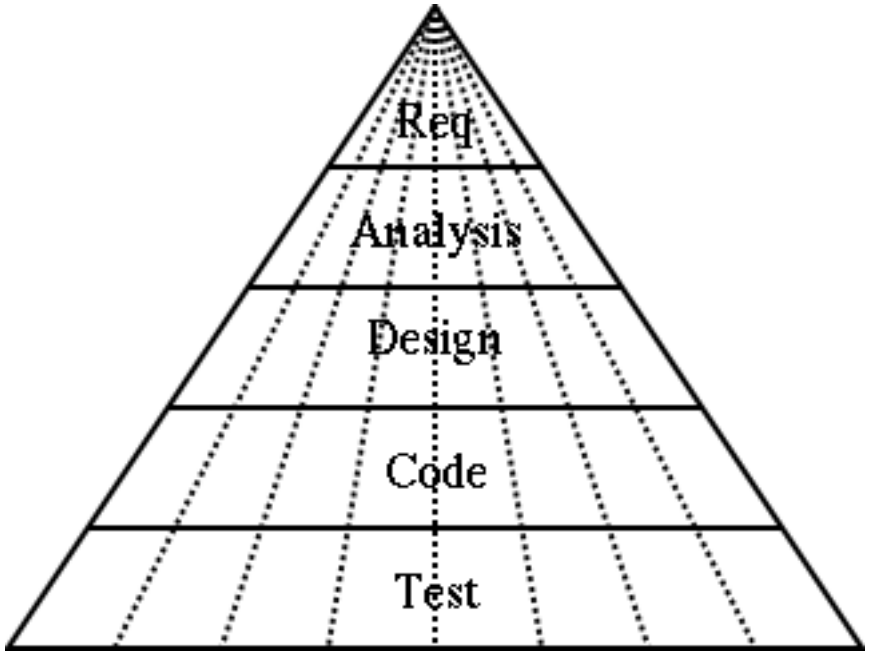
These processes use a series of “mini-waterfalls”, defining a few requirements (the most important) first, taking them through analysis, design and build to get an early version of the product, with limited functionality, related to the most important requirements. Feedback from this can then be used to refine the requirements, spot problems etc before more work is done.

The process is then repeated for further requirements to construct a product with a step up in functionality. Again further feedback can be applied to the requirements.

The process is repeated, until eventually all requirements have been implemented and the product is complete. It is this *iteration* that gives these processes their name. Figure 3.3, “Effort Involved in the

Steps of an Iterative Process" shows how this process compares to the pyramid structure of the Waterfall Process.

Figure 3.3. Effort Involved in the Steps of an Iterative Process



The growth in popularity of iterative processes is closely tied to the growth of OOA&D. It is the clean encapsulation of objects that allows a part of a system to be built with stubs for the remaining code clearly defined.

3.2.1.2.1. The Rational Unified Process

Perhaps the best known Iterative Process is the Rational Unified Process (RUP) from Rational Software (www.rational.com [http://www.rational.com]).

This process recognizes that our pyramid view of even slices of the waterfall is not realistic. In practice the early iterations tend to be heavy on the requirements end of things (you need to define a reasonable amount even to get started), while the later iterations have more of their effort in the design and build areas.

RUP recognizes that iterations can be grouped into a number of *phases* according to their stage in the overall project. Each phase may have one or more iterations.

- In the *inception phase* iterations tend to be heavy on the requirements/analysis end, while any build activity may be limited to emulation of the design within a CASE tool.
- In the *elaboration phase* iterations tend to be completing the specification of the requirements, and starting to focus on the analysis and design, and possibly the first real built code.
- In the *construction phase* iterations the requirements and analysis are more or less completed, and the effort is mostly in design and build.
- Finally, in the *deployment phase* iterations are largely about build activity, and in particular the testing of the software.

Note

It should be clear that testing is an integral part of all phases. Even in the early phases the requirements and design should be tested, and this is facilitated by a good CASE tool.

We shall use an iterative process in this manual, that is loosely based on the RUP.

3.2.1.2.2. Iteration Size

A good rule of thumb is that an iteration should take between six and ten weeks for typical commercial projects. Any longer and you have probably bitten off too many requirements to do in one go. You also lose focus on getting the next working iteration completed. Any shorter and you probably haven't got enough requirements to make a significant advance. In this case the additional overhead associated with an interation will become a problem.

The total number of iterations depends on the size of project. Take the estimated time (working out/guessing that is a whole subject on its own), and divide it into 8 week chunks. Experience seems to suggest that the iterations will divide in the ratio of around 1:2:3:3 into RUP style inception, elaboration, construction and deployment phases. A project that has great vagueness in its specification (some advanced research projects for example) will tend to be heavier on the early phases.

When building a product to contract for a customer the end point is well defined. However when developing a new product for the market place, a strategy that can be used is to decide the product launch date, and hence the end date for completion of engineering (some time before). The time is then divided into iterations, and as much of the product as can be built in that time developed. The iterative process is very effective where time to market is more important than the exact functionality.

3.2.1.3. Recursive Development Processes

Very few software systems are conceived as monolithic artifacts. They are broken down into subsystems, modules etc.

Software processes are the same, with early parts of the process defining a top level structure, and the process reapplying to parts of the structure in turn to define ever greater details.

For example the initial design of a telephone system might identify objects to i) handle the phone lines, ii) process the calls, iii) manage the system and iv) bill the customer. The software process can then be reapplied to each of these four components to identify their design.

OOA&D with its clean boundaries to objects, naturally supports this approach. Such OOA&D with recursive development is sometimes abbreviated as OOA&D/RD.

Recursive development can be applied equally well to waterfall or iterative processes. It is not an alternative to them.

3.2.2. A Development Process for This Tutorial

For the purpose of this tutorial we will use a stripped down iterative process with recursive development, loosely akin to RUP. The case study will take us through the first iteration, although at the end of the tutorial section of the manual we will look at how the project will develop to completion.

Within that first iteration, we will tackle each of the requirements capture, analysis, design and build activities in turn. Not all parts of the process are based on UML or ArgoUML. We will look at what other material is needed outside.

Within this process we will have an opportunity to see the various UML diagrams in use. The full range

of UML diagrams and how they are supported is described in the reference manual (see Section 16.8, “Diagram”).

3.2.2.1. Requirements Capture

Our requirements capture will use the UML concept of *Use Cases*. Starting with a *Vision Document* we will see how Use Cases can be developed to describe all aspects of the system's behavior in the problem domain.

3.2.2.2. Analysis

During the analysis stage, we will introduce the UML concept of *classes* to allow us to build a top level view of the objects that will make up the solution—sometimes known as a *concept diagram*.

We will introduce the UML *sequence diagram* and *statechart diagram* to capture requirements for the overall behavior of the system.

Finally we will take the Use Cases from the requirements capture stage, and recast them in the language of the solution domain. This will illustrate the UML ideas of *stereotyping* and *realization*.

3.2.2.3. Design

We use the UML *package diagram* to organize the components of the project. We then revisit the class diagram, sequence diagram and statechart diagram, to show how they can be used recursively to design the complete solution.

During this part of the process, we need to develop our system architecture, to define how all the components will fit together and operate.

Although not strictly part of our process, we'll look at how the UML *collaboration diagram* can be used as an alternative to, or to complement the *sequence diagram*. Similarly we will look at the UML *activity diagram* as an alternative or complement to the statechart diagram.

Finally we shall use the UML *deployment diagram* to specify how the system will actually be realized.

3.2.2.4. Build

UML is not really concerned with code writing. However at this stage we will show how ArgoUML can be used for code generation.

We will also look at how the UML Use Case Diagram and Use Case Specification are invaluable tools for a test program.

3.3. Why ArgoUML is Different

In the introduction, we listed the four key things that make ArgoUML different: i) it makes use of ideas from cognitive psychology, ii) it is based on open standards; iii) it is 100% pure Java; and iv) it is an open source project.

3.3.1. Cognitive Psychology

3.3.1.1. Theory

ArgoUML is particularly inspired by three theories within cognitive psychology: i) reflection-in-action, ii) opportunistic design iii) and comprehension and problem solving.

- *Reflection-in-Action*

  This theory observes that designers of complex systems do not conceive a design fully-formed. Instead, they must construct a partial design, evaluate, reflect on, and revise it, until they are ready to extend it further.

  As developers work hands-on with the design, their mental model of the problem situation improves, hence improving their design.

- *Opportunistic Design*

  A theory within cognitive psychology suggesting that although designers plan and describe their work in an ordered, hierarchical fashion, in reality, they choose successive tasks based on the criteria of cognitive cost.

  Simply stated, designers do not follow even their own plans in order, but choose steps that are mentally least expensive among alternatives.

- *Comprehension and Problem Solving*

  A design visualization theory within cognitive psychology. The theory notes that designers must bridge a gap between their mental model of the problem or situation and the formal model of a solution or system.

  This theory suggests that programmers will benefit from:

  1. Multiple representations such as program syntactic decomposition, state transitions, control flow, and data flow. These allow the programmer to better identify elements and relationships in the problem and solution and thus more readily create a mapping between their situation models and working system models.

  2. Familiar aspects of a situation model, which improve designers' abilities to formulate solutions.

3.3.1.2. Practical Application in ArgoUML

ArgoUML implements these theories using a number of techniques.

1. The design of a user interface which allows the user to view the design from a number of different perspectives, and allows the user to achieve goals through a number of alternative routes.

2. The the use of processes running in parallel with the design tool, evaluating the current design against models of how “best practice” design might work. These processes are known as *design critics*.

3. The use of *to-do lists* to convey suggestions from the design critics to the user, as well as allowing the user to record areas for future action.

4. The use of checklists, to guide the user through a complex process.

3.3.2. Open Standards

UML is itself an open standard. ArgoUML throughout has tried to use open standards for all its interfaces.

The key advantage of adherence to open standards is that it permits easy inter-working between applica-

tions, and the ability to move from one application to another as necessary.

3.3.2.1. XML Metadata Interchange (XMI)

XML Metadata Interchange (XMI) is the standard for saving the meta-data that make up a particular UML model. In principle this will allow you to take the model you have created in ArgoUML and import it into another tool.

This clearly has advantages in allowing UML to meet its goal of being a standard for communication between designers.

The reality is not quite this good. Prior to UML 2.0 the XMI file includes no information about the graphical representation of the models, so diagram layout is lost. ArgoUML gets round this by saving graphical information separate from the model (see Section 3.4.3.1, “Loading and Saving”).

3.3.2.2. Graphics Formats - EPS, GIF, PGML, PNG, PS, SVG

- *Encapsulated PostScript (EPS)* [http://en.wikipedia.org/wiki/Encapsulated_PostScript] file is a PostScript file which satisfies additional restrictions. These restrictions are intended to make it easier for software to embed an EPS file within another PostScript document.

- *Graphics Interchange Format (GIF)* [http://en.wikipedia.org/wiki/GIF] is a patent encumbered format, although the patents will run out in August of 2006.

- *Precision Graphics Markup Language (PGML)* [http://en.wikipedia.org/wiki/PGML] is an XML-based language for representing vector graphics. It was a W3C draft, but was not adopted as a recommendation. PGML and VML, another XML-based vector graphics language, were later joined and improved upon to create SVG (see below).

- *Portable Network Graphics (PNG)* [http://en.wikipedia.org/wiki/PNG] is an ISO/IEC standard (15948:2004) and is also a W3C recommendation. PNG is a bitmap image format that employs lossless data compression. PNG was created to both improve upon and replace the GIF format with an image file format that does not require a patent license to use. PNG is officially pronounced "ping" but it is often just spelled out — probably to avoid confusion with the network tool ping. PNG is supported by the libpng reference library, a platform-independent library that contains C functions for handling PNG images.

- *PostScript (PS)* [http://en.wikipedia.org/wiki/PostScript/] is a page description language and programming language used primarily in the electronic and desktop publishing areas.

- *Scalable Vector Graphics (SVG)* [http://en.wikipedia.org/wiki/Scalable_Vector_Graphics] is an XML markup language for describing two-dimensional vector graphics, both static and animated, and either declarative or scripted. It is an open standard created by the World Wide Web Consortium. The use of SVG on the web is in its infancy. There is a great deal of inertia due to the long-time use of pure raster formats and other formats like Macromedia Flash or Java applets, but also browser support is still uneven, with native support in Opera and Firefox, but Safari and Internet Explorer require a plugin. See PGML above.

3.3.2.3. Object Constraint Language (OCL)

Object Constraint Language (OCL) [[http://en.wikipedia.org/wiki/Object_Constraint_Language] is a declarative language for describing rules that apply to UML models. It was developed at IBM and is now part of the UML standard. Initially OCL was only a formal specification language extension to UML. OCL may now be used with any Meta-Object Facility (MOF) compliant metamodel, including UML. The Object Constraint Language is a precise text language that provides constraint and object query ex-

pressions on any MOF model or metamodel that cannot otherwise be expressed by diagrammatic notation.

3.3.3. 100% Pure Java

Java was conceived as an interpreted language. It doesn't have a compiler to produce code for any particular target machine. It compiles code for its own target, the *Java Virtual Machine (JVM)*.

Writing an interpreter for a JVM is much easier than writing a compiler, and such machines are now incorporated into almost every Web Browser. As a result most machines can run Java, with no further work.

(In case you wonder why all languages aren't like this, it is because interpreted languages tend to be slower than compiled languages. However with the high performance of modern PCs, the trade-off for portability is worthwhile for many applications. Furthermore modern multi-level caches can mean that interpreted languages, which produce denser code, may actually not be that much slower anyway.)

By choosing to write ArgoUML in pure Java, it is immediately made available to the maximum number of users with the minimum amount of effort.

3.3.4. Open Source

ArgoUML is an *open source* project. That means anyone can have a free copy of the source code, change it, use it for new purposes and so on. The only (major) obligation is that you pass your code on in the same way to others. The precise nature of what you can and can't do varies from project to project, but the principle is the same.

The advantage is that a small project like ArgoUML suddenly is open to a lot of additional help from those who can chip in their ideas for how the program might be improved. At any one time there may be 10, 15, 20 or more people making significant contributions to ArgoUML. To do that commercially would cost $1m+ per year.

Its not just a spirit of pure altruism. Contributing is a way of learning “hands-on” about leading edge software. Its a way of getting a lot of visibility (over 1,125,000 people had downloaded ArgoUML by the end of 2005). That's a lot of good experience on a résumé and a lot of potential employers seeing you!

And its great for the ego!

Open Source doesn't preclude making money. Gentleware www.gentleware.com [http://www.gentleware.com] sell a commercial version of ArgoUML, Poseidon. Their value proposition is not a piece of private code. Its the commercial polish and support that take risk out of using ArgoUML in a commercial development, allowing customers to take advantage of ArgoUML's leading edge technology.

3.4. ArgoUML Basics

The aim of this section is to get you started with ArgoUML. It takes you through obtaining the code and getting it running.

3.4.1. Getting Started

3.4.1.1. System Requirements

Since ArgoUML is written in 100% pure Java, it should run on any machine with Java installed. Java, version 1.4 or later is needed. You may have this in place, but if not it can be downloaded free from

www.java.com [http://www.java.com]. Note that you only need the Java Runtime Environment (JRE), there is no need to download the whole Java Development Kit (JDK).

ArgoUML needs a reasonable amount of computing resource. A PC with 200MHz processor, 64Mb RAM and 10Mb of space available on a harddisk should be adequate. Download the code from Download section of the project website argouml.tigris.org [http://argouml.tigris.org]. Choose the version that suits your needs as described in the section below.

3.4.1.2. Downloading Options

You have three options for obtaining ArgoUML.

1. Run ArgoUML directly from the Web Site using Java Web Start. This is the easiest option.
2. Download the binary executable code. This is the right option if you intend using ArgoUML regularly and is not that difficult.
3. Download the source code using CVS and build your own version. Choose this option if you want to look at the internal workings of ArgoUML, or want to join in as a developer. This option does require the whole JDK (see Section 3.4.1.1, “System Requirements”).

All three options are freely available through the project web site, argouml.tigris.org [http://argouml.tigris.org].

3.4.1.3. ArgoUML Using Java Web Start

There are two steps to this.

1. Install Java Web Start on your machine. This is available from java.sun.com/products/javawebstart [http://java.sun.com/products/javawebstart], or via the `Java Web Start` link on the ArgoUML home page [http://argouml.tigris.org].
2. Click on the `Launch latest stable release` link on the ArgoUML home page [http://argouml.tigris.org].

Java Web Start will download ArgoUML, cache it and start it the first time, then on subsequent starts, check if ArgoUML is updated and only download any updated parts and then start it. The ArgoUML home page [http://argouml.tigris.org] also provides details on starting ArgoUML from the Java Web Start console.

3.4.1.4. Downloading the Binary Executable

If you choose to download the binary executable, you will have a choice of downloading the latest stable version of the code (which will be more reliable, but not have all the latest features), or the current version (which will be less reliable, but have more features). Choose according to your own situation.

ArgoUML comes in `.zip` or `tar.gz` flavors. Choose the former if you are a Microsoft Windows user, and the latter if you are running some flavor of Unix. Unpacking is as follows.

- On Windows. Unzip the `.zip` file with WinZip, or on later versions of Windows (ME, XP) copy the files out of the compressed folder and put them into a directory of your choosing.
- On Unix. Use GNU tar to unzip and break out the files to a directory of your choice `tar zxvf <file>.tar.gz`. If you have an older version of tar, the `z` option may not be avail-

able, so use `gunzip < file.tar.gz | tar xvf -`.

You should have a directory containing a number of `.jar` files and a `README.txt`.

3.4.1.5. Problems Downloading

If you get completely stuck and you have no local assistance, try the web site, particularly the FAQ [http://argouml.tigris.org/faqs/users.html]. If this still doesn't solve the problem, try the ArgoUML users' mailing list.

You can subscribe through the mailing lists section of the project web site argouml.tigris.org [http://argouml.tigris.org], or send an empty message to users@argouml.org [mailto:users@argouml.org] with the subject line `subscribe`.

You can then send your problem to users@argouml.org [mailto:users@argouml.org] and see how other users are able to help.

The users' mailing list is an excellent introduction to the live activity of the project. If you want to get further involved there are additional mailing lists that cover the development of the product and issues in the current and future releases.

3.4.1.6. Running ArgoUML

To run ArgoUML depends on whether you use Microsoft Windows or some flavor of Unix.

- On Windows. Start an MSDOS shell window by e.g. using Start/Run with “command” in the text window. In the window change to the directory holding your ArgoUML files and type `java -jar argouml.jar`. This method has the advantage that progress and debugging information is visible in the DOS window. Alternatively create a batch file (.bat) containing the above command, with a shortcut to it on the desktop. The batch file should end with a "pause" statement in case any debugging information is created during a run. On some systems, simply (double) clicking on the `argouml.jar` file works. On others doing so initiates a zip utility. Refer to you operating system instructions or help facility to determine how to configure this.
- On Unix. Start a shell window and type `java -jar argouml.jar`

3.4.1.7. Problems Running ArgoUML

It's unusual to encounter problems if you have made a successful download. If you can't solve the problem. Try the users' mailing list (see Section 3.4.1.5, “Problems Downloading”).

- Wrong JRE. The most common issue is not having a new enough Java Runtime Environment (it must be 1.4 or later).
- Wrong language. If the product came up in a language you can't read or just don't want, go to the second leftmost menu item in the menu bar at the top of the screen. Select the bottom most menu entry in the drop down. Figure 3.5, “Setting Language in the Appearance Pane” shows this in Russian. Then click on the second tab from the bottom in the column of tabs on the left. Drop down the list as shown in Figure 3.5, “Setting Language in the Appearance Pane” and select a language. Note that the languages are listed in themselves. The language shown as being selected is German in which the word for “German” is “Deutsch”. You will have to exit ArgoUML and restart it for the change to take effect. Use the X button at the upper right.

Figure 3.4. Finding the Settings Wizard

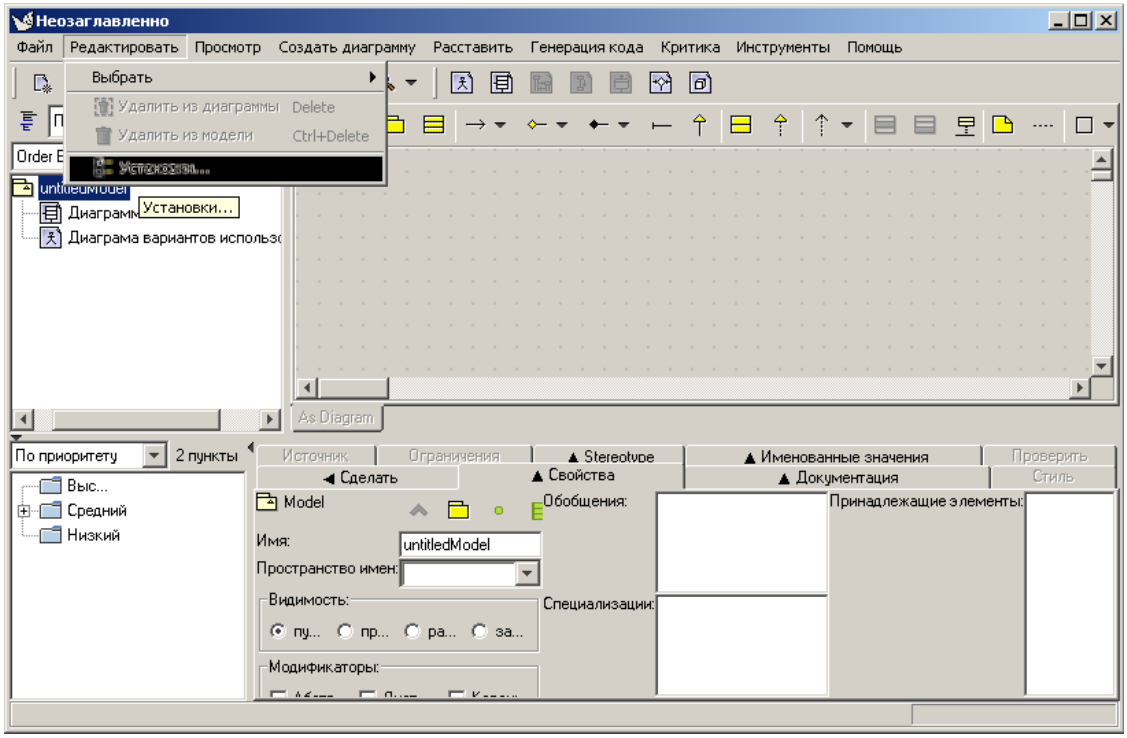
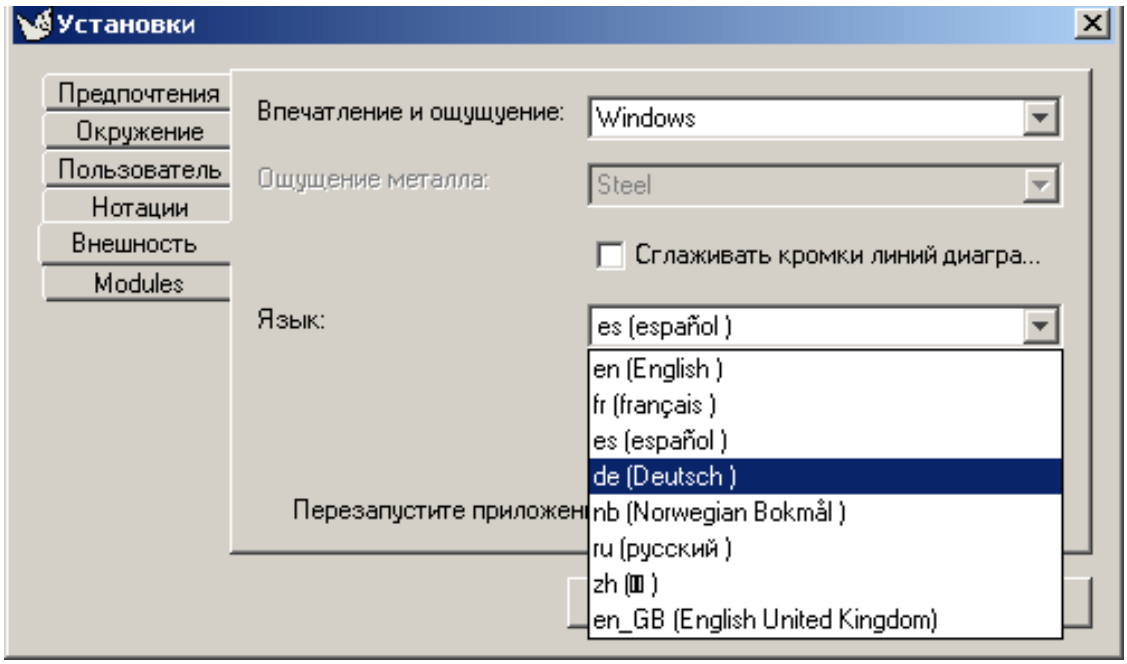


Figure 3.5. Setting Language in the Appearance Pane



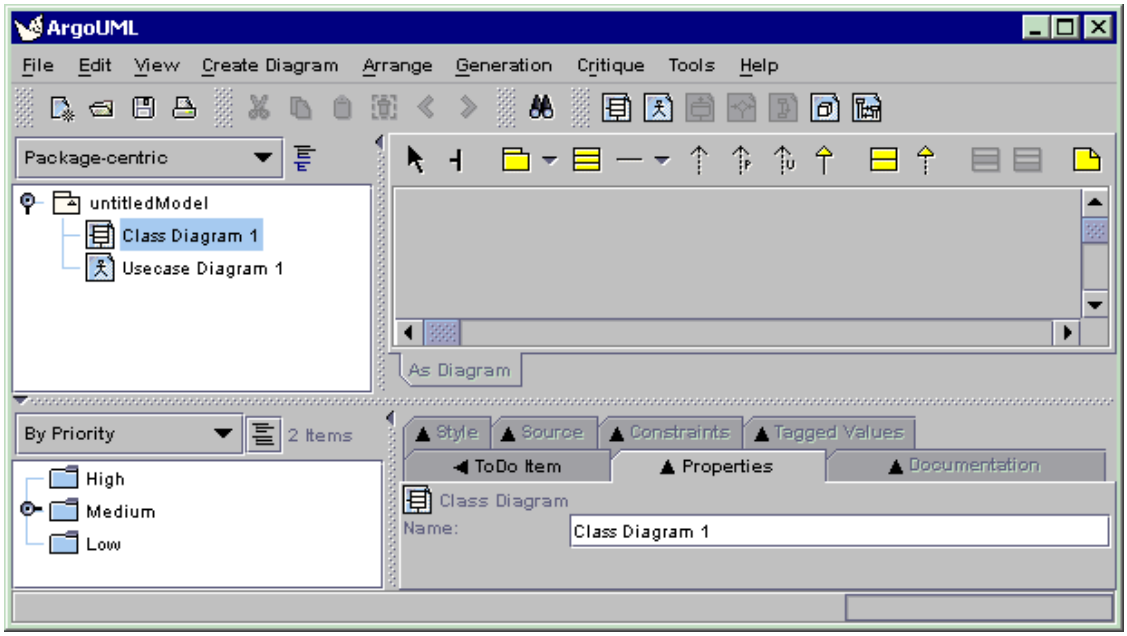
3.4.2. The ArgoUML User Interface

Before beginning the Case Study, you need to become familiar with the user interface. Start by reading the introduction to the User Interface Reference. See Chapter 8, *Introduction*. You should also read the Section 8.2, “General Mouse Behavior in ArgoUML”

As you go through this tutorial you will be told what to do, and when to do it but how to do it will often be left to the User Interface Reference. It is not necessary at this point to read all of the Reference, but you should leaf through enough of it to become familiar with how to find things in it. Every attempt will be made to direct you to the appropriate part of the Reference at those points in the tutorial where they apply.

Figure 3.6, “Initial ArgoUML window”, shows the main ArgoUML window as it appears when ArgoUML is first entered.

Figure 3.6. Initial ArgoUML window



Grab the vertical divider bars and move them back and forth. Grab the horizontal divider bar and move it up and down. Play around a little with the little arrows at the left or top of the divider bars. See Section 8.3, “General Information About Panes”.

3.4.2.1. The Menu Bar and Toolbars

The menu bar and toolbars give access to all the main features of ArgoUML. As is conventional, menu options and toolbar options that are not available (disabled) are grayed out and menu items that invoke a dialog box are followed by an ellipsis (...). At this time you should read Chapter 9, *The Toolbar* and Chapter 10, *The Menu bar*.

File menu. The standard file menu entries present no surprises and we will just use them when needed without first showing how they work. A number of other actions are available that are peculiar to ArgoUML and we will go over them here.

1. *File=>Revert to Saved.* This has the same effect as File=>Open Project selecting the current

project.

2. *Export/Import.* Select the project line at the top of the Explorer. It should say "untitledModel" unless you have changed it. Perform a *File=>Export XMI* action using "DeleteThis" for an output name in the file chooser dialog. Select the "Properties" tab in the "Details Pane" and change the name to something else, anything will do. Perform a *File=>Import XMI* action. It will ask you whether you want to save the changes you have just made. Click on "No" and then in the file choosed that comes up select the "DeleteThis.xmi" file that you just wrote out. Observe that the name of the model has reverted back to what you had saved.

3. *File=>Import Sources.* We will cover this later. You can't test it now unless you have some Java source code of your own handy.

4. *File=>Export (All) Graphics.* In the Explorer Pane select one of the diagrams. Either "Class Diagram 1" or "Use Case Diagram 1" (assuming you haven't renamed or deleted them). Perform a *File=>Export Graphics* action. When the file chooser opens it defaults to the last name you saved something to (even from a project no longer open). The file chooser allows you to select from a number of formats. Drop down the "Files of type" combobox and observe the choices. Cancel out as there is nothing useful to save. Perform a *File=>Export All Graphics* action. Notice that this time you can't specify a file name and you can't select a file format. ArgoUML will allow you only to select an output directory. It will then create a file for each of your diagrams using the diagram name for the file name and an extension determined by the default graphics format. Actually, although you can't select file names in the browser panel, you can type one into the edit box. But, if you do that, nothing at all will happen. You will learn more about the default graphics format when we get to the *Edit* menu.

5. *File=>Notation.* We are going to get a little ahead of ourselves here and do a little class diagram work so you can see what notation is all about. In the Explorer Pane select or create a class diagram. See Section 10.6, “The Create Menu” and Section 12.4.3, “Drawing Tools”. Create a class in the diagram. Go to the Detail Pane and create an attribute in the class. See Section 18.6.2, “Class Property Toolbar”. In the Properties tab of the Detail Pane change the multiplicity to "1..*". Now go the the *File Menu* and select *Notation*. Go back and forth between UML and Java observing the changes in the display in the Edit Pane.

6. *File=>Properties.* You can change the Notation language in the Properties dialog as well. Click on *File=>Properties* and select the Notations tab. Set the Notation Language to UML1.4. Turn on all of the options and click Apply. Then turn off all of the options and click Apply observing the changes in the diagram. Set the Default Shadow Width to 8 and click Apply. Notice that nothing happens. This is because you are not setting the Shadow Width, but its default. The next time you create a class in a diagram, this new shadow value will apply.

Edit menu. The edit menu does not look like what you are used to in other products. There are no "Cut", "Copy", or "Paste" actions. All of the choices are peculiar to ArgoUML so we are going to cover all of them in detail.

1. *Edit=>Select.*

   - Select a class diagram in the Explorer Pane. If there is none there create one using *Create=>New Class Diagram*. Create three classes using the class tool described in the User Interface Reference section on Class Diagram Specific Tools. Double click on it and then click in the Edit Pane for the class diagram in three different locations.

   - Undo the current mode by clicking on the "Select" tool. See Section 12.4.1, “Layout Tools”. This allows you to do things in the Edit Pane other than creating classes.

   - Each of the classes in the diagram has three vertically spaced sections. Double click in the top section of each class and enter a name for the class then hit the enter key. Just name the classes

"A", "B", and "C". Select class A, then class B, and then class C either in the Edit Pane or in the Explorer Pane.

- Now do an *Edit=>Select=>Navigate Back*. Class B should now be selected. Do another *Edit=>Select=>Navigate Back*. Class A should now be selected. Finally, do an *Edit=>Select=>Navigate Forward*. Class B should be selected again.

- Do an *Edit=>Select=>Invert Selection*. Classes A and C should now be selected. Do another *Edit=>Select=>Invert Selection*. Class B should be selected again.

- Do an *Edit=>Select=>Remove From Diagram*. Notice that class B is gone from the diagram but still exists in the Explorer Pane.

- Select class B in the Explorer Pane, right click on it and choose "Add to Diagram". Move the cursor back onto the Edit Pane and left click on some part of the diagram where you think it will fit. You should be pretty much right back where you were before you removed it from the diagram. Do an *Edit=>Select=>Delete From Model*. Now class B should be gone both from the diagram and from the Explorer Pane.

2. *Edit=>Configure Perspectives*. Read Section 11.5, “Configuring Perspectives”. We aren't going to go into this at this point as it needs much larger projects to be displayed than we have available at this point.

3. *Edit=>Settings=>Preferences*. Enter Help=>About ArgoUML. Look at the panel in the "splash" tab. This is known as the Splash Panel. Go to Edit=>Settings=>Preferences. Turn off the "Show Splash Panel" and the "Preload Common Classes" check buttons. Exit from ArgoUML and restart it. Note that the splash panel does not show during the load.

4. *Edit=>Settings=>Environment*. Do *File=>Export Graphics* and observe the file extension that shows at the bottom of the file chooser dialog in the "Files of type" combobox. Go to the *Edit=>Settings=>Environment* editor and pick some other value for Default graphics format. Click "Apply" and then "OK". Go back to the *File=>Export Graphics* dialog and notice that the new format is now the default.

5. *Edit=>Settings=>User*. Enter your name and email address.

6. *Edit=>Settings=>Appearance*. Change the "Look and Feel:" to Metal." Note that the "Metal Theme:" editor becomes anabled. Change the theme to "Very Large Fonts." Click on "Apply" and then "OK." Notice that nothing has happened. Exit from ArgoUML and reopen it. The display should be markedly different. You can change it back or leave it that way as you prefer.

7. *Edit=>Settings=>Notations*. We played around with this earlier with the *File=>Notation* and *File=>Properties* menu items. Start another copy of ArgoUML resize each copy so they can be seen at the same time next to each other. On one of them set the Notation Language to UML (the actual choice will have a version number with it). On the other set the Notation Language to Java.

   On both of them do the following. Turn all of the check boxes on. Do a *File=>New*, create a class in a class diagram. Double click in the attributes section to create an attribute. Double click in the operation section to create a method. Observe the difference in the displays.

8. *Edit=>Settings=>Modules*. This is work in progress. We are not going to mess with it in this version of the tutorial.

View menu. This allows you to switch between diagrams, find model elements in the model, zoom in a diagram, adjust the grid, toggle page break display, and show an XML representation of the project. Do a *File=>New* to get back to a known point. Create an example of each diagram type not already in the Explorer Pane. Click on the (+) sign widgets in the Explorer Pane to expand the tree nodes. Select the

class diagram and give it a name.

1. *View=>Goto Diagram* brings up a Go To Diagram panel. Select the class diagram entry in this panel and click on the "Go to Selection" button. There should be 0 nodes and 0 edges in the left-most column. Click on the "Close" button. In the Details Pane (Properties tab) enter the name as "Blort". Create two classes (See ...) in the class diagram and go back to *View=>Goto Diagram*. You should now see 2 nodes and 0 edges shown. Click on the "Close" button again and link the classes with one of the "line" items like association or generalization. Go back to *View=>Goto Diagram* and you should see 2 nodes and 1 edge(s). Click on the "Close" button again and create a third class. Run the mouse over the icons in the toolbar until you find the one with the tooltip "New Association Class." Click on this tool and then connect the new class to one of the others. Having clicked on the "New Association Class" tool move the mouse over the new class. Press and hold down button 1. Move the mouse over one of the other classes and release button 1. Go back to *View=>Goto Diagram* and you should see 3 nodes and 2 edge(s). Even though it is a class and has a two dimensional representation, it counts as an edge not a node. Select other entries in this panel and click on the "Go to Selection" button in the Go To Diagram panel. Observe the changes in the Explorer Panel.

2. *View=>Find.* At this point you should have three normal classes and an association class in the Explorer Pane. Name them "AA", "AB", "B", and "C". Perform a *View=>Find* operation. Click on the "Find" button. Notice that an "* in *" tab is created below. This tab should show pretty much everything. In the "In Diagram" editor change the "*" to "B*" and click on the "Find Button" observing the contents of the new tab with "* in B*" as a tab label. You should see the three classes, the link (such as an association), and the association class. In the Element Type drop down box select "Interface" and click on the Find button. The new tab "* in B* Inte..." should have no entries in it as we have defined no interfaces. In the Element Type drop down box select "Class" and click on the Find button. The new tab "* in B* Class" should have one fewer entries in it than the "* in B*" tab. Switch back and forth between these two observing the difference. In various of these tabs select an item and click on the "Go To Selection" button observing the change in the selection shown in the diagram and in the Explorer Pane.

3. *View=>Zoom.* As an exeption to a general rule the toolbar equivalent of *View=>Zoom* does not operate in the same way as the correspinding menu item. Highlight *View=>Zoom.* and a submenu will appear that contains "Zoom Out", "Zoom Reset" and "Zoom In". Click on these a few times observing the effect on the diagram then click on the Zoom tool bar icon. This is a magnifying glass next to a down arrow head. You should see a graduated slider bar tool. Grab the pointer in this tool and move it up and down observing the effect on the diagram.

4. *View=>Adjust Grid.*

5. *View=>Adjust Grid Snap.*

6. *View=>Page Breaks.*

7. *View=>XML Dump.*

Create Diagram menu. This allows you to create any one of the seven UML diagram types (class, use case, state, activity, collaboration, deployment and sequence) supported by ArgoUML.

State and activity diagrams can only be created when a class or actor is selected, even though the relevant menu entries are *not* grayed out if this has not been done (nothing will happen under this circumstance).

Arrange menu. This allows you to align, distribute and reorder model elements on a diagram and set the layout strategy for the diagram.

Generation menu. This allows you to generate Java code for selected classes or all classes.

Critique menu. This allows you to toggle the auto-critique on and off, set the level of importance of design issues and design goals and browse the critics available.

Tools menu. This menu is permanently grayed out unless there is some tool available in your version of ArgoUML.

Help menu. This menu gives access to details of those who authored the system, and where additional help may be found.

File Toolbar. This toolbar contains some of the tools from the File menu.

Edit Toolbar. This toolbar contains some of the tools from the Edit menu.

View Toolbar. This toolbar contains some of the tools from the View menu.

Create Diagram Toolbar. This toolbar contains some of the tools from the Create Diagram menu.

3.4.2.2. The Explorer Pane

At this time you should take the time to read Chapter 11, *The Explorer*. The Explorer Pane is fundamental to almost everything that you do and we will be coming back to it again and again in what follows. In fact you will recall we have had to use it already.

There is an expand or contract control in front of the package symbol for “untitledModel” in the Explorer Pane and the package symbol for “Medium” in the To-Do Pane. Click on these controls and observe that these panes are tree widgets that behave pretty much as you would expect them to. The expand or contract control is either plus (+)/minus (-) sign or knob with a right or bottom pointer depending upon the look and feel that you have chosen for an appearance.

Select alternately Class Diagram 1 and Use Case Diagram 1 observing that the detail pane changes to track to the selected item in the Explorer. The detail pane is described in Chapter 12. It is not necessary to read Chapter 12 at this point, but it couldn't hurt.

3.4.2.3. The Editing Pane

At this point take some time to read Chapter 12, *The Editing Pane*.

As we go through the Editing pane changes will sometimes occur in the Details and the To-Do panes. Pay no attention to them for now. We will attend to them when we cover those panes.

Select "Class Diagram 1" in the Explorers Pane. The name is unimportant, if you have changed it, just select the new name. If you have deleted it, first perform a Create=>New Class Diagram action. Click on the "New Package" button in the Edit Pane tool bar. Click somewhere in the edit pane. In the Explorer notice that a package appears named (unnamed Package).

Double click on the "New Class" button in Edit Pane the tool bar. Click first within the package and once outside of it. Notice that within the Explorer, two classes appear in the tree both named (unnamed Class) one of them attached to the model node and the other attached to the (unnamed Package) node.

Click the Select button in the Edit Pane tool bar so you can do things in the Edit Pane without adding new Classes. In the Explorer select the class that is not subordinate to the package. This selects the corresponding class in the diagram. Grab this class and move it into the package. Notice that in the Explorer this class is now also subordinate to the package node.

In the diagram select the other class. Notice that in the Explorer, the selected node changes correspondingly. Grab this class and move it outside of the package and watch what happens in the Explorer.

3.4.2.4. The Details Pane

At this point take some time to read Chapter 13, *The Details Pane*.

Note

- To-Do Item. Discuss differences with other tabs about locations of items selected. Hold particulars for discussion of To-Do Pane.
- Properties,
- Documentation,
- Presentation,
- Source,
- Constraints,
- Stereotype,
- Tagged Values,
- Checklist.
- Remove "images/tutorial/detailsoverview.gif" from file system.

3.4.2.5. The To-Do Pane

At this point take some time to read Chapter 14, *The To-Do Pane*.

Note

- Describe priorities.
- Resolving items.
- Relation to ToDo Item tab in details pane.
- Remove "images/tutorial/todooverview.gif" from file system.

3.4.2.6. Drawing Diagrams

In general diagrams are drawn by using the edit pane toolbar to select the model element desired and clicking in the diagram at the position required.

Model elements that are already in the model, but not on a diagram, may be added to a diagram by selecting the model element in the explorer, using `Add to Diagram` from the drop down menu (button 2) over that model element, and then clicking button 1 at the desired location on the diagram.

As well as UML model elements, the Edit pane toolbar provides for general drawing objects (rectangles, circles, lines, polygons, curves, text) to provide supplementary information on diagrams.

3.4.2.6.1. Moving Diagram Elements

There are several ways to move diagram elements.

3.4.2.6.1.1. Using the Mouse Keys

Select the elements you want to move. By holding down the Ctrl key while selecting you can select several elements to move at the same time.

Now hit your arrow keys. Your elements move a little with every key stroke.

If you also hold down the Shift key, they move a bit faster.

3.4.2.6.1.2. Using the Edit Pane Toolbar

Click on the broom button on the toolbar. Move your mouse to the diagram pane, right click and hold. Now moving your mouse will align elements.

3.4.2.6.2. Arranging Elements

The menu item `Arrange` allows you to align or group elements.

3.4.2.7. Working with Projects

3.4.2.7.1. The Start-Up Window

Figure 3.6, “Initial ArgoUML window” shows the ArgoUML main window as it appears as right after start-up

The main window's client area, below the menu and toolbar, is subdivided into four panes. Starting at the leftmost top pane, and working around the clock, you can see the Explorer, showing a tree view of your UML model, the Editing Pane with its toolbar, two scroll bars and gray drawing area, the Details Pane with the ToDoItem tab selected, and the To-Do Pane with a tree view of the to do items, ranked in various ways selected via the drop down list at the top of the pane.

Each time ArgoUML is started up without a project file as an argument, a new blank project is created. This project contains a model called `untitledModel`. This model contains a blank Class Diagram, called `class diagram 1`, and a blank Use Case Diagram called `use case diagram 1`.

The model and both empty diagrams can be seen in the explorer, which is the main tool for you to navigate through your model.

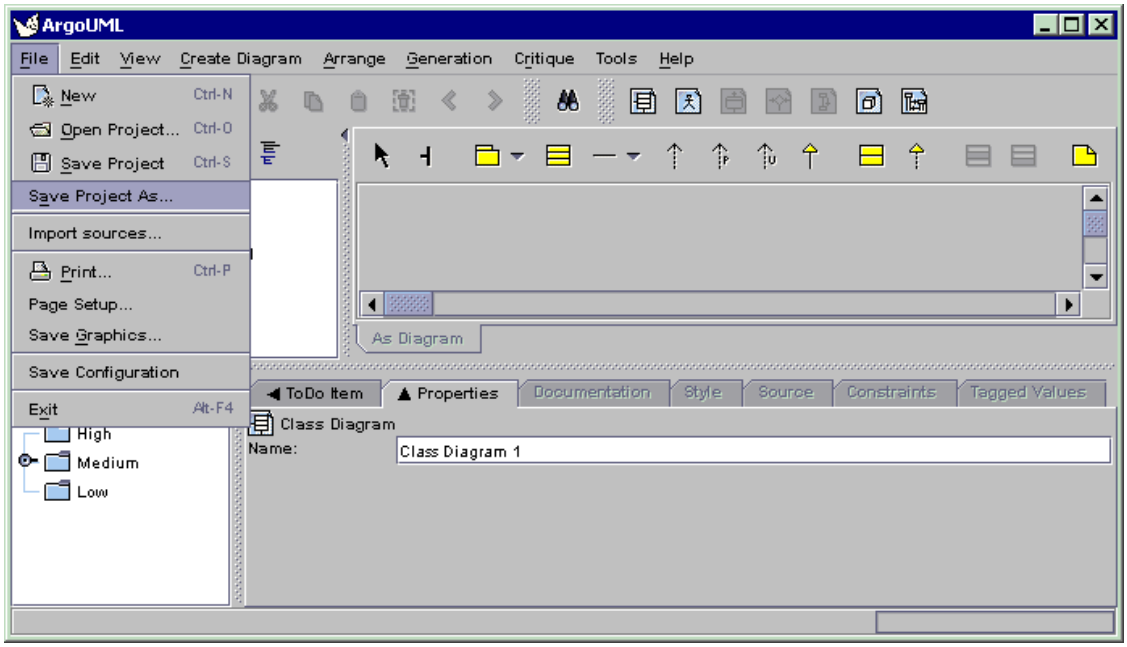
Let's assume for a moment that this is the point where you want to start modeling a new purchasing system. You want to give the name “purchasingmodel” to your model, and you want to store it in a file called `FirstProject`.

3.4.2.7.2. Saving a Project - The File Menu

For now ArgoUML; saves diagrams using an earlier proposed standard, *Precision Graphics Markup Language (PGML)*. However it has the option to export graphical data as SVG for those who can make use of it. When ArgoUML; supports UML 2.0, it will store diagrams using the UML 2.0 Diagram Interchange format.

First, let's save the model in it's current (empty and unnamed) state. On the menu bar, click on `File`, then on `Save Project As...` as shown in Figure 3.7, “Invoking `Save Project As...` ”.

Figure 3.7. Invoking `Save Project As...`



Please notice that the File menu contains the usual options for creating a new project, for opening an existing project, for saving a project under a new name, for printing the currently displayed diagram, for saving the currently displayed diagram as a file, and for program Exit.

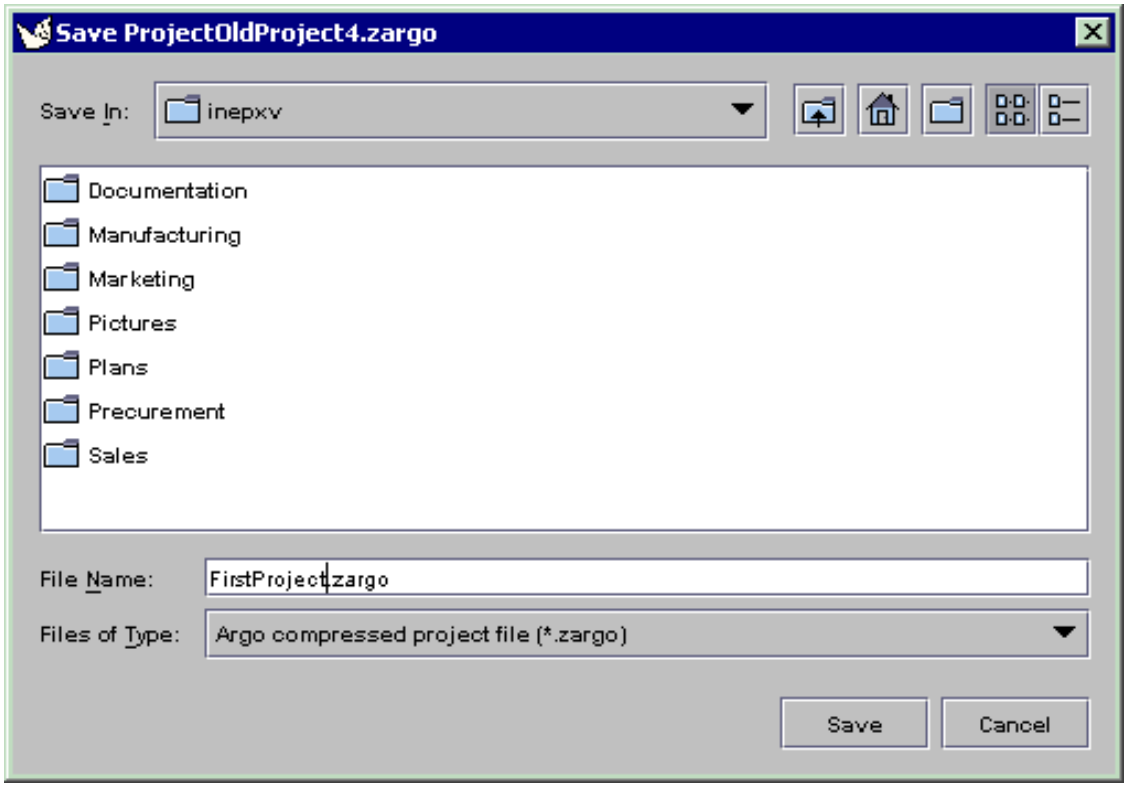
Some of these menu commands can be invoked by pressing key combinations, as indicated on the drop-down menu. For instance, holding down the “Ctrl” key, and pressing “N”, will create a new project.

In the current version, ArgoUML can only contain one active project at a time. In addition, a project can only contain one UML model. Since an UML model can contain an unlimited number of elements and diagrams, this should not present any serious limitations, even for modeling quite large and complex systems.

3.4.2.7.3. The File Chooser Dialog

But let's go back to saving our project. After clicking on the `Save Project As...` menu command, we get the file chooser dialog to enter the file name we wish to use as shown in Figure 3.8, “File Chooser Dialog”.

Figure 3.8. File Chooser Dialog



This is a standard Java FileChooser. Let's go over it in some detail.

The main, outstanding feature, is the scrollable folders list in the center of the dialog. By using the scroll bar on the right, you can move up and down in the list of folders contained inside the currently selected folder. If it is scrollable or not depends on the amount of files and folders shown and also how they are shown. If everything fits the window is not scrollable as seen in the picture.

Double-clicking on one of the displayed folders navigates you into that folder, allowing you to quickly navigate down into the folders hierarchy on your hard disk.

Notice that only folder names, and no file names are displayed in the scrollable area. Indeed, the dialog is currently set up in order to show only ArgoUML project files with an extension of `.zargo`, as can be seen on the lower drop-down control labeled `Files of Type:`.

Also notice that the currently selected folder's name is displayed in the upper drop-down control labeled `Look in:`. A single click on a folder inside the scrollable area does select that folder on screen but does not select the folder for saving.

At the top of the dialog, above the scrollable folder chooser area, there are a few more folder navigation tools.

- 

The Folder drop-down control. Clicking on the down-arrow displays a tree view of the folder hierarchy, allowing you to navigate quickly up the hierarchy, and at the same time to quickly determine where in the hierarchy we are currently positioned.
-

 The Folder-Up icon. Clicking on this icon will bring us to the parent folder of the current folder.

-  The Home Folder icon. Clicking on this icon will bring us to our home directory.
-  The New Folder icon. Clicking on this icon will create a new folder called "New Folder" under the current folder. After the folder is created selecting it an clicking in the name allows us to select the name of our choice.
-  The Folders Presentation Icon.

OK, now we navigate to the directory where we want to save our ArgoUML project, fill in the `File name:` with an appropriate name, such as “FirstProject” and click on the `Save` button.

You have now an active project called `FirstProject`, connected to the file `FirstProject.zargo`.

3.4.3. Output

3.4.3.1. Loading and Saving

3.4.3.1.1. Saving XMI files in ArgoUML

ArgoUML saves the diagram information in a PGML file (with extension `.pgml`, the model information in an XMI file (with extension `.xmi` and information about the project in a file with extension `.argo`. See Section 3.4.3.2.2, “Precision Graphics Markup Language (PGML)” and Section 3.4.3.3, “XMI” for more about PGML and XMI respectively.

All of these are then zipped to a file with extension `.zargo`. You can easily extract the `.xmi` file from the `.zargo` file using any old generic ZIP application. Give it a try and look into the magic of Argo.

Warning

Be aware that double clicking will launch a `ZIP utility`, if one is installed, and NOT Argo.

3.4.3.2. Graphics and Printing

3.4.3.2.1. The Graph Editing Framework (GEF)

GEF is the software package that is the foundation of the diagrams that appear in the Editing Pane. GEF was an integral part of ArgoUML but has been separated. Like ArgoUML it is an open source project available via Tigris [http://www.tigris.org].

3.4.3.2.2. Precision Graphics Markup Language (PGML)

PGML is the current storage format for diagram information used in ArgoUML. In the future, PGML

will be replaced by the UML 2.0 Diagram Interchange format.

3.4.3.2.3. Applications Which Open PGML

PGML is a predecessor of SVG (see Section 3.4.3.2.5, “Scalable Vector Graphics (SVG)”. It was dropped by the W3C Consortium.

Currently there are no other tools that we know of working on PGML.

3.4.3.2.4. Printing Diagrams

Select a diagram, then go to `File=>Export Diagrams`. You can generate GIF, PostScript, Encapsulated PostScript or SVG format.

3.4.3.2.5. Scalable Vector Graphics (SVG)

A World Wide Web Consortium (W3C) standard vector graphics format (http://www.w3.org/TR/SVG/ [http://www.w3.org/TR/SVG/]).

Support is built in to modern browsers, but you can also get a plugin for older browsers from adobe.com [http://www.adobe.com].

3.4.3.2.6. Saving Diagrams as SVG

1. Select `.svg` as the file type.
2. Type the name of the file as you like with the `.svg` tag at the end. Example `myumldiagram.svg`

Et viola! SVG! Give it a try and zoom around a little... They are not pretty though, so if you know anything about rendering beautiful SVG let us know.

Most modern browsers support SVG. If yours doesn't try Firefox [http://www.mozilla.com/firefox/] or get a plugin for your current browser from adobe.com [http://www.adobe.com]

Note

You will not have scroll bars for your SVG unless it is embedded in HTML. Good luck and let us know what you find!

3.4.3.3. XMI

ArgoUML supports XMI 1.0, 1.1, and 1.2 files which contain UML 1.3 and UML 1.4 models. For best compatibility with ArgoUML, export your models using UML 1.4 and XMI 1.1 or 1.2. Be sure to turn off any proprietary extensions (such as Poseidon's diagram data).

With UML versions earlier than UML 2.0, it isn't possible to save diagram information, so no diagrams will be transferred.

There is also a tool that converts XMI to HTML. For more information, see http://www.objectsbydesign.com/projects/xmi_to_html_2.html [http://www.objectsbydesign.com/projects/xmi_to_html_2.html].

3.4.3.3.1. Using XMI from Rational Rose

...

3.4.3.3.2. Using Models Created by Poseidon

In the `Export project to XMI` dialog, but sure to clear the selection of `Save with diagram data`literal>.

3.4.3.3.3. Using Models Created by MagicDraw

...

3.4.3.3.4. XMI Compatibility with other versions of ArgoUML

Versions of ArgoUML prior to 0.19.7 supported UML 1.3/XMI 1.0. After this time, the save format is UML 1.4/XMI 1.2 which is not backward compatible. Newer versions of ArgoUML will read projects written by older versions, but not vice versa. If you might need to return to an older version of ArgoUML you should be careful to save a backup of your old projects.

Additionally, if you write XMI files which need to be read by other tools, you should take into account the different versions. Most modern UML modelling tools should read UML 1.4, but you may have in-house code generators or other tools which are tied to UML 1.3.

3.4.3.3.5. Importing Other XMI Formats into ArgoUML

XMI compatibility between UML modeling tools has improved over the years, but you may still occasionally run into problems.

ArgoUML will not read XMI files which contain UML 1.5 or UML 2.0 models, but it should be able to open most UML 1.4 and UML 1.3 files. If you find one that it can't open, please file a bug report so that a developer can investigate.

3.4.3.3.6. Generating XMI Format

Select the command `File`=>`Export as XMI` and choose a filename.

3.4.3.4. Code Generation

3.4.3.4.1. Code Generated by ArgoUML

It is possible to compile your generated code with ArgoUML, you still need to implement method bodies, though, to get usable results.

3.4.3.4.2. Generating Code for Methods

At the moment you cannot write code for methods (operations) within ArgoUML. The source pane is editable, but the changes are ignored. ArgoUML is a pure design tool for now, no IDE functionality but the desire is there. You might consider using Forte and ArgoUML together—it's a good work around!

You can help us out there if you'd like!

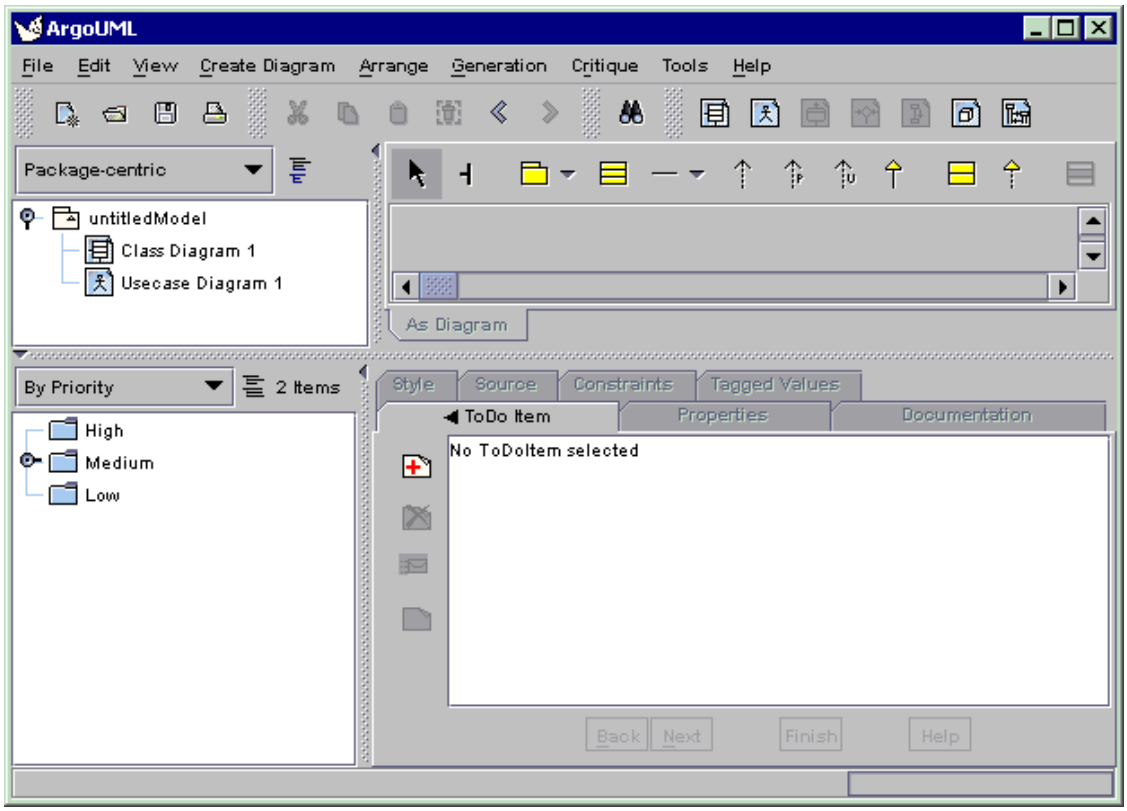
3.4.4. Working With Design Critics

Design critics are part of the practical application of the theories of Cognitive Psychology that are implemented in ArgoUML. See Section 3.3.1, “Cognitive Psychology”

3.4.4.1. Messages From the Design Critics

Where do we stand now? A new project has been created, and is stored in the file `FirstProject.zargo`. Figure 3.9, “ ArgoUML Window Having Saved `FirstProject.zargo` ” shows how your ArgoUML window should look at this stage.

Figure 3.9. ArgoUML Window Having Saved `FirstProject.zargo`



The project contains a top-level package, called `untitledModel`, which contains a class diagram and a use case diagram.

If we look carefully at the screen, we can see that the "Medium" folder in the To-Do Pane (the lower left pane) must contain some items, since its activation icon  is displayed.

Clicking on this icon will open the "Medium" folder. An open folder is indicated by the  icon.

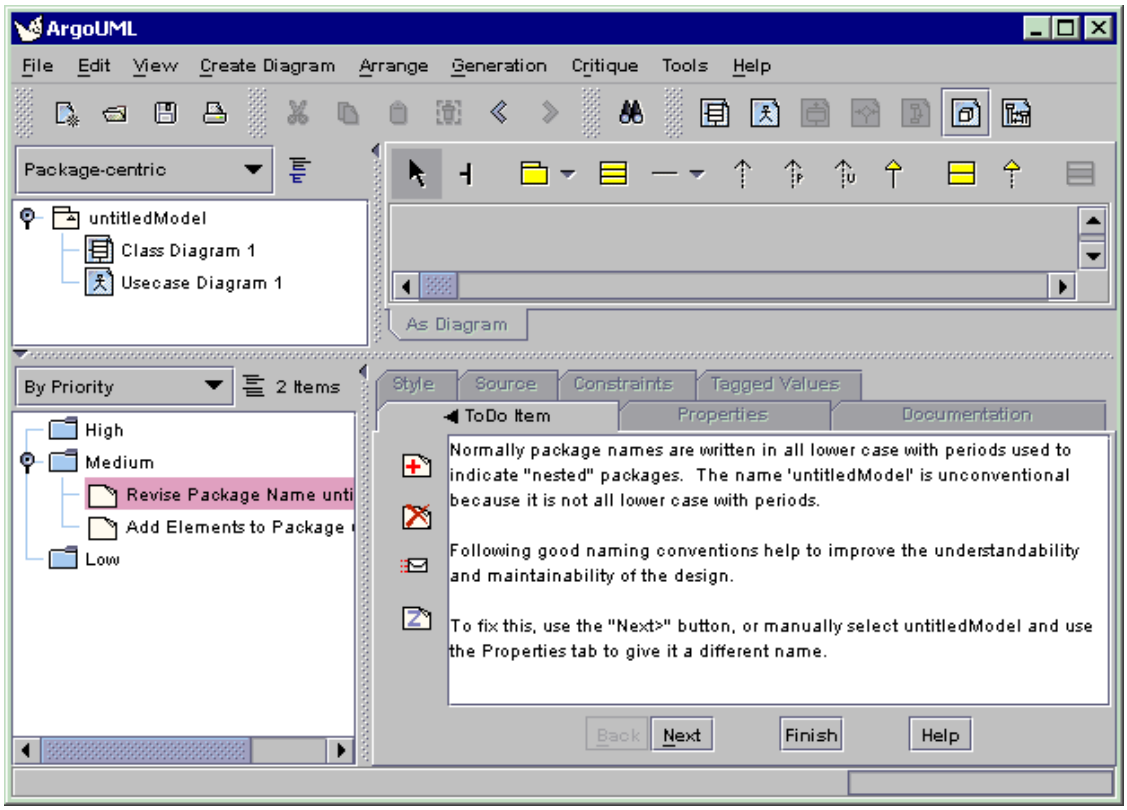
But what is this “To-Do” Pane anyway. You haven't recorded anything yet that has to be done, so where do these to do items originate.

The answer is simple, and is at the same time one of the strong points of ArgoUML. While you are working on your UML model, your work is monitored continuously and invisibly by a piece of code called a *design critic*. This is like a personal mentor that watches over your shoulder and notifies you each time he sees something questionable in your design.

Critics are quite unobtrusive. They give you a friendly warning, but they do not force you into design principles that you don't want or like to follow. Let us take a look at what the critics are telling us. Click on the  icon next to the `Medium` folder, and click on the `Revise Package Name Untitled-Model` item.

Figure 3.10, “ArgoUML Window Showing the Critic Item `Revise Package Name Untitled-Model` ” shows how your screen should now look.

Figure 3.10. ArgoUML Window Showing the Critic Item `Revise Package Name UntitledModel`



Notice that your selection is highlighted in red in the To-Do Pane, and that a full explanation appears now in the Details Pane (the lower right pane). You may have to re-size your Details Pane or to scroll down in order to see the full message as displayed in our example.

What ArgoUML is trying to tell you is that usually, package names are written in lower cases. The default top level package created by ArgoUML is called `untitledModel` and therefore violates a sound design principle. (Actually, this could be considered as a bug within ArgoUML, but it comes in handy to demonstrate the working of critics).

At this point, you can choose to change the package name manually or to impose silence on the design critic for some time or permanently

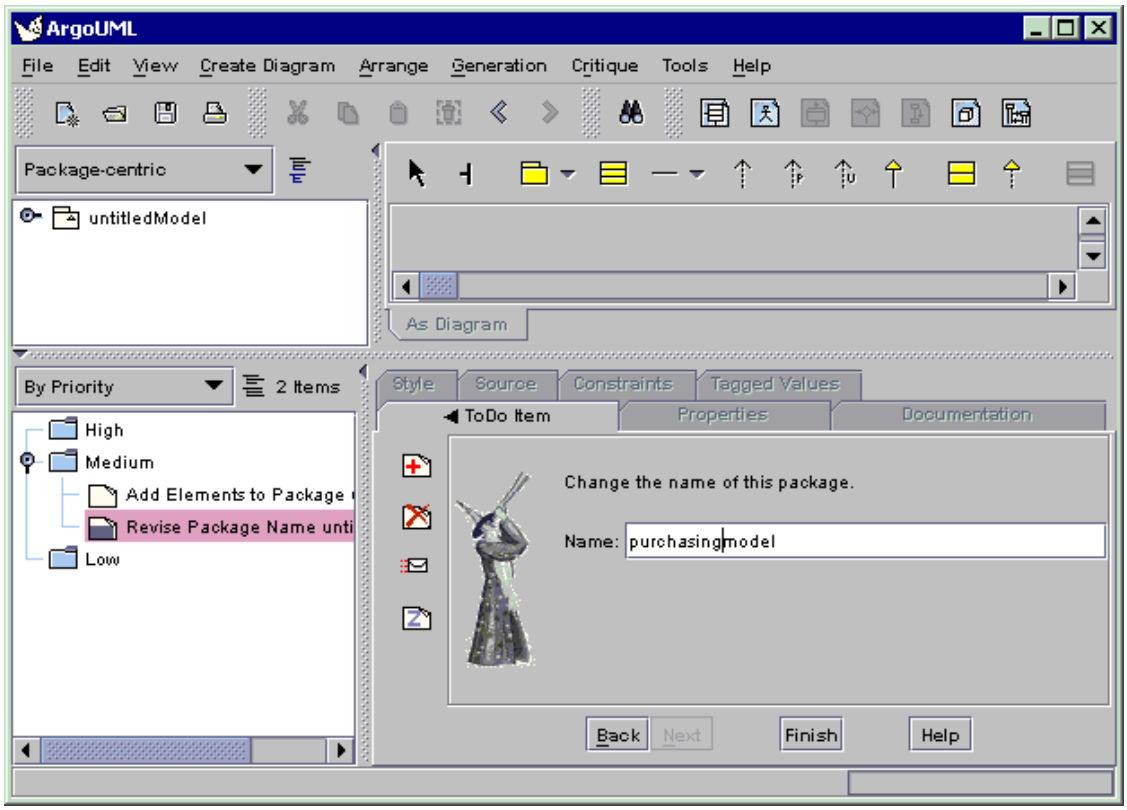
We will do nothing of this (we'll come back to it when we talk about the design critics in more detail) but we'll use another handy feature of ArgoUML—an auto-correct feature.

In order to do that, just click on the `Next` button on the Details Pane. This will cause a renaming wizard to be displayed inside the properties panel, proposing to use the name `untitledmodel` (all in lower case).

3.4.4.2. Design Critics at Work: The Rename Package Wizard

Replace the name `untitledmodel` with `purchasingmodel`, and click on the `Finish` button. Figure 3.11, “ArgoUML Window Showing the Critic Wizard to Rename the Package” shows how the ArgoUML window will now look.

Figure 3.11. ArgoUML Window Showing the Critic Wizard to Rename the Package



Watch now how the design critic note in the To Do panel disappears, leaving only the `Add Elements to Package purchasingmodel` note in the To-Do list.

If this doesn't happen at once, wait for a few seconds. ArgoUML makes heavy use of several threads of execution that execute in parallel. This can cause delays of a few seconds before the information gets updated on the screen.

The package name change should also be reflected in the explorer, in the top left corner of your ArgoUML window.

We are now ready to create our first UML diagram, a Use Case diagram, but first let's save what we've done so far.

Click on the `File` menu item, and select `Save Project`. You can now safely exit ArgoUML without losing your work so far, or go on creating your first diagram.

3.5. The Case Study (To be written)

Here is where we are going to start the Case Study. It is at this point that you define your project; not your product, but your project. It can be argued that modeling concepts should apply here as well but

this has not been well established. If you can take the time to look into the ArgoUML project, you will find that there are a large number of "lines of code" and lines of documentation that are part of the project, but not part of the product. For example, this document is part of the product while the Cookbook and the build.xml files are part of the project only. At a minimum the file structure of the project could be shown in a package diagram.

...

Chapter 4. Requirements Capture

4.1. Introduction

Requirements capture is the process of identifying what the “customer” wants from the proposed system.

The key at this stage is that we are in the problem domain. At this stage we must describe everything from the “customer” perspective and in the language of the “customer”.

The biggest risk we have in requirements capture is to start thinking in terms of possible solutions. That must wait until the *Analysis Phase* (see Chapter 5, *Analysis*). One of the steps of the Analysis Phase will be to take the output of the Requirements Phase and recast it in the language of a deemed solution.

Remember we are using both a *incremental*, and an *iterative* process.

We may well come back to the requirements process again as we break down the problem into smaller chunks, each of which must have its requirements captured.

We will certainly come back through the requirements phase on each iteration as we seek to define the requirements of more and more of the system

Note

The only part of the requirements notation specified by the UML standard is the use case diagram. The remainder is process specific. The process described in this chapter draws heavily on the Rational Unified Process.

4.2. The Requirements Capture Process

We start with a top-level view of the problem we are solving and the key areas of functionality that we must address in any solution. This is our *vision document*, and should be just a few pages long.

For example the top-level view of an automated teller machine (ATM) might be that it should support the following.

1. Cash deposit, cash withdrawal and account inquiries by customers.
2. Maintenance of the equipment by the bank's engineers, and unloading of deposits and loading of cash by the local bank branch.
3. Audit trail for all activities sent to the bank's central computer.

From this top-level view we can extract the principal activities of the system, and the external agents (people, equipment) that are involved in those activities. These activities are known as *use cases* and the external agents are known as *actors*.

Actors may be people or machines. From a practical standpoint it is worth knowing the stakeholder behind any machine, since only they will be able to engage with the requirements capture process.

Use cases should be significant activities for the system. For example customer use of the ATM machine is a use case. Entering a PIN number is not.

There is a gray area between these two extremes. As we shall see it is often useful to break very large use cases into smaller sub-use cases. For example we may have sub-use cases covering cash deposit, cash withdrawal and account inquiry.

There is no hard and fast rule. Some architects will prefer a small number of relatively large use cases, others will prefer a larger number of smaller use cases. A useful rule of thumb is that any practical project ought to require no more than about 30 use cases (if it needs more, it should be broken into separate projects).

We then show the relationship between use cases and actors on one or more use case diagrams. For a large project more than one diagram will be needed. Usually groups of related use cases are shown on one diagram.

We must then give a more detailed specification of each use case. This covers its normal behavior, alternative behaviors and any pre- and post-conditions. This is captured in a document variously known as a *use case specification* or *use case scenario*.

Finally, since use cases are functional in nature, we need a document to capture the non-functional requirements (capacity, performance, environmental needs etc). These requirements are captured in a document known as a *supplementary requirements specification*.

4.2.1. Process Steps

The steps in the requirements capture process can be summarized as follows.

1. Capture an overall view of the problem, and the desired characteristics of its solution in the *vision document*.
2. Identify the *use case* and *actors* from the vision document and show their relationships on one or more *use case diagrams*.
3. Give detailed *use case specifications* for each use case, covering normal and alternate behavior, pre- and post-conditions.
4. Capture all non-functional requirements in a *supplementary requirements specification*.

In any iterative development process, we will prioritize, and early iterations will focus on capturing the key behavior of the most important use cases.

Most modern requirements capture processes agree that it is essential that the authoritative representative of the customer is fully involved throughout the process.

4.3. Output of the Requirements Capture Process

Almost all the output of the requirements capture process is documentary. The only diagram is the use case diagram, showing the relationships between use cases and actors.

4.3.1. Vision Document

Typical sections of this document would be as follows.

- *Summary*. A statement of the context, problem and solution goals.
- *Goals*. What are we trying to achieve (and how do we wish to achieve it).
- *Market Context* or *Contractual Arrangements*. For a market led development, this should indicate target markets, competitive differentiators, compelling events and so forth. For a contractual development this should explain the key contractual drivers.
- *Stakeholders*. The users (in the widest sense) of the system. Many of these will map in to actors, or control equipment that maps into actors.
- *Key Features*. At the very highest level what are they key functional aspects of the problem/desired solution. These will largely map down to the use cases. It is helpful to give some prioritization here.
- *Constraints*. A high level view of the non-functional parameters of the system. These will be worked out in detail in the supplementary requirements specification.
- *Appendix*. A listing of the actors and use cases that will be needed to meet this vision. It is useful to link to these from the earlier sections to ensure comprehensive coverage.

4.3.2. Use Case Diagram

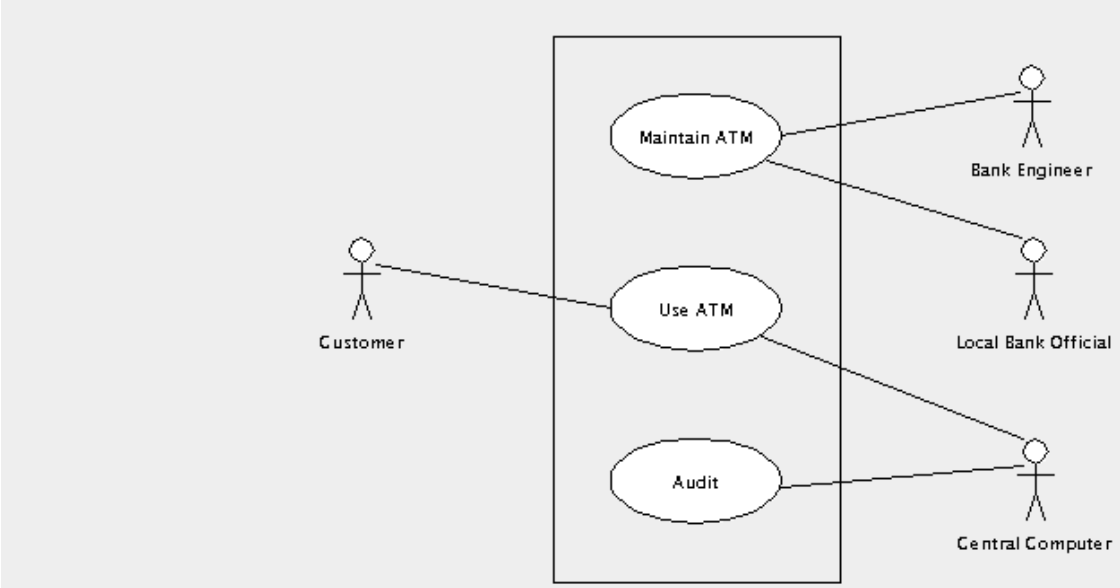
The vision document has identified the use cases and actors. The use case diagram captures how they interact. In our ATM example we have identified “customer uses machine”, “maintain machine” and “audit” as the three main use cases. We have identified “customer”, maintenance engineer“,” “local branch official” and “central computer” as the actors.

Figure 4.1, “Basic use case diagram for an ATM system” shows how this could be displayed on a use case diagram. The use cases are shown as ovals, the actors as stick people (even where they are machines), with lines (known as *associations* connecting use cases to the actors who are involved with them. A box around the use cases emphasizes the boundary between the system (defined by the use cases) and the actors who are external.

Note

Not all analysts like to use a box around the use cases. It is a matter of personal choice.

Figure 4.1. Basic use case diagram for an ATM system



The following sections show how the basic use case diagram can be extended to show additional information about the system being designed.

4.3.2.1. Active and Passive Actors

Active actors initiate interaction with the system. This can be shown by placing an arrow on the association from the actor pointing toward the use case. In the ATM example, the customer is an active actor.

Interaction with *passive* actors is initiated by the system. This can be shown by placing an arrow on the association from the use case pointing toward the actor. In the ATM example, the central computer is a passive actor.

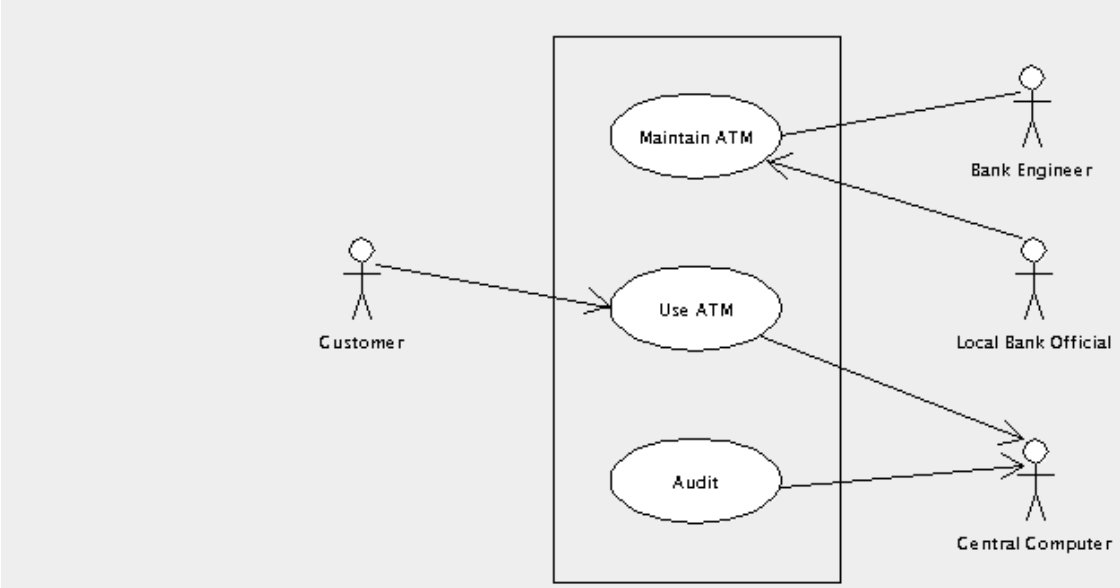
This is a good example where the arrow helps, since it allows us to distinguish an event driven system (the ATM initiates interaction with the central computer) from a polling system (the central computer interrogates the ATM from time to time).

Where an actor may be either active or passive, depending on circumstances, the arrow may be omitted. In the ATM example the bank engineer fits into this category. Normally he is active, turning up on a regular cycle to service the machine. However if the ATM detects a fault, it may summon the engineer to fix it.

The use of arrows on associations is referred to as the *navigation* of the association. We shall see this used elsewhere in UML later on. The choice, by the OMG, of zero vice two arrowheads to show a bidirectional association is unfortunate. Under this convention you cannot distinguish between an association whose navigation has yet to be determined and one that is bidirectional.

Figure 4.2, “Use case diagram for an ATM system showing navigation.” shows the ATM use case diagram with navigation displayed.

Figure 4.2. Use case diagram for an ATM system showing navigation.



4.3.2.2. Multiplicity

It can be useful to show the *multiplicity* of associations between actors and use cases. By this we mean how many instances of an actor interact with how many instances of the use case.

By default we assume one instance of an actor interacts with one instance of a use case. In other cases we can label the multiplicity of one end of the association, either with a number to indicate how many instances are involved, or with a range separated by two periods (`..`). An asterisk (`*`) is used to indicate an arbitrary number.

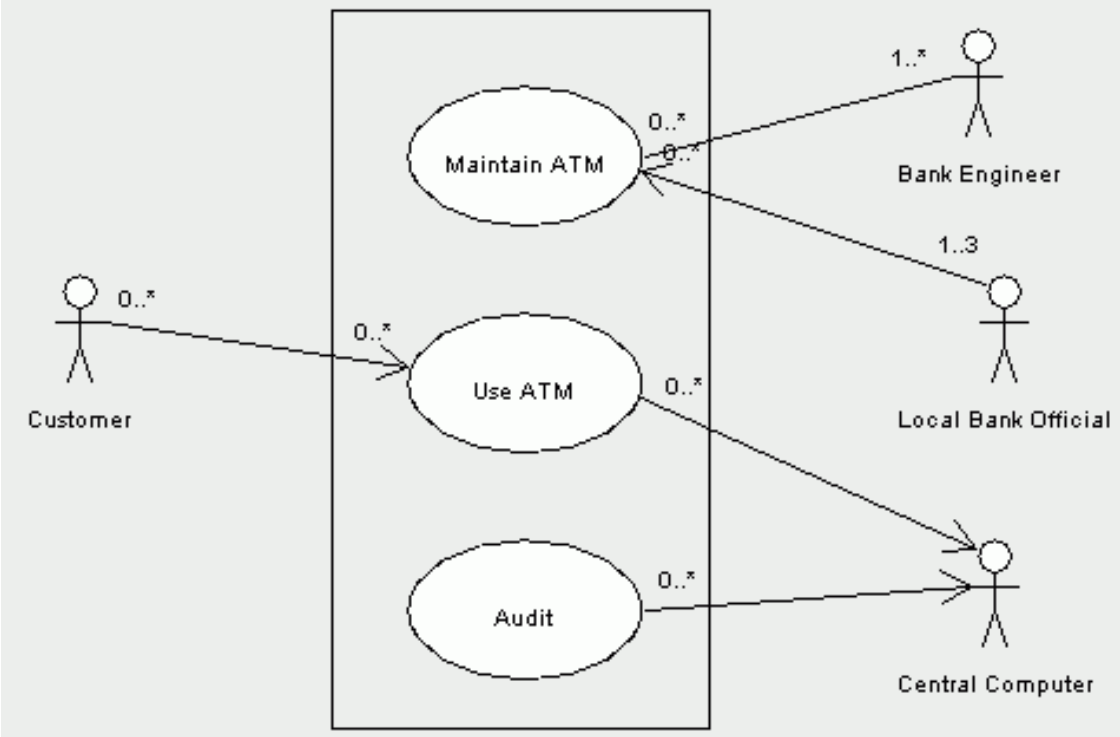
In the ATM example, there is only one central computer, but it may be auditing any number of ATM uses. So we place the label `0..*` at the use case end. There is no need for a label at the other end, since the default is one.

A local bank will have up to three officials authorized to unload and load ATM machines. So at the actor end of the relationship with the use case `Maintain ATM`, we place the label `1..3`. They may be dealing with any number of ATM machines, so at the other end we place the label `0..*`.

There may be any number of customers and there may be any number of ATM systems they could use. So at each end of the association we place the label `0..*`.

Figure 4.3, “Use case diagram for an ATM system showing multiplicity.” shows the ATM use case diagram with multiplicity displayed.

Figure 4.3. Use case diagram for an ATM system showing multiplicity.



Multiplicity can clutter a diagram, and is often not shown, except where it is critical to understanding. In the ATM example we would only choose to show `1..3` against the local bank official, since all others are obvious from the context.

4.3.2.3. Hierarchies of Use Cases

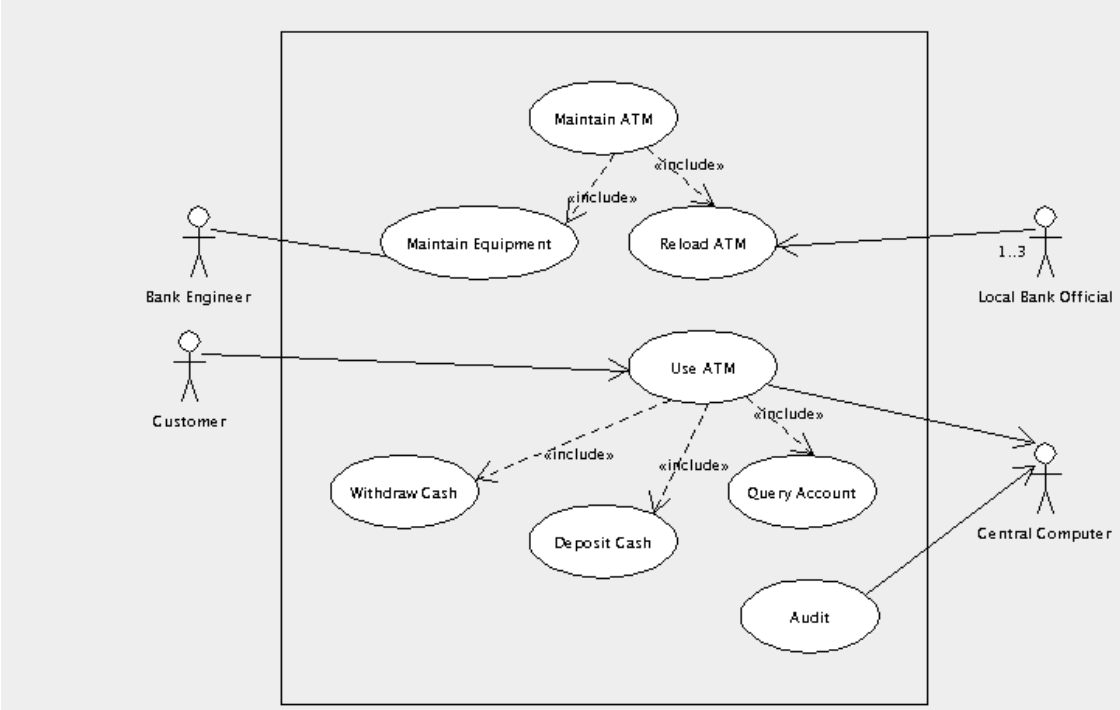
In our ATM example so far we have just three use cases to describe all the behavior of the system. While use cases should always describe a significant chunk of system behavior, if they are too general they can be difficult to describe.

We could for example define the behavior of the use case “Use ATM” in terms of the behavior of three simpler use cases, “Deposit Cash”, “Withdraw Cash” and “Query Account”. The main use case could be specified by *including* the behavior of the subsidiary use cases where needed.

Similarly the “Maintain ATM” use case could be defined in terms of two use cases “Maintain Equipment” and “Reload ATM”. In this case the two actors involved in the main use case are really only involved in one or other of the two subsidiary use cases and this can be shown on the diagram.

The decomposition of a use case into simpler sub-use cases is shown in UML by using an *include relationship*, a dotted arrow from the main use case to the subsidiary, with the label *«include»*.

Figure 4.4. Use case diagram for an ATM system showing include relationships.



Include relationships are fine for breaking down the use case behaviors in to hierarchies. However we may also want to show a use case that is an *extension* to an existing use case to cater for a particular circumstance.

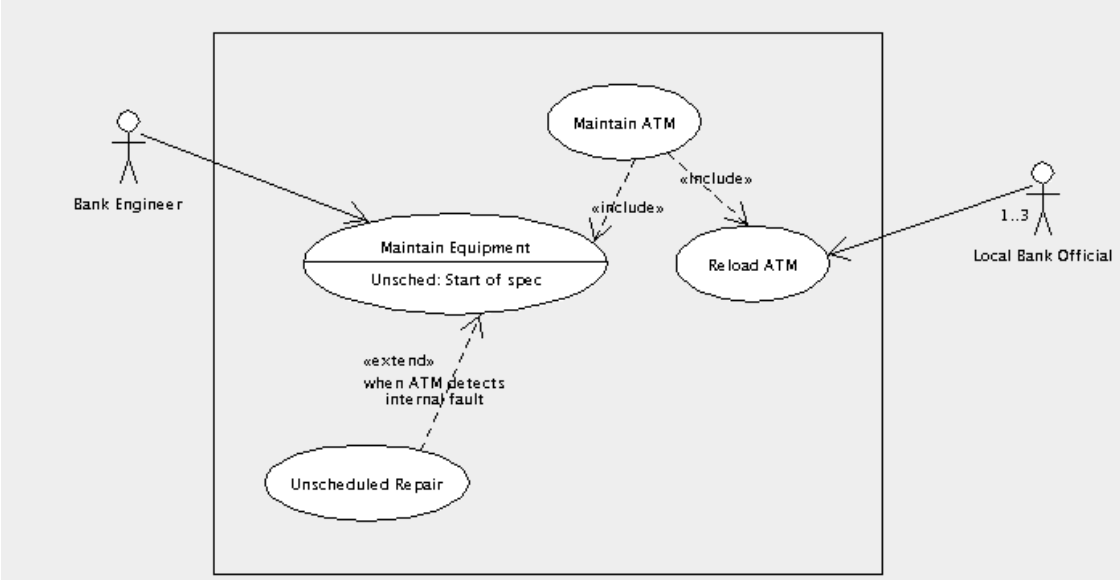
In the ATM example we have a use case covering routine maintenance of the ATM, “Maintain Equipment”. We also want to cover the special case of an unscheduled repair caused by the ATM detecting an internal fault.

This is shown in UML by the *extend* relationship. In the main use case, we specify a name for a location in the description, where an extension to the behavior could be attached. The name and location are shown in a separate compartment within the use case oval. The representation extend relationship is the same as the include relationship, but with the label *«extend»*. Alongside the extend relationship, we specify the condition under which that behavior will be attached.

Figure 4.5, “Use case diagram for an ATM system showing an extend relationship.” shows the ATM use case diagram with an extend relationship to a use case for unscheduled repairs. The diagram is now getting rather complex, and so we have split it into two, one for the maintenance side of things, the other for customer usage and audit.

The “Maintain Equipment” use case defines a name “Unsched”, at the start of its description. The extending use case “Unscheduled Repair” is attached there when the ATM detects an internal error.

Figure 4.5. Use case diagram for an ATM system showing an extend relationship.



Use cases may be linked together in one other way. One use case may be a *generalization* of a subsidiary use case (or alternatively the subsidiary is a *specialization* of the main use case). This is very like the extends relationship, but without the constraint of specific extension points at which the main use case may be extended, and with no condition on when the subsidiary use case may be used.

Generalization is shown on a use case diagram by an arrow with solid line and solid white head from the subsidiary to the main use case. This may be useful when a subsidiary use case specializes the behavior of the main use case at a large number of positions and under a wide range of circumstances. However the lack of any restriction makes generalization very hard to specify precisely. In general use an extend relationship instead.

4.3.3. The Use Case Specification

Each use case must be documented to explain in detail the behavior it is specifying. ArgoUML assists in this area through the generation of graphic files for inclusion in this documentation. This document is known by different names in different processes: *use case specification*, *use case scenario* or even (confusingly) just *use case*.

A typical use case specification will include the following sections.

- *Name*. The name of the use case to which this relates.
- *Goal*. A one or two line summary of what this use case achieves *for its actors*.
- *Actors*. The actors involved in this use case, and any context regarding their involvement.

Note

This should not be a description of the actor. That should be associated with the actor on the use case diagram.

- *Pre-condition*. These would be better named “pre-assumptions”, but the term used everywhere is

pre-conditions. This is a statement of any simplifying assumptions we can make at the start of the use case.

In the ATM example we might make the assumption for the“Maintain Equipment” use case that an engineer is always available, and we do not need to worry about the case where a routine maintenance visit is missed.

Caution

Avoid pre-conditions wherever possible. You need to be absolutely certain that the pre-condition holds under all possible circumstances. If not your system will be under specified and hence will fail when the pre-condition is not true. Alternatively, when you cannot be certain the pre-condition is always true, you will need to specify a second use case to handle the pre-condition being false. In the first case, pre-conditions are a source of problems, in the second a source of more work.

- *Basic Flow*. The linear sequence of steps that describe the behavior of the use case in the “normal” scenario. Where a use case has a number of scenarios that could be normal, one is arbitrarily selected. Specifying the basic flow is described in more detail in Section 4.3.3.1, “Specifying the Basic Flow” below.

- *Alternate Flows*. A series of linear sequences describing each of the alternative behaviors to the basic flow. Specifying alternate flows is described in more detail in Section 4.3.3.2, “Specifying the Alternate Flows”.

- *Post-conditions*. These would be better named “post-assumptions”. This is a statement of any assumptions that we can make at the end of the use case. Most useful where the use case is one of a series of subsidiary use cases that are included in a main use case, where they can form the pre-conditions of the next use case to be included.

  ## Caution

  Like pre-conditions, post-conditions are best avoided. They place a burden on the specification of the use case flows, to ensure that the post-condition always holds. They therefore are also a source of problems and extra work.

- *Requirements*. In an ideal world the vision document, use case diagrams, use case specifications and supplementary requirements specification would form the requirements for a project.

  For most market-led developments, where ownership of requirements is within the same business as the team who will do the development, this is now usually the case. The marketing department can learn use case based requirements capture and analysis to link to their customer facing activities.

  However for external contract developments, customers may insist on a traditional “list of features” as the basis of the contract. Where this is the case, this section of the use case specification should link to the contract features that are covered by the use case.

  This is often done through a third party tool that can link documents, providing automated checking of coverage, in which case this section is not needed, or may be generated automatically.

The final size of the use case specification will depend on the complexity of the use case. As a rule of thumb, most use cases take around 10-15 pages to specify, the bulk of which is alternate flows. If you

are much larger than this, consider breaking the use case down. If you are much smaller consider whether the use case is addressing too small a chunk of behavior.

4.3.3.1. Specifying the Basic Flow

All flows in a use case specification are linear—that is there is no conditional branching. Any choices in flows are handled by specifying another alternate flow that takes over at the choice point. It is important to remember we are specifying behavior here, not programming it.

A flow is specified as a series of numbered steps. Each step must involve some interaction with an actor, or at least generate a change that is observable externally by an actor. Requirements capture should not be specifying hidden internal behavior of a system.

For example we might give the following sequence of steps for the basic flow of the use case "Withdraw Cash" in our ATM example.

1. Customer indicates a receipt is required.
2. Customer enters amount of cash required.
3. ATM verifies with the central computer that the customer can make this withdrawal.
4. ATM dispenses cash to the customer.
5. ATM issues receipt to customer.

Remember this is a sub-use case included in the main “Use ATM” use case, which will presumably handle checking of cards and PINs before invoking this included use case.

Note

The first step is not a condition. We take as our basic flow the case where the customer does want a receipt. The case where the customer does not want a receipt will be an alternative flow.

4.3.3.2. Specifying the Alternate Flows

This captures the alternative scenarios, as linear flows, by reference to the basic flow. Initially we just build a list of the alternate flows.

A.

A.1. Customer does not require a receipt.

A.2. Customer's account will not support the withdrawal.

A.3. Communication to the central computer is down.

A.4. The customer cancels the transaction.

A.5. The customer fails to take the dispensed cash.

Subsequently we flesh out each alternate flow, by reference to the basic flow. For example the first alternate flow might look like.

A.
A.1. Customer does not require a receipt.

A.1. At step 1 of the basic flow the customer indicates they do not want a receipt.
1.
A.1. The basic flow proceeds from step 2 to step 4, and step 5 is not used.
2.

The convention is to number the various alternate flows as A.1, A.2, A.3, etc. The steps within an alternate flow are then numbered from this. So the steps of the first alternate flow would be A.1.1, A.1.2, A.1.3, etc.

4.3.3.3. Iterative Development of Use Case Specifications

Iterative development will prioritize the use cases, and the first iterations will address the most important.

Early iterations will capture the basic flows of the most important use cases with only essential detail and list the headings of the main alternate flows.

Later iterations will address the remaining use cases, flesh out the steps on individual alternate flows and possibly provide more detail on individual steps.

4.3.4. Supplementary Requirement Specification

This captures the non-functional requirements or constraints placed on the system. Since use cases are inherently functional in nature, they cannot capture this sort of information.

Note

Some analysts like to place non-functional requirements in a section at the end of each use case specification, containing the non-functional requirements relevant to the use case.

This can cause some problems. First key non-functional requirements (for example about performance) may need to appear in many use cases and it is bad practice to replicate information. Secondly there are invariably some non-functional requirements that are system wide and need a system wide document. Hence my preference for a single supplementary requirements specification.

There should be a section for each of the main areas of non-functional requirements. The checklist provided by Ian Sommerville in his book *Software Engineering* (Third Edn, Addison-Wesley, 1989) is a useful guide.

- *Speed*. Processor performance, user/event response times, screen refresh time.
- *Size*. Main memory (and possibly caches), disc capacity.
- *Ease of use*. Training time, style and detail of help system.
- *Reliability*. Mean time to failure, probability of unavailability, rate of failure, availability.
- *Robustness*. Time to restart after failure, percentage of events causing failure, probability of data corruption on failure.
- *Portability*. Percentage of target-dependent code/classes, number of target systems.

To this we should add sections on environment (temperature, humidity, lightening protection status) and standards compliance.

4.4. Using Use Cases in ArgoUML

ArgoUML allows you to draw use case diagrams. When you create a new project it has a use case diagram created by default, named `use case diagram 1`. Select this by button 1 click on the diagram name in the explorer (the upper left quadrant of the user screen).

New use case diagrams can be created as needed through `Create Diagram` on the main menu bar or on the Create Diagram Toolbar. They are edited in the editing pane (the upper right quadrant of the user screen).

4.4.1. Actors

To add an actor to the diagram use button 1 click on the actor icon on the editing pane toolbar () and then button 1 click at the location where you wish to place it. The actor can be moved subsequently by button 1 motion (i.e. button 1 down over the actor to select it, move to the new position and button 1 release to drop the actor in place.

Multiple actors can be added in one go, by using button 1 double click on the actor icon. Each subsequent button 1 click will drop an actor on the diagram. A button 1 click on the select icon () will stop adding actors.

The actors name is set in its property panel. First select the actor (if not already selected) on the editing pane using button 1 click. Then click on the `Properties` tab in the details pane. The name is entered in the name field, and will appear on the screen.

As a shortcut, double button 1 click on the name of the actor in the editing pane (or just typing on the keyboard when an actor is selected) will allow the name to be edited directly. This is a convenient way to enter a name for a new actor.

Having created the actor, you will see it appear in the explorer (the upper left quadrant of the user screen). This shows all the model elements created within the UML design. A drop down at the top of the explorer controls the ordering of model elements in the explorer. The most useful are the `Package-centric` (default) and `Diagram-centric`. The latter shows model elements grouped by the diagram on which they appear.

4.4.2. Use Cases

The procedure for adding use cases is the same as that for adding actors, but using the use case icon on the editing pane toolbar ().

By default use cases in ArgoUML do not display their extension points (for use in extend relationships). You can show the extension point compartment in one of two ways.

1. Select the use case in the editing pane with button 1 click, then select the `Style` tab in the details pane and button 1 click on the `Display: Extension Points` check box.

2. Use button 2 click over the use case in the editing pane to display a context-sensitive pop-up menu and from that choose `Show/Show Extension Point Compartment`.

The same approaches can be used to hide the extension point compartment.

4.4.2.1. Adding an Extension Point to a Use Case

There are two ways to add an extension point to a use case.

1. Select the use case on the editing pane with button 1 click. Then click on the `Add Extension Point` icon () on the toolbar, and a new extension point with default name and location will be added after any existing extension points.

   Note

   The `Add Extension Point` icon is grayed out and unusable until a use case is selected.

2. Select the use case on the editing pane with button 1 click and then select its property tab in the details pane. A button 2 click over the `Extension Points:` field will bring up a context-sensitive pop-up menu. Select `Add` to add a new extension point.

   If any extension points already exist, they will be shown in this field on the property tab. The new extension point will be inserted immediately before the entry over which the pop-up menu was invoked. This ordering can be changed later by using the `Move Up` and `Move Down` entries on the pop-up menu.

Whichever method is used, the new extension point is selected, and its property tab can be displayed in the details pane. The name and location of the extension point are free text, set in the corresponding fields of the property tab.

An existing extension point can be edited from its property tab. The property tab can be reached in two ways.

1. If the extension point compartment for the use case is displayed on the diagram, select the use case with button 1 click and then select the extension point with a further button 1 click. The property tab can then be selected in the details pane.

2. Otherwise select the use case and its property tab in the details pane. A button 1 click on the desired entry in the `Extension Points` field will bring up the property tab for the extension point in the details pane.

The name and location fields of the extension point may then be edited.

As a shortcut, where the extension point compartment is displayed, double click on the extension point allows text to be typed in directly. This is parsed to set name and location for the extension point.

Extension points may be deleted, or their ordering changed by using the button 2 pop-up menu over the `Extension Points` field in the use case property tab.

Having created an extension point, it will appear in the explorer (upper left quadrant of the user screen). Extension points are always shown in a sub-tree beneath their owning use case.

4.4.3. Associations

To join a use case to an actor on the diagram use button 1 click on the association icon on the editing pane toolbar ( ). Hold button 1 down at the use case, move to the actor and release button 1 (or alternatively start at the actor and finish at the use case).

This will create a straight line between actor and use case. You can segment the line by holding down button 1 down on the line and moving before releasing. A vertex will be added to the line, which you can move by button 1 motion. A vertex can be removed by picking it up and sliding to one end of the line.

Multiple associations can be added in one go, by using button 1 double click on the association icon. Each subsequent button 1 down/motion/release sequence will join an actor to a use case. Use button 1 on the select icon () to stop adding associations.

It is also possible to add associations using small “handles” that appear to the left and right of a use case or actor when it is selected and the mouse is over it. Dragging the handle from a use case to an actor will create an association to that actor (and similarly by dragging a handle from an actor to a use case).

Dragging a handle from a use case into empty space will create a new actor to go on the other end. Similarly dragging a handle from an actor into empty space will create a new use case.

It is possible to give an association a name, describing the relationship of the actor to the use case, although this is not usually necessary. This is done through the property tab of the association. Such a name appears alongside the association near its center.

4.4.3.1. Setting Navigation

There are two ways of setting the navigation of an association.

1. Use button 2 click on the association to bring up a context-sensitive pop-up menu. The `Navigability` sub-menu has options for bi-directional navigation (the default, with no arrows) and for navigability Actor->Use Case and Use Case->Actor.

2. Use button 1 to select the association and select its property tab in the details pane. This shows a field named `Association Ends:`, with entries for each end labeled by the actor or use case name and its multiplicity. Select the end that should be at the tail of the arrow with button 1 click. This brings up the property tab for the association end. Use button 1 click to uncheck the `Navigability` box.

Note

This may seem counter-intuitive, but in fact associations by default are navigable in both directions (when no arrows are shown). This process is *turning off* navigation at one end, rather than turning it on at the other.

You will see it is possible to give an association end a name in its property tab. This name will appear at that end of the association, and can be used to indicate the *role* being played by an actor or use case in an association.

For example a time management system for a business may have use cases for completing time sheets and for signing off time sheets. An employee actor may be involved in both, one as an employee, but the other in a role as manager.

4.4.3.2. Setting Multiplicity

There are two ways of setting multiplicity at the end of an association.

1. Button 2 click over the end of an association will cause a context-sensitive pop-up menu to appear with a sub-menu labeled `Multiplicity`. This allows you to select from `1` (the default), `0..1`, `0..*` and `1..*`.

2. Bring up the property sheet for the association end as described for setting navigation (see the second option in Section 4.4.3.1, “Setting Navigation”). A drop down menu gives a range of multiplicity options that may be selected.

The second of these two approaches has a wider range of options, although ArgoUML does not currently allow the user to set an arbitrary multiplicity.

4.4.4. Hierarchical Use Cases

UML as originally designed allowed use cases to be organized by grouping them in packages as well as by specifying relations among them. In ArgoUML only the relations mechanism is supported. All Three of the relations that apply to use cases are supported. These are *include*, *extend* and *generalization*.

4.4.4.1. Includes

The procedure for adding an include relationship is the same as that for adding an association, but using the include icon from the editing pane toolbar () to join two use cases.

Since include relationships are directional the order in which the two ends are selected is important. The *including* (main) use case should be selected first (button 1 down) and the *included* (subsidiary) use case second (button 1 release).

It is possible to name include relationships using the property tab, but this is rarely done, and will not be displayed on the use case diagram.

4.4.4.2. Extends

The procedure for adding an extend relationship is the same as that for adding an include relationship, but using the extend icon from the editing pane toolbar () to join two use cases.

As with include relationships, the order of selection matters. In this case, the *extending* (subsidiary) use case should be selected first (button 1 down) and the *extending* (main) use case second (button 1 release).

Note

This is the reverse of the include relationship, but reflects the way that designer's tend to think. The fact that the extend icon's arrow points upward (the opposite of the include icon) should help remind you of this.

To set a condition for the extend relationship, select the extend relationship in the editing pane (button 1 click) and then bring up its property tab in the details pane ((button 1 click on the tab). The text of the condition may be typed in the `Condition` field. Long conditions may be split over several lines if desired. The condition is displayed under the «`extend`» label on the diagram.

It is possible to name extend relationships using the property tab, but this is rarely done, and will not be

displayed on the use case diagram.

4.4.4.3. Generalization

The procedure for adding generalizations, is the same as for adding extend relationships, but using the generalization icon from the editing pane toolbar ().

Since generalization is a directed relationship, the order of selection matters. The specialized use case should be selected first (button 1 down) and the generalized second (button 1 release).

It is also possible to add generalizations using small “handles” that appear to the top and bottom of a use case when it is selected. Dragging the handle at the top to another use case will create a generalization. The original use case is the specializing end, and the use case to which the handle was dragged will be the generalizing end. Dragging into empty space will create a new use case to be the generalizing end.

Similarly dragging on the bottom handle will create a generalization in which the original use case is the *generalizing* end.

Generalization is also permitted between actors, although its use is beyond the scope of this tutorial. Unlike use cases there are no generalization handles on actors, so generalizations must be created using the toolbar icon.

It is possible to name generalization relationships using the property tab, but this is rarely done. If a name is provided, it will be displayed on the use case diagram.

4.4.5. Stereotypes

UML has the concept of *stereotyping* as a way of extending the basic notation. It may prove useful for example to model a problem at both the business level and the engineering level. It is for this reason that the OMG distinguishes between a PIM and a PSM. For both of these we will need use cases, but the use cases at the business level hold a different sort of information to those at the engineering level. Very likely they use different language and notation in their underlying use case specifications.

Stereotypes are used to label UML model elements such as use cases, to indicate that they belong to a certain category. Such labels are shown in guillemots (« ») above the name of the model element on the diagram. The UML standard defines a number of standard stereotypes, and the user may define more stereotypes of his own.

You will see that ArgoUML has a drop down selector, `Stereotype` on every property tab. This is populated with the standard stereotypes, to which you may add your own user defined ones.

The details of stereotyping are beyond the scope of this tutorial. The reference manual (see Section 16.6, “Stereotype”) documents the support provided in ArgoUML.

Warning

ArgoUML is missing a few of the standard UML stereotypes. In addition not all model elements will actually display the stereotype on the diagram. At present this includes use cases and actors.

4.4.6. Documentation

ArgoUML has some simple documentation facilities associated with model elements on a diagram. In general these should be used only to record the location of material in documents that can be handled by a mainstream editor or word processor, not the actual documentation itself.

Documentation for a particular model element is recorded through the documentation tab in the details pane (the quadrant of the user screen at the bottom right).

In addition annotation may be added to diagrams using the text icon on the editing pane toolbar ().

The recommendation is that a use case diagram should use the documentation tab of actors to record information about the actor, or if the actor is complex to refer to a separate document that holds information about the actor.

The documentation tab of use cases should record the location of the use case specification. The information in a use case specification (for all but the simplest use cases) is too complex to be placed directly in the tab.

The project should also have a separate vision document and supplementary requirements specification. A text annotation on diagrams may be used to refer to these if the user finds this helpful.

Warning

The documentation tab includes a `Deprecated` check box. The state of this flag is not preserved over save and load in the current release of ArgoUML

4.4.7. System Boundary Box

ArgoUML provides a series of tools to provide arbitrary graphical annotation on diagrams (we have already mentioned the text tool). These are found at the right hand end of the editing pane toolbar and are fully documented in the reference manual (see Chapter 12, *The Editing Pane*).

The rectangle tool can be used to draw the boundary box. Use the button 2 context-sensitive `Ordering` pop-up menu to place it behind everything else. However there is no way to change its fill color from the default white. You may therefore prefer to draw the boundary box as four lines. This is the method used for the diagrams in this chapter.

Note

The editing pane in ArgoUML has a grid to which objects snap to aid in drawing. The size of this grid and its effect may be altered through the `View` menu (using `Adjust Grid` and `Adjust Grid Snap`). This is described fully in the reference manual (see Chapter 10, *The Menu bar*).

4.5. Case Study

4.5.1. Vision Document

A vision document contains more than those things needed for the modeling effort. It also contains financial and scheduling pertinent information. The following sections are those parts of the Vision Document spelled out in Section 4.3.1, “Vision Document” above. In practice this format need not be followed religiously, but is used here for consistency.

4.5.1.1. Summary

The company wishes to produce and market a line of ATM devices. The purpose of this project is to produce the hardware and the software to drive it that are both maintainable and robust.

4.5.1.2. Goals

To produce better designed products based on newer technology. Follow the MDA philosophy of the OMG by producing first a Platform Independent Model (PIM). As current modeling technology does not admit of maintaining the integrity of the connection between the PIM and Platform Specific Models (PSMs), the PIM will become comparatively stable before the first iteration of the PSM is produced. The software platform will be Java technology. The system will use a simple userid (from ATM card) and password (or PIN) mechanism.

4.5.1.3. Market Context

Equipment currently on the market is based on older technology for both hardware and software. This technology has not reached the end of its useful life, making it unlikely that the vendors of that gear are going to update it in the near future. On the other hand newer technology is available that would put us at a competitive advantage if implemented now.

4.5.1.4. Stakeholders

Among the stakeholders for this system are the Engineering Department, the Maintenance Department, and the Central Computer Facility. The full list of these stakeholders and the specific individuals representing them are.

- *Engineering*. Bunny, Bugs
- *Maintenance*. Hardy, Oliver
- *Computer Facility*. Laurel, Stanley
- *Chief Executive Officer*. Hun, Atilla The
- *Marketing*. Harry, Oil Can

4.5.1.5. Key Features

Cash deposit, cash withdrawal, and account inquiries by customers. Customers include people who have accounts at the owning bank as well as people who wish to make withdrawals from accounts in other banks or from credit card accounts.

Maintenance of the equipment by the bank's engineers. This action may be initiated by the engineer on a routine basis. It may also be initiated by the equipment that can call the engineer when it detects an internal fault.

Unloading of deposits and loading of cash by officials of the local bank branch. These actions occur either on a scheduled basis or when the central computer determines that the cash supply is low or the deposit receptacle is liable to be getting full.

An audit trail for all activities will be maintained and sent periodically to the bank's central computer. It will be possible for the maintenance engineer to save a copy of the audit trail to a diskette for transporting to the central computer.

Both dialup and leased line support will be provided. The ATM will continue to provide services to customers when communication with the central computer is not available.

4.5.1.6. Constraints

The project must be completed within nine months. It must cost no more than 1,750,000 USD excluding production costs. Components may be contracted out, but the basic architecture as well as the infrastructure will be designed in house. Close liaison must be maintained between the software development and the design, development and production of the hardware. Neither the hardware nor the software shall be considered the independent variable, but rather they shall be considered equal.

4.5.1.7. Appendix

The following are the actors that directly support this vision. Additional actors may be identified later that are needed to support this or that technology. They should not be added to this list unless they are deemed to directly support the vision as described in this document.

- Central Computer
- Customer
- Local Branch Official
- Maintenance Engineer

The following are the use cases that directly support this vision. Additional use cases may be identified later that are needed to support this or that technology or to support the use cases listed here. They should not be added to this list unless they are deemed to directly support the vision as described in this document.

- Audit
- Customer Uses Machine
- Maintain Machine

4.5.2. Identifying Actors and Use Cases

For the ATM case study, we will elaborate on the examples in Section 4.3, “Output of the Requirements Capture Process”, Figure 4.4, “Use case diagram for an ATM system showing include relationships.” and Figure 4.5, “Use case diagram for an ATM system showing an extend relationship.”, and progress to identify additional actors and use cases that comprise our model of the ATM system. Figure 4.4, “Use case diagram for an ATM system showing include relationships.” and Figure 4.5, “Use case diagram for an ATM system showing an extend relationship.” exemplified the essential concepts and components of a use case diagram such as, use cases, actors, multiplicity, and include / extend relationships. They showed the relationships between the actors and use cases, and demonstrated how these actors and use cases interact.

In Figure 4.4, “Use case diagram for an ATM system showing include relationships.” we see a use case diagram for an ATM system consisting of «include» relationships for the use cases, Maintain ATM and Use ATM. Maintain ATM was further defined by two use cases, "Maintain Equipment" and "Reload ATM". Use ATM was further defined in terms of the behavior of three simpler use cases: "Deposit Cash", "Withdraw Cash" and "Query Account".
More to be written...

4.5.3. Associations (To be written)

To be written...

4.5.4. Advanced Diagram Features (To be written)

To be written...

4.5.5. Use Case Specifications (To be written)

To be written...

4.5.6. Supplementary Requirements Specification (To be written)

To be written...

Chapter 5. Analysis

Analysis is the process of taking the “customer” requirements and re-casting them in the language of, and from the perspective of, a putative solution.

We are not actually trying the flesh out the detailed solution at this stage. That occurs in the *Design Phase* (see Chapter 6, *Design*).

Unlike the boundary between Requirements and Analysis Phases, the boundary between Analysis and Design Phases is inherently blurred. The key is that analysis should define the solution no further than is necessary to specify the requirements in the language of the solution. The model elements in Analysis generally represent a high level of abstraction.

Once again the *recursive*, and *iterative* nature of our process means we will come back to the Analysis phase many times in the future.

5.1. The Analysis Process

There are three schools of thought on how Analysis should be approached. The ontologist defines the data (actually the metadata) first and worries about processes later. The true ontologist would prefer not to have to think about processes at all. The phenomenonologist reverses this and favors process over data. The panparadigmist considers both process and data to be equally important and addresses both from the start.

When it comes to being a purist the ontologist has the upper hand. It is possible to define and build a database into which data can be entered and retrieved without concern for what happens to it or is done with it. On the other hand implementing a process without having any data structures for it to operate on is not very meaningful.

5.1.1. Class, Responsibilities, and Collaborators (CRC) Cards

The CRC methodology favors the phenomenonologists preference for analysis. It is the equivalent of starting with the use cases, the process aspects (operations) of the class diagrams, and scenarios from which sequence diagrams can be initiated.

CRC cards and the associated methodology are described in detail in Appendix G, *The CRC Card Methodology*. They are used again in the design phase and are further discussed in Chapter 6, *Design*.

The strength of CRC cards during analysis.

- Common Project Vocabulary -
- Spread Domain Knowledge -
- Making the Paradigm Shift -
- Live Prototyping -
- Identifying Holes in Requirements -

In this phase the group should consist of two or three domain experts, one object-oriented technology facilitator, and the rest of the group made up of people who are responsible for delivering the system.

The first time that the Analysis phase occurs a special case of the CRC session happens as there are no classes or scenarios to choose from to define a CRC session. At this point a special type of session known as brainstorming is held. During this session you identify the initial set of classes in the problem domain by using the problem statement or requirements document or whatever you know about the desired result for a starting point. The nouns that are found in whatever you are starting from are a good key to an initial set of classes in the system. In a brainstorming session there should be little or no discussion of the ideas. Record them and filter the results after the brainstorming. At this stage the distinction between class and object is blurred.

Once a reasonable set of classes has been defined by the group, responsibilities can be added. Add responsibilities that are obvious from the requirements or the name of the class. You don't need to find them all (or any for that matter). The scenarios will make them more obvious. The advantage of finding some in the beginning is that it helps provide a starting place.

Select the initial scenarios from the requirements document by examining it's verbs in much the same way that we scanned its nouns earlier. Then as many walk through sessions as necessary to complete the analysis phase are performed.

When is enough of the analysis complete that design can begin? When all the different responsibilities are in place and the system has become stable. After all the normal behavior has been covered, exceptional behavior needs to be simulated. When you notice that the responsibilities are all in place to support the new scenarios, and there is little change to the cards, this is a sign the you are ready to start design.

5.1.2. Concept Diagram (To be written)

To be written...

5.1.3. System Sequence Diagram (To be written)

To be written...

5.1.4. System Statechart Diagram (To be written)

To be written...

5.1.5. Realization Use Case Diagram (To be written)

To be written...

5.1.6. Documents (To be written)

Use Case Specifications and Supplementary Requirements Specifications recast in solution language. To be written...

5.2. Class Diagrams (To be written)

To be written...

5.2.1. The Class Diagram (To be written)

To be written...

5.2.2. Advanced Class Diagrams (To be written)

To be written...

5.2.2.1. Association Classes (To be written)

To be written...

5.3. Creating Class Diagrams in ArgoUML

5.3.1. Classes

Identifying class diagrams from existing materials (Vision, Use Cases etc). To be written...

5.3.1.1. Using the Note Icon in the Tool Bar

Click on your target class. Then click on the note icon. ArgoUML will generate the link automatically.

You can also right click to add a note as well! Be aware that you can add an undefined number of notes to any one class!

Warning

Be aware that your note will not appear in the source code documentation tab.

5.3.2. Associations (To be written)

To be written...

5.3.2.1. Aggregation (To be written)

To be written...

5.3.3. Class Attributes and Operations (To be written)

To be written...

5.3.3.1. Entering Data Into Attributes and Methods Windows

Click directly in the class model element and start typing. Do not use the properties window dialog fields—they are not fully functional and liable to cause you a little frustration.

In fact, it would be interesting to see if you can type stereotypes right into the class attribute box for generating XML diagrams.

5.3.3.2. Class Attributes (To be written)

To be written...

5.3.3.3. Class Operations (To be written)

To be written...

5.3.4. Advanced Class Features (To be written)

5.3.4.1. Association Classes (To be written)

To be written...

5.3.4.2. Stereotypes (To be written)

To be written...

5.4. Sequence Diagrams (To be written)

To be written...

5.4.1. The Sequence Diagram (To be written)

To be written...

5.4.2. Identifying Actions (To be written)

To be written...

5.4.3. Advanced Sequence Diagrams (To be written)

To be written...

5.5. Creating Sequence Diagrams in ArgoUML

5.5.1. Sequence Diagrams

5.5.1.1. Creating a Sequence Diagram

Normally, you can just start a sequence diagram right away. On the `Create Diagram` menu choose `Sequence`.

5.5.2. Actions (To be written)

To be written...

5.5.3. Advanced Sequence Diagrams (To be written)

To be written...

5.6. Statechart Diagrams (To be written)

To be written...

5.6.1. The Statechart Diagram (To be written)

Types of statechart diagram (Moore, Mealy); Hierarchical diagrams. To be written...

5.6.2. Advanced Statechart Diagrams (To be written)

To be written...

5.6.2.1. Hierarchical Statechart Diagrams (To be written)

To be written...

5.7. Creating Statechart Diagrams in ArgoUML

5.7.1. Statechart Diagrams (To be written)

To be written...

5.7.1.1. Creating a Statechart Diagram

Select a class, then you can create a statechart diagram.

5.7.2. States (To be written)

To be written...

5.7.2.1. Editing a Composite State

When editing a composite state, how do you provide do and event for a composite state?

The answer is to select a class, then you can create a statechart diagram.

5.7.3. Transitions (To be written)

To be written...

5.7.4. Actions (To be written)

To be written...

5.7.5. Advanced Statechart Diagrams (To be written)

To be written...

5.7.5.1. Hierarchical Statechart Diagrams (To be written)

To be written...

5.8. Realization Use Cases (To be written)

To be written...

5.9. Creating Realization Use Cases in ArgoUML (To be written)

To be written...

5.10. Case Study (To be written)

Regardless of which methodology you use, at this time you are undoubtedly going to take the problem statement from Section 4.5, “Case Study” and extract the nouns from it. This list should be compacted to contain only those nouns that are expected to result in a class. This effort results in the following.

- Account
- Audit trail
- Bank
- Cash
- Customer

5.10.1. CRC Cards

The project manager convenes a CRC session at which the initial set of classes are to be defined. The facilitator reminds the participants that we are in the analysis phase and are only interested in what needs to be done (at the business level) and are to leave out anything that smacks of how to do it. As a general rule of thumb this means a subset of the nouns from the problem statement (see above). The group starts with a complete list of all of the nouns in the statement, examines each one, and decides which are inappropriate crossing them off the list. Each class is then assigned to one of the participants.

to be continued...

5.10.2. Concept Class Diagrams (To be written)

To be written...

5.10.2.1. Identifying classes (To be written)

To be written...

5.10.2.2. Identifying associations (To be written)

To be written...

5.10.3. System Sequence Diagrams (To be written)

To be written...

5.10.3.1. Identifying actions (To be written)

To be written...

5.10.4. System Statechart Diagrams (To be written)

To be written...

5.10.5. Realization Use Cases (To be written)

To be written...

Chapter 6. Design

We now have the problem we are trying to solve specified in the language of a putative solution. In the Design Phase, we construct all the details of that solution.

The blurred boundary between Analysis and Design is reflected in their use of many of the same UML tools. In this chapter we will mostly be reusing UML technology we have already met once. The big step is casting everything into concrete terms. We move from the abstract concepts of analysis to their concrete realization.

Once again the *recursive*, and *iterative* nature of our process means we will come back to the Design phase many times in the future.

6.1. The Design Process (To be written)

The design process extends the modeling effort beyond the business concerns and into the solutions space. It is during this effort that you decide whether you are going to use Java, C++, J2EE, CORBA, SOAP, Dial up line, internet connection dedicated line, XML, etc. Many of these decisions will impact directly the PSM model, others may only be reflected in the documents produced.

...

6.1.1. Class, Responsibilities, and Collaborators (CRC) Cards

Strength of CRC cards during Design

- Spreading Objet-Oriented Design Expertise
- Design Reviews
- Framework for Implementation
- Informal Notation
- Choice of supporting software components
- Performance Requirements

In this phase developers replace some of the domain experts in the group, but there should always be at least one domain expert in the group.

The focus of the group moves from what is to be done to how to do it. The classes from the solution domain are added to those defined in the analysis phase. Think about what classes are needed to make the system work. Do you need a List class to hold objects? Do you need classes to handle exceptions? Do you need wrapper classes for other subsystems? New classes that are looked for in this part, are classes that support the implementation of the system.

During the design phase the distinction between class and object becomes important. Think about the objects in your scenarios. Who creates the objects? What happens when it is created and destroyed? What is the lifetime of the object vs. the lifetime of the information held be the object?

Now is the time to look at what information the objects hold compared to what is requested from other

classes or computed on the fly. Use the back of the card to record the attributes found for the classes. Break you responsibilities into subresponsibilities and list the subresponsibilities indented under the main responsibilities. Move the collaborators next to the subresponsibilities that use them.

After the Collaborator class on your card list the responsibility of the used class that is used in the collaboration. After the collaborating responsibilities on your cards, list the data passed back by the collaborating object in parenthesis.

Redo the scenarios you did in the analysis phase, but know take into consideration all of the design heuristics discussed. Make up your own scenarios and try them.

6.1.2. Package Diagram (To be written)

To be written...

6.1.3. Realization Class Diagrams (To be written)

To be written...

6.1.4. Sequence Diagrams and Collaboration Diagrams (To be written)

To be written...

6.1.5. Statechart Diagrams and Activity Diagrams (To be written)

To be written...

6.1.6. Deployment Diagram (To be written)

To be written...

6.1.7. Documents (To be written)

System Architecture. To be written...

6.2. Package Diagrams (To be written)

To be written...

6.2.1. The Package Diagram (To be written)

To be written...

6.2.2. Advanced Package Diagrams (To be written)

To be written...

6.2.2.1. Subpackages (To be written)

To be written...

6.2.2.2. Adding DataTypes (To be written)

To be written...

6.2.2.3. Adding Stereotypes (To be written)

To be written...

6.3. Creating Package Diagrams in ArgoUML

6.3.1. Packages

How to work out what goes in packages. To be written...

6.3.1.1. Subpackages (To be written)

To be written...

6.3.2. Relationships between packages (To be written)

To be written...

6.3.2.1. Dependency (To be written)

To be written...

6.3.2.2. Generalization (To be written)

To be written...

6.3.2.3. Realization and Abstraction (To be written)

To be written...

6.3.3. Advanced Package Features (To be written)

To be written...

6.3.3.1. Creating New Datatypes (To be written)

To be written...

6.3.3.2. Creating New Stereotypes (To be written)

To be written...

6.4. More on Class Diagrams (To be written)

To be written...

6.4.1. The Class Diagram (To be written)

To be written...

6.4.1.1. Class Attributes (To be written)

To be written...

6.4.1.2. Class Operations (To be written)

To be written...

6.4.2. Advanced Class Diagrams (To be written)

To be written...

6.4.2.1. Realization and Abstraction (To be written)

To be written...

6.5. More on Class Diagrams in ArgoUML (To

be written)

6.5.1. Classes (To be written)

More on identifying classes from existing materials and use of stereotypes. To be written...

6.5.2. Class Attributes and Operations (To be written)

To be written...

6.5.2.1. Class Attributes (To be written)

To be written...

6.5.2.2. Class Operations (To be written)

To be written...

6.5.3. Advanced Class Features

6.5.3.1. Operations on Interfaces

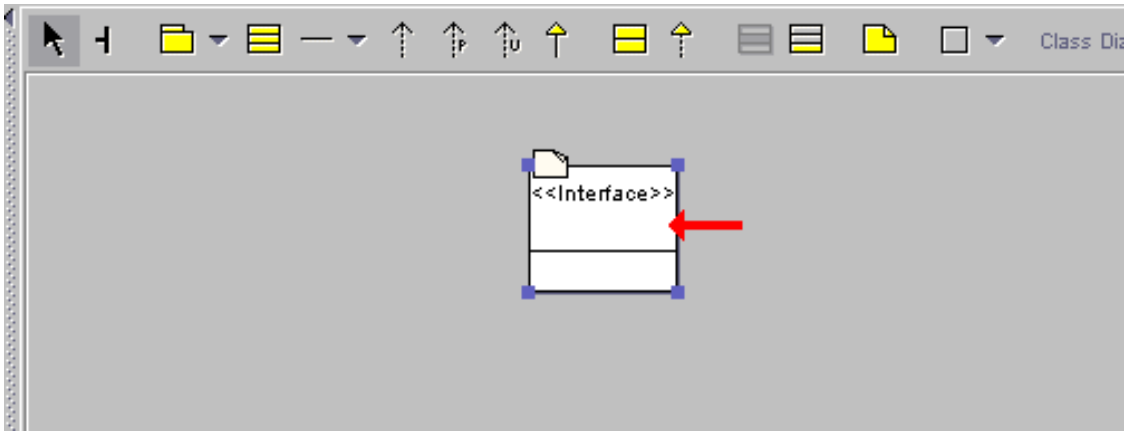
6.5.3.1.1. Interfaces that extend interfaces

Add a unnamed interface to the current classdiagram by single-clicking on the interface icon in the tool bar and then clicking at the diagram pane (see Figure 6.1, “Selecting the Interface tool”).

Figure 6.1. Selecting the Interface tool

Then double click on the interfaces name field to change it's name as shown in Figure 6.2, “Interface model element on the Class Diagram”.

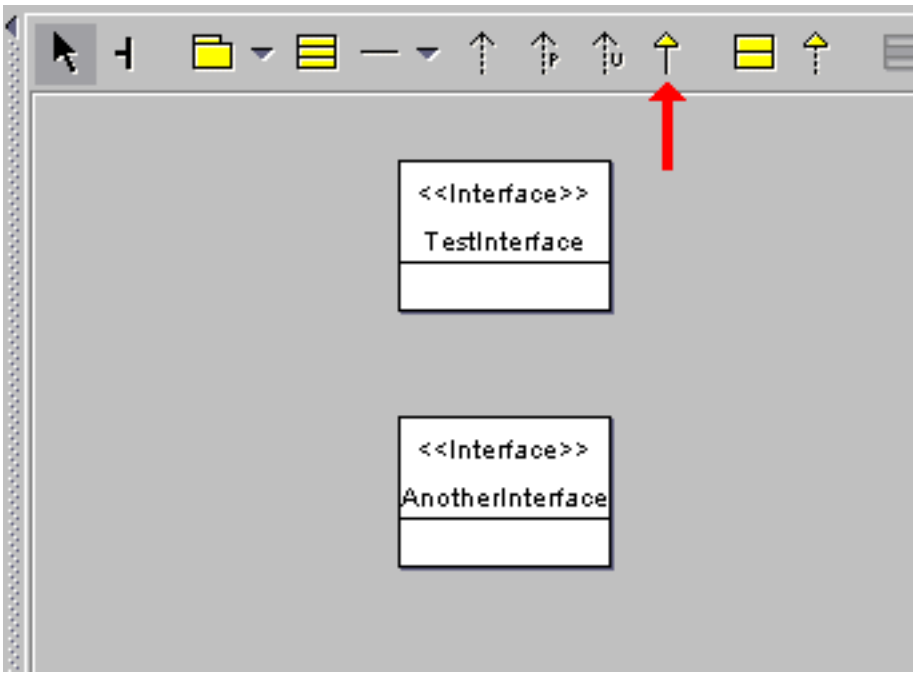
Figure 6.2. Interface model element on the Class Diagram



and type a name for it (like `TestInterface` in this case). Press “Enter” when the name is complete. (You could also enter the name by going to the Properties Tab in the Details Pane after adding the interface.)

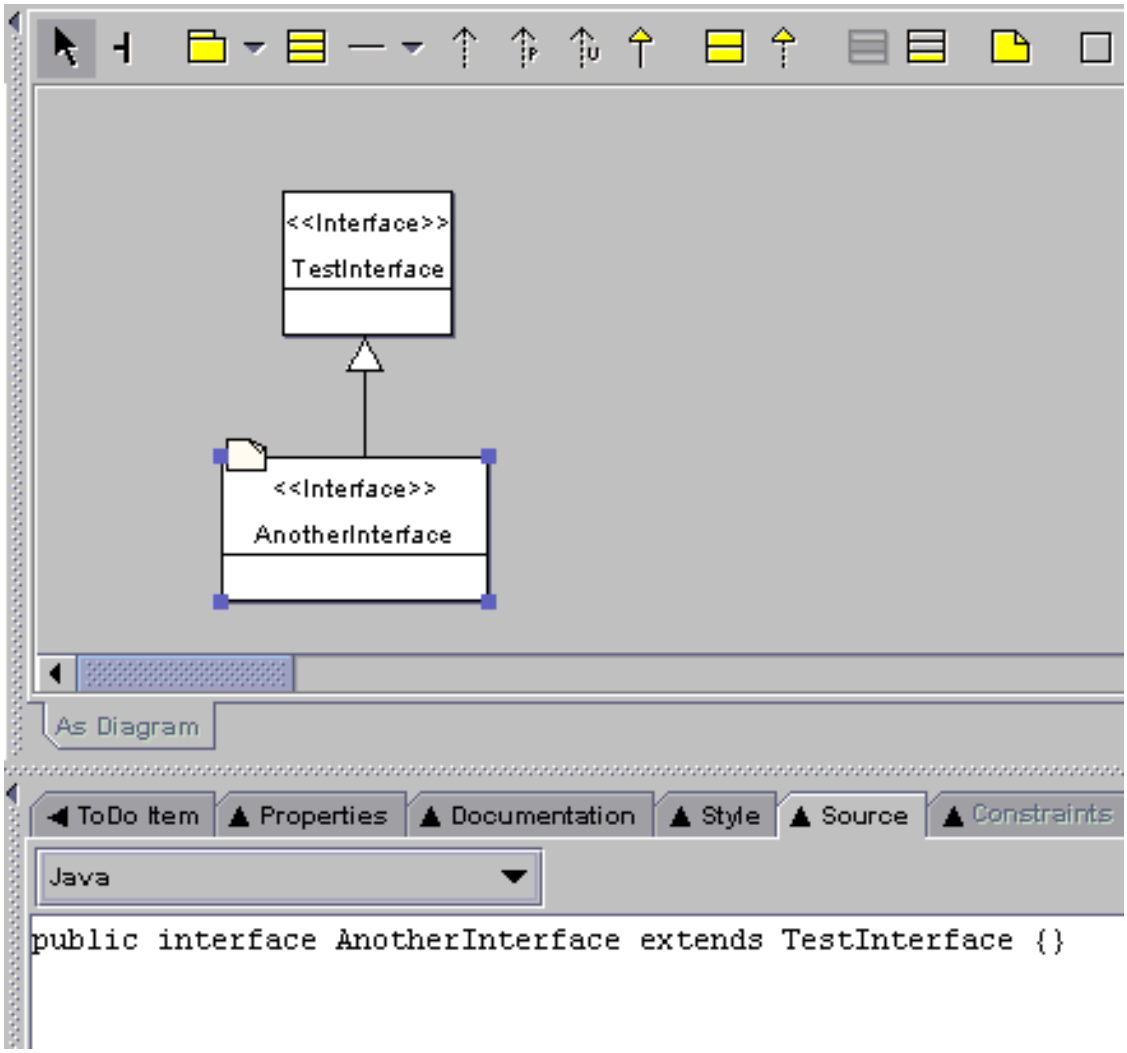
Add another interface with a different by repeating the last 2 steps. Then single-click on the Generalization icon in the tool bar as shown in Figure 6.3, “Generalization on the Class Diagram tool bar”.

Figure 6.3. Generalization on the Class Diagram tool bar



move the mouse pointer to the subinterface, press the left mouse button and drag the generalization to the superinterface, where you release the mouse button. Figure Figure 6.4, “Generalization between two Interfaces.” shows how your diagram should look now.

Figure 6.4. Generalization between two Interfaces.



By clicking on the subinterface and the source tab properties pane, and then selecting Java Notation for the source tab, you can see that the interface actually extends it's superinterface.

6.5.3.2. Stereotypes (To be written)

To be written...

6.6. Sequence and Collaboration Diagrams (To be written)

Note

Sequence diagrams does not work in ArgoUML version 0.14.

To be written...

6.6.1. More on the Sequence Diagram (To be written)

To be written...

6.6.2. The Collaboration Diagram (To be written)

To be written...

6.6.2.1. Messages (To be written)

To be written...

6.6.2.2. Actions (To be written)

To be written...

6.6.3. Advanced Collaboration Diagrams (To be written)

To be written...

6.7. Creating Collaboration Diagrams in ArgoUML (To be written)

6.7.1. Collaboration Diagrams (To be written)

To be written...

6.7.2. Messages (To be written)

To be written...

6.7.2.1. Actions (To be written)

To be written...

6.7.3. Advanced Collaboration Diagrams (To be written)

To be written...

6.8. Statechart Diagrams (To be written)

To be written...

6.8.1. The Statechart Diagram (To be written)

More on this. To be written...

6.8.2. Advanced Statechart Diagrams (To be written)

To be written...

6.8.2.1. Actions (To be written)

To be written...

6.8.2.2. Transitions (To be written)

To be written...

6.8.2.2.1. Triggers (To be written)

To be written...

6.8.2.2.2. Guards (To be written)

To be written...

6.8.2.2.3. Effectss (To be written)

To be written...

6.8.2.3. Pseudo States (To be written)

To be written...

6.8.2.3.1. Junction and Choice (To be written)

To be written...

6.8.2.3.2. Fork and Join (To be written)

To be written...

6.8.2.4. Hierarchical State Machines (To be written)

To be written...

6.8.2.5. Models for State History (To be written)

Shallow v Deep. To be written...

6.9. Creating Statechart Diagrams in ArgoUML (To be written)

6.9.1. Statechart Diagrams (To be written)

To be written...

6.9.2. States (To be written)

To be written...

6.9.3. Transitions (To be written)

To be written...

6.9.4. Actions (To be written)

To be written...

6.9.5. Advanced Statechart Diagrams (To be written)

To be written...

6.9.5.1. Transitions (To be written)

To be written...

6.9.5.1.1. Triggers (To be written)

To be written...

6.9.5.1.2. Guards (To be written)

To be written...

6.9.5.1.3. Effectss (To be written)

To be written...

6.9.5.2. Pseudo States (To be written)

To be written...

6.9.5.2.1. Junction and Choice (To be written)

To be written...

6.9.5.2.2. Fork and Join (To be written)

To be written...

6.9.5.3. Hierarchical State Machines (To be written)

To be written...

6.9.5.4. History (To be written)

Shallow v Deep. To be written...

6.10. Activity Diagrams (To be written)

To be written...

6.10.1. The Activity Diagram (To be written)

More on this. To be written...

6.10.1.1. Action States (To be written)

To be written...

6.11. Creating Activity Diagrams in ArgoUML (To be written)

6.11.1. Activity Diagrams (To be written)

To be written...

6.11.1.1. Creating an Activity Diagram

Select a use case or class, then you can create an activity diagram.

6.11.2. Action States (To be written)

To be written...

6.12. Deployment Diagrams (To be written)

To be written...

6.12.1. The Deployment Diagram (To be written)

To be written...

6.13. Creating Deployment Diagrams in ArgoUML (To be written)

6.13.1. Nodes (To be written)

To be written...

6.13.1.1. Node Instances (To be written)

To be written...

6.13.2. Components (To be written)

To be written...

6.13.2.1. Component Instances (To be written)

To be written...

6.13.3. Relationships between nodes and components (To be written)

To be written...

6.13.3.1. Dependency (To be written)

To be written...

6.13.3.2. Associations (To be written)

To be written...

6.13.3.3. Links (To be written)

To be written...

6.14. System Architecture (To be written)

To be written...

6.15. Case Study (To be written)

6.15.1. CRC Cards (To be written)

To be written...

6.15.2. Packages (To be written)

To be written...

6.15.2.1. Identifying Packages (To be written)

To be written...

6.15.2.2. Datatypes and Stereotypes (To be written)

To be written...

6.15.3. Class Diagrams (To be written)

To be written...

6.15.3.1. Identifying classes (To be written)

To be written...

6.15.3.2. Identifying associations (To be written)

To be written...

6.15.3.3. Specifying Attributes and Operations (To be written)

To be written...

6.15.4. Sequence Diagrams (To be written)

To be written...

6.15.4.1. Identifying actions (To be written)

To be written...

6.15.5. Collaboration Diagrams (To be written)

To be written...

6.15.5.1. Identifying Messages (To be written)

To be written...

6.15.6. Statechart Diagrams (To be written)

To be written...

6.15.7. Activity Diagrams (To be written)

To be written...

6.15.8. The Deployment Diagram (To be written)

To be written...

6.15.9. The System Architecture (To be written)

To be written...

Chapter 7. Code Generation, Reverse Engineering, and Round Trip Engineering

7.1. Introduction

We now have our design fully specified. With the right simulator we could actually execute the design and see if it works. (ArgoUML does not provide such functionality, but this functionality has been provided in alternative tools.)

ArgoUML does allow you to generate code from the design in several different programming languages. We, most likely, already in the design had a programming language in mind because some of the design considerations are to care for a specific language.

The output of this process is the set of files that constitute the program that solves the problem.

Once again the *recursive*, and *iterative* nature of our process means we will come back to the Build phase many times in the future.

There is also another side to this and that is the reverse engineering side. If we happen to have an old program that we would like to examine then we could take the files and reverse engineer them to create a design. This can be used when trying to understand some not so well documented program or as a quick start for the design work.

The process of going back and forth between doing changes in the design followed by a code generation and then doing changes in the code followed by a reverse engineering using for every change, the best possible perspective, is called Round-trip Engineering.

7.2. Code Generation

The output of the Code Generation is the completed program. Depending on the contents of the design, we could also generate Unit test cases.

To do the work we need the design model, containing both static and dynamic descriptions of the program.

7.2.1. Generating Code from the Static Structure

It is rather straightforward to do this generation, at least as long as we do it for an object-oriented language. This is some of the basic rules:

- A class will become a class.

  In some target languages (like java, c++) they also become files and compilation units.

- A generalization will become an inheritance.

  If the target language does not support inheritance and we didn't address this during the design, some special conversions are required to solve this.

- An attribute will become a member variable.
- A navigable association will become a member variable.

  Depending on the target language, target platform, and the association multiplicities this will be a pointer, a reference, a collection class, an entry in some table or map.
- A non-abstract operation in a class will become a method.
- An abstract operation in a class will become an abstract method.
- An in parameter in an operation will become a parameter in the method.

  For simple types (int, boolean), this is the normal case. For C++, these will probably const classes. For Java, this cannot be enforced for classes.
- An out or in/out parameter in an operation will become a referenced parameter in the method.

  For C++, these will be referenced non-const parameters. For Java classes, this is the default. Simple types (int, boolean) must, in java, be converted to an object of a corresponding class (Integer, Boolean).
- The visibilities of the attributes, associations, and operations will become visibilities on the member variables or methods.
- Packages will become directories, namespaces, or both.

7.2.2. Generating code from interactions and state machines

This conversion is not as straight-forward as the conversion of the static structure. It is much more depending on the target language and target platform.

In general it is only possible to say the following for interactions:

- A message is converted into a function call.

  The class of the recipient will have to have a function with the correct name and signature.

  The sender function in the class of the sender will have a call to the function in the recipient.
- An asynchronous message is converted to either posting a message to be handled by some other thread or a function call to a function that starts a new thread.

The following describes one possible way to generate state machines:

- A State Machine is generated to a set of member variables that each method in this class refer to when deciding behavior.
- A State is generated to a closed set of combination of values on these member variables.
- An Event is generated as a call to a member method that can change the state.

  These methods would then typically have one big switch statement splitting on the current state.

- A Guard is generated to an if statement in the event member method in the branch for the correct state.
- A Transition is generated as an assignment of some state variable.
- An Action is generated as a function call.

7.3. Code Generation in ArgoUML

7.3.1. Static Structure

Most of the generation can be done automatically by the provided language modules. Files are generated in a directory hierarchy that need to be filled in by the actual code.

7.3.2. Interactions and statechart diagrams

There is currently no support for this in ArgoUML, not for any language.

7.4. Reverse Engineering

Reverse Engineering is used for two main purposes:

1. To get previously developed classed into the model to build upon.
2. To get a UML view of previously developed classes to understand how they work.

Essentially this does the opposite of Code Generation.

7.5. Round-Trip Engineering

Round-Trip Engineering makes it possible to switch perspective while doing the design. Create some classes in a class diagram. Write some code for some of the operations or functions using your favorite editor. Move the operations from one class to another in the class diagram...

ArgoUML currently does not support this for any language.

Part 2. User Interface Reference

Chapter 8. Introduction

This chapter describes the overall behavior of the user interface. Description of the various component parts—the menu bar, panes and various diagrams— is in separate chapters.

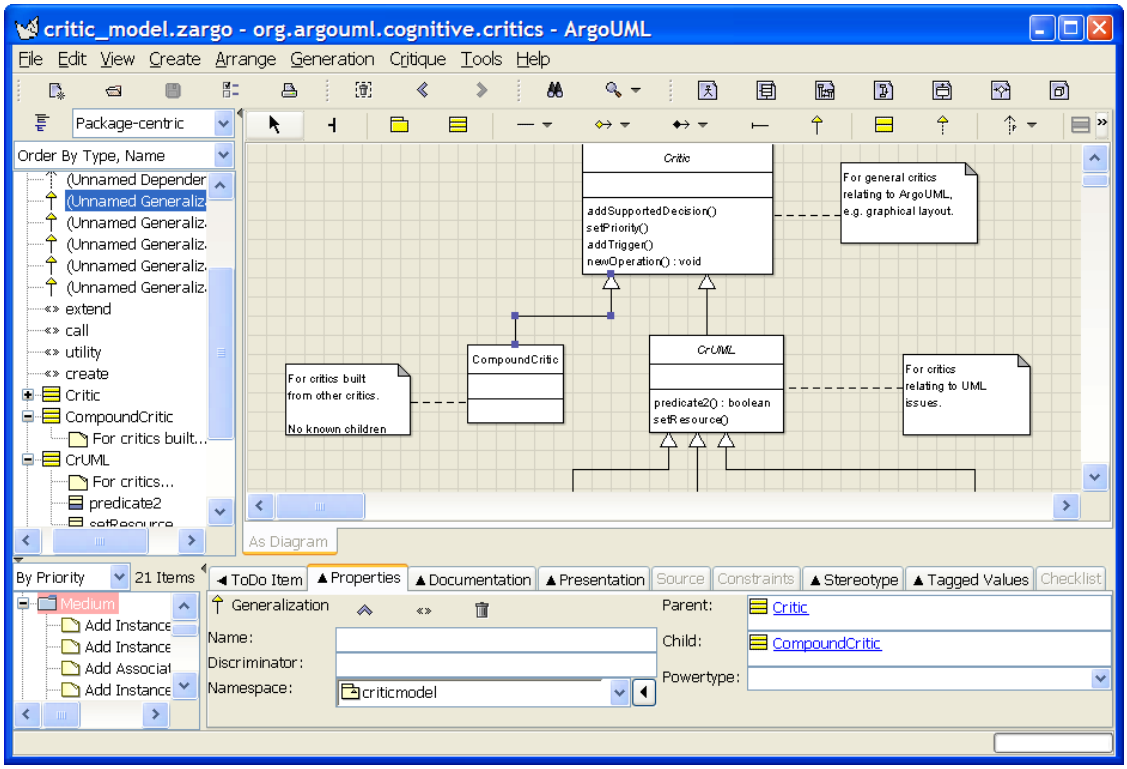
8.1. Overview of the Window

Figure 8.1, “Overview of the ArgoUML window” shows the main ArgoUML window.

The titlebar of the window shows the following 4 parts of information, separated from each other by a dash.

- The current filename. If no filename for the project is set yet, then the titlebar shows "Unititled".
- The name of the currently active diagram.
- The name “ArgoUML”.
- An asterisk (*). This item is only present if the current project file is “dirty”, i.e. it is altered, but not yet saved. In other words, if the asterisk is absent, then the current file has not been altered.

Figure 8.1. Overview of the ArgoUML window



At the top of screen is a *menu bar*, which is described in Chapter 10, *The Menu bar*. Below that is the *toolbar*, as described in Chapter 9, *The Toolbar*.

The bulk of the window comprises four sub-windows or *panes*. Clockwise from top left these are the *explorer* (see Chapter 11, *The Explorer*), *editing pane* (see Chapter 12, *The Editing Pane*), *details pane* (see Chapter 13, *The Details Pane*) and *to-do pane* (see Chapter 14, *The To-Do Pane*). All 4 panes have a *tool bar* at the top (in the *details pane* it is located under the *properties* tab). An overview of the panes is given in Section 8.3, “General Information About Panes”. Finally at the bottom of the window is a *status bar* described in Section 8.4, “The status bar”.

8.2. General Mouse Behavior in ArgoUML

Mouse behavior that is specific to the various panes of ArgoUML (see Section 8.3, “General Information About Panes”) or the menu bar, is discussed in the chapters covering those panes and the menu bar. In this section we cover behavior that is general across all of ArgoUML.

In a number of places in ArgoUML text may be directly edited (for example the constraint editor—see Section 13.7.1, “The Constraint Editor”). The behavior of the mouse when handling text is discussed in the sections that follow.

8.2.1. Mouse Button Terminology

ArgoUML assumes a two button mouse. We will refer to the buttons as “button 1” and “button 2”. Button 1 is the leftmost button on a right-handed mouse, and sometimes referred to as the *select* button. Button 2 is the rightmost button on a right-handed mouse, and is sometimes referred to as the *adjust* button.

A single depress and release of a mouse button with the mouse is referred to as a *click*. Two clicks in quick succession is referred to as a *double click*. Moving the mouse while holding a button down is referred to as *button motion* with the starting point being at *button down* and the end point at *button up*.

8.2.2. Button 1 Click

Clicking on an user-interface object or on a diagram model element may establish many different things. Most of the behaviour is experienced quite intuitive by the user, mainly because the high degree of standardisation, even spanning different computer platforms (Macintosh, PC, UNIX,...). ArgoUML follows the *Java Look and Feel Design Guidelines* by Sun. See http://java.sun.com/products/jlf/. Hence, behaviour of common user-interface components is generally not discussed in this document.

On the other hand, mouse actions in a diagram may not seem so intuitive to the user, since it is specific for ArgoUML. Hence they are explained here. In short, clicking selects or activates the object beneath the mouse-pointer, and moves the focus (i.e. navigation).

More in detail, the button 1 click may cause the following result:

8.2.2.1. Selection

Here button 1 is used to choose (select) a model element (in a list or tree or on a diagram) on which subsequent operations will take place. Multiple model elements may be selected by using Shift and/or Ctrl in combination with button 1, see Section 8.2.5, “Shift and Ctrl modifiers with Button 1”. Selection is always clearly indicated by a colored background.

On a diagram, the selected model element is indicated with colored "blocks" at the corners/ends of the object. Model elements can be selected or deselected in different ways:

- Button 1 click. Deselects all model elements, and selects the one clicked on.
- Button 1 motion. Button motion (moving the mouse with the button down) in the diagram, not on any model element, allows to draw a rectangle around model elements which will be selected when

the button 1 is released.

- Menu functions and shortcuts. Many menu operations change selection as side-effect, e.g. creating a new diagram. Many keyboard shortcuts for menu operations change the selection, e.g. Ctrl-A, which stands for the `Select All` function.

8.2.2.2. Activation

Here button 1 is used to activate the user interface component, e.g. a button. The object is usually highlighted when the mouse button is pressed and then activated when the mouse button is released. Activating an user-interface object means that its function is executed.

8.2.2.3. Navigation

Here button 1 is used to move the focus from one user interface component or diagram model element to another. It is better known under the term keyboard focus. This because keyboard commands usually work on the model element that has the focus. The focus is indicated by a (hardly visible) box around the model element, or for a text entry box, by a flashing cursor.

8.2.2.4. General Behavior When Editing Text

Here button 1 is used to select the point within the text at which operations (text entry and deletion) will take place.

8.2.3. Button 1 Double Click

The behavior of button 1 double click varies betweens panes and is discussed in their chapters.

8.2.3.1. General Behavior When Editing Text

Here button 1 double click is used to select a complete word, or other syntactic unit within the text. Subsequent operations (text entry and deletion) will replace the selected text.

8.2.4. Button 1 Motion

8.2.4.1. General Behavior When Editing Text

Here button 1 motion is used to select a range of text. Subsequent operations (text entry and deletion) will replace the selected text.

8.2.5. Shift and Ctrl modifiers with Button 1

8.2.5.1. Within Lists

This behavior applies where there is a list of things that may be selected. This includes various dialog boxes, and the to-do pane, where there is a list of to-do items to be selected.

Where selections are to be made, the SHIFT key is used to with button 1 to *extend* from the original button 1 selection to the current position.

Similarly the CTRL key with button 1 is used to add individual items to the current selection. Where Ctrl-button 1 is used on an item already selected, that item is removed from the selection.

Caution

Users of Microsoft Windows might be familiar with the use of SHIFT-CTRL-Click (i.e. holding both the Shift and Ctrl key down when clicking), to add sub-lists to an existing selection. ArgoUML does not support this. SHIFT-CTRL-Click will behave as CTRL-Click.

8.2.5.2. General Behavior When Editing Text

In a number of places in ArgoUML text may be directly edited (for example when naming a model—element in the properties pane, or when typing a UML note / comment). Here SHIFT button 1 is used to select a range of text from the previously selected point. Subsequent operations (text entry and deletion) will replace the selected text.

8.2.6. Alt with Button 1: Panning

When holding down the Alt key during button 1 down on a diagram, movement of the mouse pans the drawing area. The function is indicated by the mousepointer which turns into a crosshair with arrows.

8.2.7. Ctrl with Button 1: Constrained Drag

When holding down the Ctrl key while dragging with mouse button 1 down on a diagram, the movement of the dragged element element will be constrained to one of eight cardinal directions : North, South, East, West, NE, SE, SW, NW.

8.2.8. Button 2 Actions

Button 2 actions are all dependent on the pane or menu bar, and discussed in their various chapters.

8.2.9. Button 2 Double Click

Button 2 actions are all dependent on the pane or menu bar, and discussed in their various chapters.

8.2.10. Button 2 Motion

Button 2 actions are all dependent on the pane or menu bar, and discussed in their various chapters.

8.3. General Information About Panes

The four sub-windows of the main ArgoUML window are called *panes*. Clockwise from top left these are the *explorer* (see Chapter 11, *The Explorer*), *editing pane* (see Chapter 12, *The Editing Pane*), *details pane* (see Chapter 13, *The Details Pane*) and *to-do pane* (see Chapter 14, *The To-Do Pane*). At the top the editing pane is a *tool bar*.

8.3.1. Re-sizing Panes

You can re-size panes by dragging on the divider bars between them. To indicate this possibility, the mouse cursor changes shape when hovering over the divider bars.

In addition you will see there are two small left pointing arrows within the vertical divider bars, one at the top of the vertical divider bar between explorer and editing pane and one at the top of the vertical divider bar between to-do pane and details pane. Button 1 click on the first of these will expand the editing pane to the full width of the window, button 1 click on the second will expand the details pane to the full

width of the window.

There is also a small downward pointing arrow within the horizontal divider bar at its leftmost end. Clicking on this will expand the explorer and editing panes to the full depth of the window.

By using both the top arrow on the vertical divider and the arrow on the horizontal divider, it is possible to expand the editing pane to use the entire window.

The original configuration can be restored by clicking again on these arrows, which are now located at the edge of the window.

8.4. The status bar

The status bar is at the very bottom of the ArgoUML window and is used to display short advisory messages. In general such messages are self explanatory. It is e.g. used for displaying parsing error messages in case a text entered on the diagram can not be interpreted.

Chapter 9. The Toolbar

9.1. File operations

These buttons have identical functions as their counterparts in the `File` menu.

- `New` See for a full description Section 10.3.1, “ New”.
- `Open Project...` See for a full description Section 10.3.2, “ Open Project...” .
- `Save Project` See for a full description Section 10.3.3, “ Save Project”.
- `Print` See for a full description Section 10.3.10, “ Print...”.

9.2. Edit operations

These buttons have identical functions as their counterparts in the `Edit` menu.

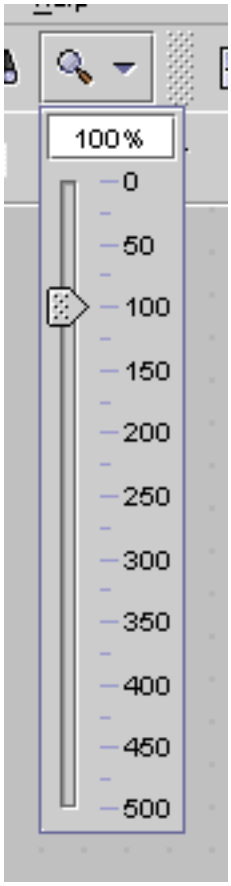
- `Remove From Diagram` See for a full description Section 10.4.2, “ Remove From Diagram”.
- `Navigate Back` See for a full description Section 10.4.1, “Select”.
- `Navigate Forward` See for a full description Section 10.4.1, “Select”.

9.3. View operations

The `Find...` button has identical behaviour as its counterpart in the `View` menu. The `Zoom` button is a more luxurously version of the function in the `View` menu.

- `Find...` See for a full description Section 10.5.2, “ Find...”.
- `Zoom` This is a different version of the menu-item for zooming, as described in Section 10.5.3, “Zoom” . Clicking with button 1 on the zoom-icon opens a panel as in the figure below.

Figure 9.1. The Zoom slider on the Toolbar



Once the panel is open, the following actions are possible:

- Clicking with button 1 on the "knob" followed by button 1 movement will adjust the zoomfactor.
- Clicking with button 1 on the shown percentage allows editing the given zoomfactor (in percent) directly with the keyboard. Double clicking on the value shown selects the whole entry for easy overtyping.
- Clicking with button 1 below or above the knob increases or decreass the zoom factor with 1%. Use this function to easily fine-adjust the percentage.
- Clicking with button 1 or button 2 on the `Zoom` tool, or anywhere outside the slider panel closes the panel.
- The keyboard can be used to operate the Zoom Slider as follows: When the `Zoom` icon in the toolbar has the focus (indicated by the thin blue box around it), then pressing the **spacebar** opens the zoon slider panel. Use the **arrow** keys to increase and decrease the percentage 1 by 1. Use **Shift-Tab** to set the focus to the percentage box, where you can edit the given value directly. Pressing **Enter** activates the changed value. When the "knob" has the focus, pressing **PageUp**/**PageDown** increases/decreases the percentage by 50. Pressing **Home** sets the percentage to 500%, and **End** sets it to 0%.

9.4. Create operations

These buttons have identical functions as their counterparts in the `Create` menu.

-  `New Use Case Diagram` See for a full description Section 10.6.1, “  New Use Case Diagram” .
-  `New Class Diagram` See for a full description Section 10.6.2, “  New Class Diagram” .
-  `New Sequence Diagram` See for a full description Section 10.6.3, “  New Sequence Diagram” .
-  `New Collaboration Diagram` See for a full description Section 10.6.4, “  New Collaboration Diagram” .
-  `New Statechart Diagram` See for a full description Section 10.6.5, “  New Statechart Diagram” .
-  `New Activity Diagram` See for a full description Section 10.6.6, “  New Activity Diagram” .
-  `New Deployment Diagram` See for a full description Section 10.6.7, “  New Deployment Diagram” .

Chapter 10. The Menu bar

10.1. Introduction

An important principle behind ArgoUML is that actions should be able to be invoked in whatever way the user finds convenient. As a result many (but not all) actions that can be carried out on the menu can be carried out in other ways as well under ArgoUML.

A number of the common menu entries are also available through keyboard shortcuts.

It is also be possible to navigate the menu from the keyboard. Each level of each menu is identified by a letter (shown underlined in the menu or entry name from the moment the ALT key is pressed). This sequence of letters while holding down the ALT key selects the entry.

The following is an explanation of why the menuitems are grouped as they are.

- The *File* menu contains operations that affect on the whole project/file. All the items in this menu can be explained as such.
- The *Edit* menu is generally intended for editing the model or changing the content of a diagram. It also contains functions to enable editing, like e.g. selecting. This menu is not intended for diagram layout functions. Most functions here do something with the selected modelelement and diagram. The items "Configure Perspectives..." and "Settings..." are a bit different, since they adjust the way ArgoUML works - but they do not belong in the File menu, since their settings are not stored in the project.
- The *View* menu is for functions that do never alter the model, nor the diagram layout, only the way you view the diagram. A good example is "zoom". Also navigational functions belong here, e.g. "Find" and "Goto Diagram...". All changes of settings in this menu apply to all diagrams (e.g. zoom).
- The *Create* menu contains all possible diagrams that can be created. These functions are context dependend, since they work on the selected modelelement.
- The *Arrange* menu allows layout changes in the current diagram, which is not the same as the items in the View menu. Functions here can not alter the UML model.
- The *Generation* menu is for Code Generation. The functions here work either on the selected modelelements, or on the whole project.
- The *Critique* menu is specific for settings related to critics, which apply for all projects.
- The *Tools* menu is currently empty. If plugins are installed, then their functions appear here.
- The *Help* menu contains the usual "information" and "about".

10.2. Mouse Behavior in the Menu Bar

Behavior of the mouse in general, and the naming of the buttons is covered in the chapter on the overall user interface (see Section 8.2, “General Mouse Behavior in ArgoUML”). There is no ArgoUML specific behaviour for the menu.

10.3. The File Menu

These are actions concerned with input and output and the overall management of projects and the ArgoUML system.

10.3.1. New

Shortcut Ctrl-N.

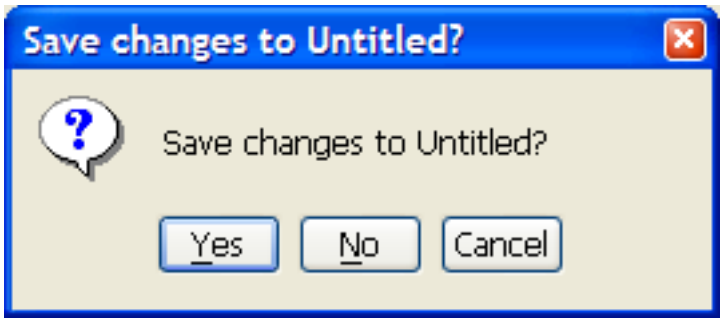
This initializes a new project within ArgoUML. The project is created without a the name. It contains a (top-level) `Model` named `untitledModel` and two empty diagrams: a class diagram and a use case diagram.

Caution

`untitledModel` is not a conventional model name (most processes suggest models should be build from lower case letters). ArgoUML permits you to use any case letters, but a critic will trigger to warn that this is not conventional. See Section 16.2, “The Model” for a discussion of this.

If the model has been altered (as indicated by the "*" in the titlebar of ArgoUML's window), then activating the "New" function is potentionally not the user's intention, since it will erase the changes. Hence a confirmation dialog appears to allow the user to save his work first, or cancel the operation completely.

Figure 10.1. The confirmation dialog for `New`.

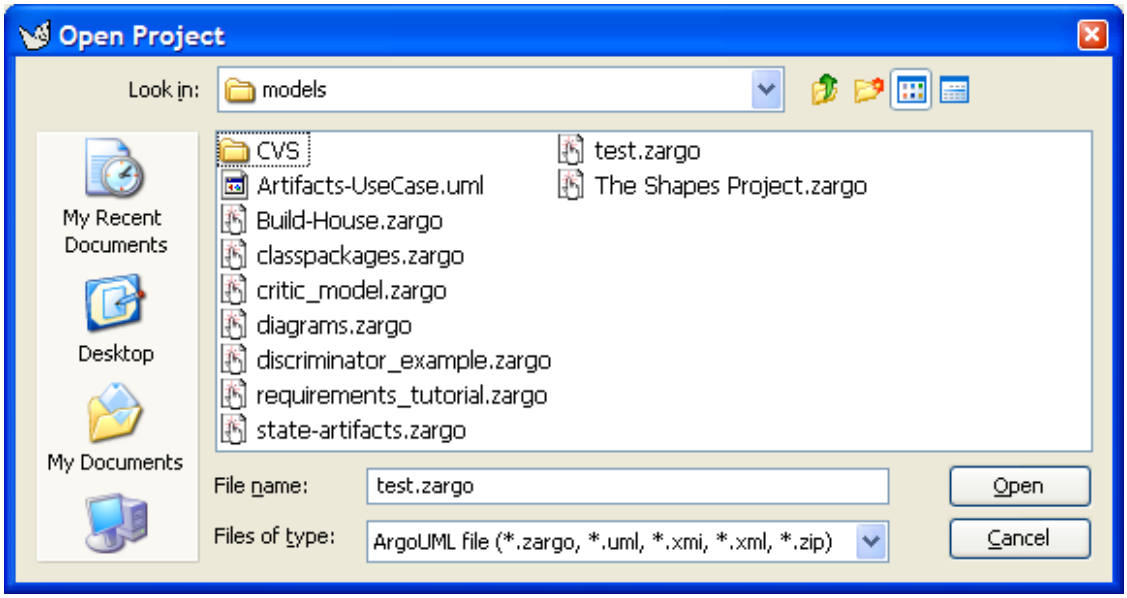


10.3.2. Open Project...

Shortcut Ctrl-O.

This opens an existing project from a file. Selecting this menu option will open a file selection dialog (see Figure 10.2, “The file selection dialog for `Open Project...`”).

Figure 10.2. The file selection dialog for `Open Project...`



The main body of the dialog is a text area with a listing of all directories and files in the currently selected directory which match the current filter (see below).

Navigating in the directory tree is possible by selecting a directory in the drop down selector at the top of this dialog. Navigating deeper in the tree may be done by double clicking button 1 on the directory shown in the main text area.

In the lower portion of the dialog is a text box labeled `File name:` for the name of the file to be opened. The file name may be typed directly in here, or selected from the directory listing above using button 1 click.

Beneath this is a drop down selector labeled `Files of type:` to specify a filter on the files to be shown in the directory listing. Only files that match the filter are listed. The available filters are listed below. The default filter is the first one, which combines all available formats.

- ArgoUML file (*.zargo, *.uml, *.xmi, *.xml, *.zip)
- ArgoUML compressed project file (*.zargo)
- ArgoUML project file (*.uml)
- XML Metadata Interchange (*.xmi)
- XML Metadata Interchange (*.xml)
- XMI compressed project file (*.zip)

10.3.3. Save Project

Shortcut Ctrl-S.

This saves the project using its current file name. Use `Save Project As...` to save the project to a different file. If no filename is given yet (e.g. after `New`), then this function works exactly as `Save Project As...`.

Note

In certain circumstances, there is nothing to save, and this menuitem is downlighted. E.g. when the user did not yet alter a loaded project. The presence of a “*” in the titlebar of ArgoUML's window indicates that the current project is “dirty” (has been altered), and can be saved.

10.3.4. Save Project As...

This opens a dialog allowing you to save the project under a different file name (or to specify a file name for the first time if the project is a new project).

The dialog box is almost identical to that for `Open Project` (see Figure 10.2, “The file selection dialog for `Open Project...`”). The extension of the filename is automatically set.

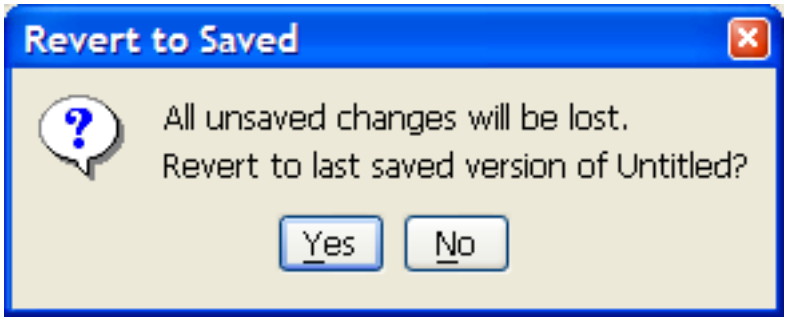
10.3.5. Revert to Saved

This menu-item allows you to throw away all your recent changes, and reload the last saved version of the current project. It works a bit like an `Undo` feature, but only restores changes done since the last time the file was saved.

This menu-item is downlighted unless the currentproject has been saved or loaded before (i.e. it has a name), and it has been altered.

When this menu-item is activated, a small confirmation dialog box opens, as shown in the figure below. This warning that all recent changes will be discarded, is needed because the action can not be undone. Selecting `No` cancels the whole action as if you did not select the menu-item in the first place. Selecting `Yes` reloads the last saved file.

Figure 10.3. The warning dialog for Revert to Saved.

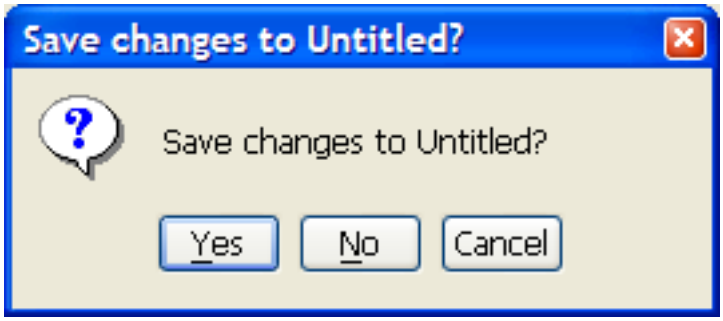


10.3.6. Import XMI...

This menu-item allows to load an UML 1.3 or 1.4 model which was exported by e.g. another tool, as a XMI file, according the XMI V1.0, V1.1 or V1.2 standard. The extension of such file should be `.xmi`.

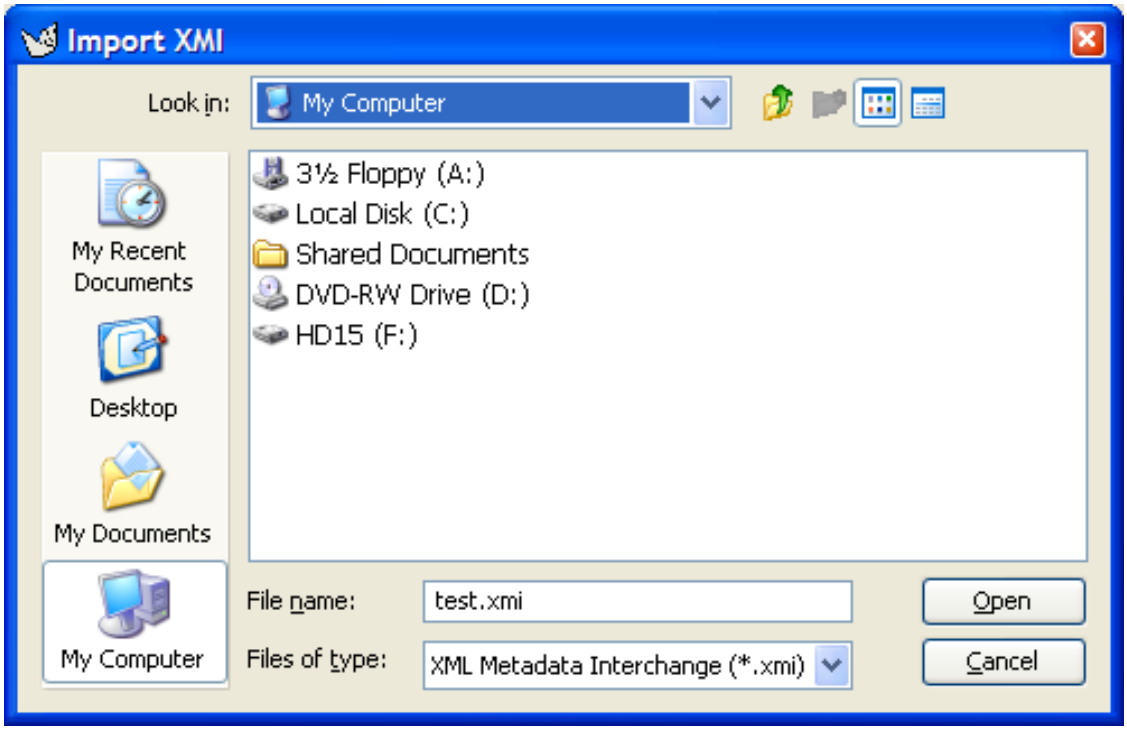
If the model has been altered (as indicated by the "*" in the titlebar of ArgoUML's window), then activating the "Import XMI..." function is potentionally not the user's intention, since it will erase the changes. Hence a confirmation dialog appears to allow the user to save his work first, or cancel the operation completely.

Figure 10.4. The confirmation dialog for `Import XMI...`



When the menu is activated, the standard filechooser appears, see Figure 10.5, “The dialog for `Import XMI...`”. Beware the fact that this file will only contain the model, not any diagram layout. Hence, the new project will not contain any diagrams.

Figure 10.5. The dialog for `Import XMI...`



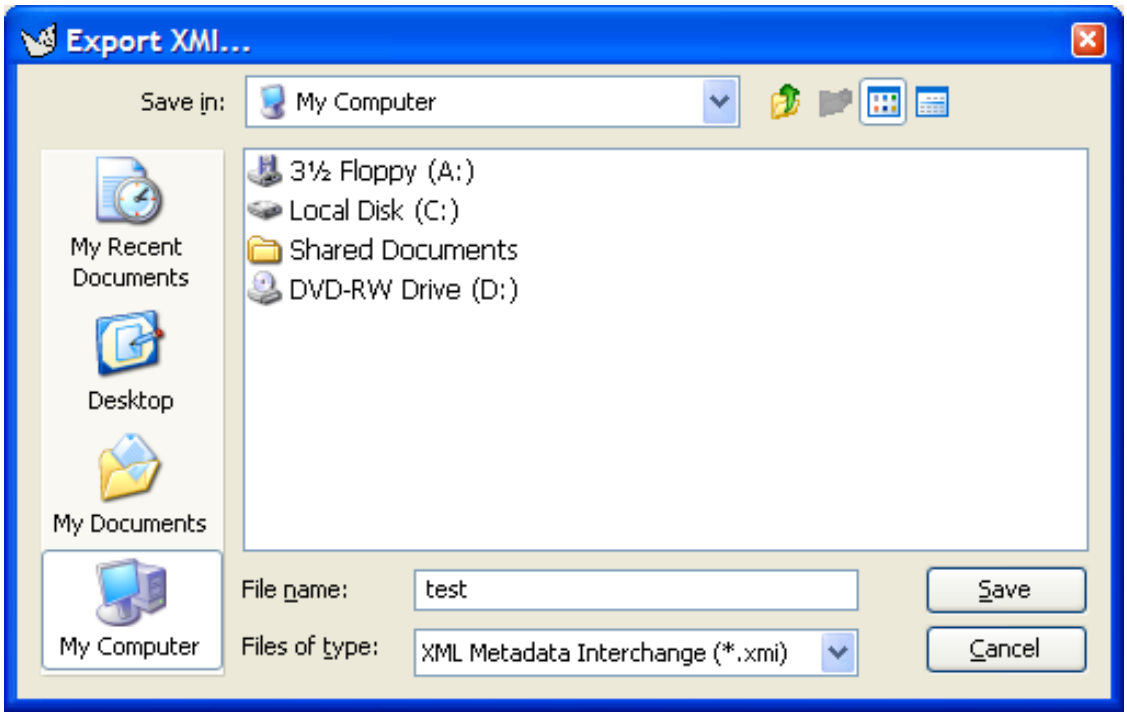
10.3.7. Export XMI...

This menu-item allows to save the complete structure of the UML 1.4 model as a XMI file, according the XMI V1.2 standard. Beware the fact that this file will only contain the model, not any diagram layout. Hence, if the XMI file is reloaded with the `File - Open Project...` menu, then the diagrams are lost.

When the menu is activated, the standard filechooser appears, see Figure 10.6, “The dialog for `Export`

XMI...".

Figure 10.6. The dialog for `Export XMI...`

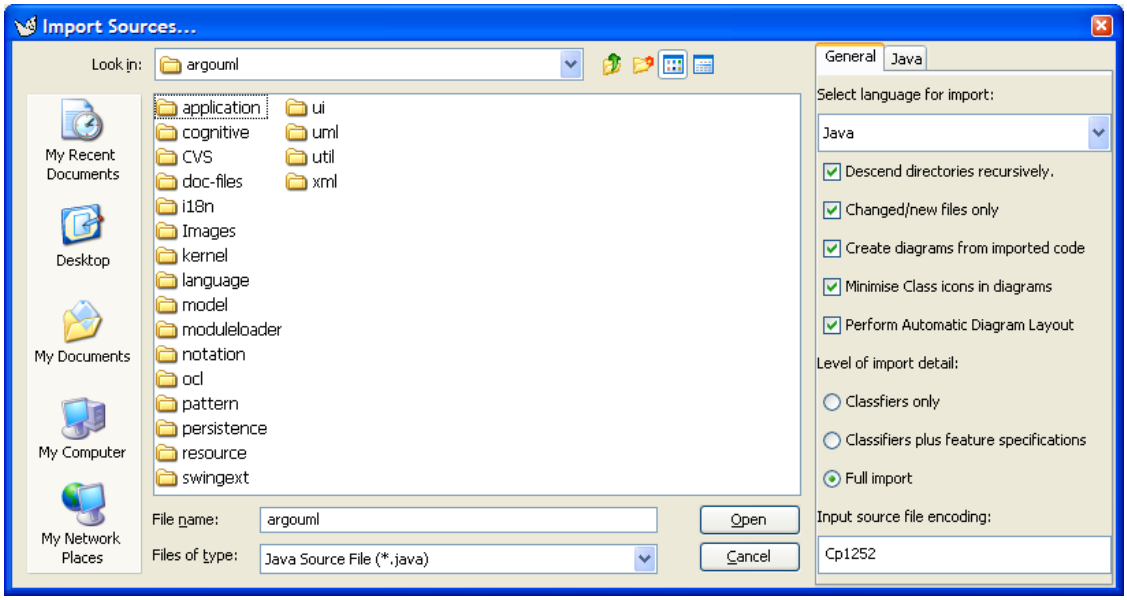


10.3.8. Import Sources...

A very powerful feature of ArgoUML is that it can "Reverse Engineer" Java code to yield a class diagram. This sub-menu entry specifies Java code to be imported for reverse engineering.

The dialog box is similar to that for `Open Project` (see Figure 10.2, "The file selection dialog for `Open Project...`"), but with two extra tabs placed alongside the directory listing, as shown in Figure 10.7, "The file selection dialog for `Import Sources...`").

Figure 10.7. The file selection dialog for `Import Sources...`



Those fields that are the same as `Open Project` behave in the same way (see Section 10.3.2, “ Open Project...”).

Next to the "All Files" file filter, there is the default filter "Java Source File (*.java)".

The first of the two tabs is labeled `General` and is selected by button 1 click on its tab. It provides a combo box for the language selection (in V0.18 of ArgoUML only Java can be chosen), and the following selections:

- `Descend directories recursively`. If enabled (the default), reverse engineering will track through sub-directories for any further Java files. If not it will restrict to the selected directory.
- `Changed/new files only`. If enabled (the default), only changed and new files are imported. If not all classes will be replaced.
- `Create diagrams from imported code`. If you unselect this, then no diagrams are created, i.e. all data will only be visible in the explorer.
- `Minimise Class icons in diagrams`. If enabled, then the attributes and operations compartiments will not be shown in the classes on the generated class diagrams. Note: This item is checked by default, and is overseen by many users, which are then surprised by the result.
- `Perform Automatic Diagram Layout`. If selected, then ArgoUML will do its best to layout the generated diagrams automatically. If not, then all items will be placed at the top left corner of the diagram.
- `Level of import detail: Classifiers only / Classifiers plus feature specifications / Full import`. The latter is the default.
- `Import source file encoding:`. The value `Cp1252` is often the default. This string represents the `coded character set identifier (CCSID)`.

The second of the two tabs is labeled `Java` and is selected by button 1 click on its tab. It provides two pairs of radio boxes.

- The first radio box allows selection between modeling attributes of Java classes as UML attributes (the default) or as UML associations to the class specified.
- The second radio box allows selection between modeling arrays as new datatypes in their own right (the default) or as their base datatype with multiplicity.

10.3.9. Page Setup...

This brings up the standard dialog box provided by the operating system to adjust printer paper size, orientation, and other options.

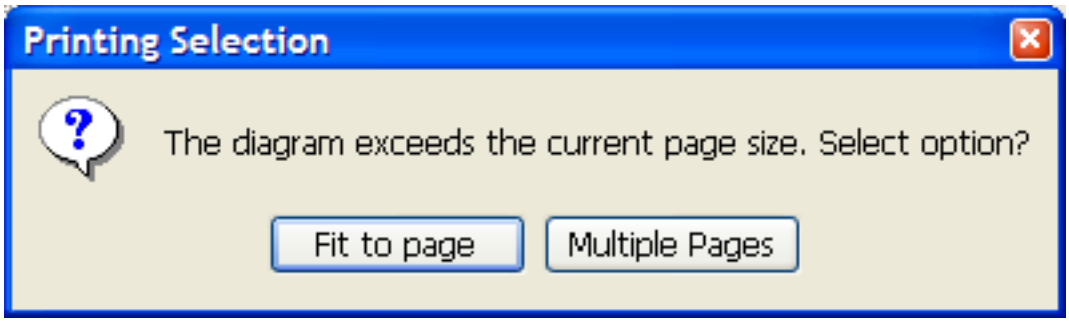
10.3.10. Print...

Shortcut Ctrl-P.

This brings up the standard dialog box provided by the operating system allowing the current diagram to be printed out.

In some cases, when the printing is started, the dialog box of Figure 10.8, “The diagram exceeds page size dialog.” appears. Selecting the "Fit to page" button does print the whole diagram fitted on one page by scaling it down. Which might cause all text to be too small to read in case of big diagrams, but it is a quick and easy way to get an usable printout. Selecting the "Multiple pages" option does print unscaled, by dividing the diagram in pieces, on as many pages as needed. Pressing the close button of the dialog does the former option.

Figure 10.8. The diagram exceeds page size dialog.



Warning

If the current diagram contains no selected model elements, then the whole diagram is printed. However, if one or more model elements are selected, then only the area they cover is printed! If scaling is selected (by the "Fit to page" choice in the dialog box descibed above), then the scaling is done on basis of the selected model elements only. If scaling is not chosen (or in case it is not needed), then all pages containing a selected model element are printed.

10.3.11. Export Graphics...

This menu entry brings up a dialog box allowing the currently selected diagram (in the editing pane) to be saved in one of a number of graphic formats.

The dialog box is identical to that for `Open Project` (see Figure 10.2, “The file selection dialog for `Open Project...`”), except for the `Files of type:`. The chosen filetype specifies the graphics format used for saving. The filename is automatically extended with the corresponding ending (if not entered already). A default filename is generated based on the diagram name.

The available graphics types are:

- GIF image (*.gif)
- Encapsulated Postscript file (*.eps)
- PNG image (*.png)
- Postscript file (*.ps)
- Scalable Vector Graphics file (*.svg)

The graphics format that is selected by default is set in the dialog under the menu entry Edit - Settings...

10.3.12. Export All Graphics...

This menu entry brings up a dialog box to select a directory. In this directory, for all diagrams in the current project, a graphics file is generated.

The names of the files are deducted from the diagram names. The graphics format that is produced is set in the dialog under the Edit menu (see Section 10.4.5, “ Settings...”).

10.3.13. Notation

This sub-menu presents a radio button selection for notation, i.e. the language in which all textual adornments are shown on the diagrams.

This feature defines the project's notation language.

There are 2 ways to set the notation language:

- In the Edit menu, see Section 10.4.5.5, “Notation Tab” in the notation tab of the settings dialog, which defines the default notation language for new projects. This choice is stored in the `argouml.properties` file.
- In the File menu, item Notation. This determines how all textual adornments of figures on all diagrams of the current project are shown. This choice is stored in the project file.

The following 2 notations are build in ArgoUML:

- `UML 1.4`. Uses UML notation as the default notation for every modelelement on any diagram.
- `Java`. Uses Java notation as the default notation for every modelelement on any diagram.

The following choices are only available if the corresponding plugin languages are installed.

- Cpp.
- CSharp.
- PHP.

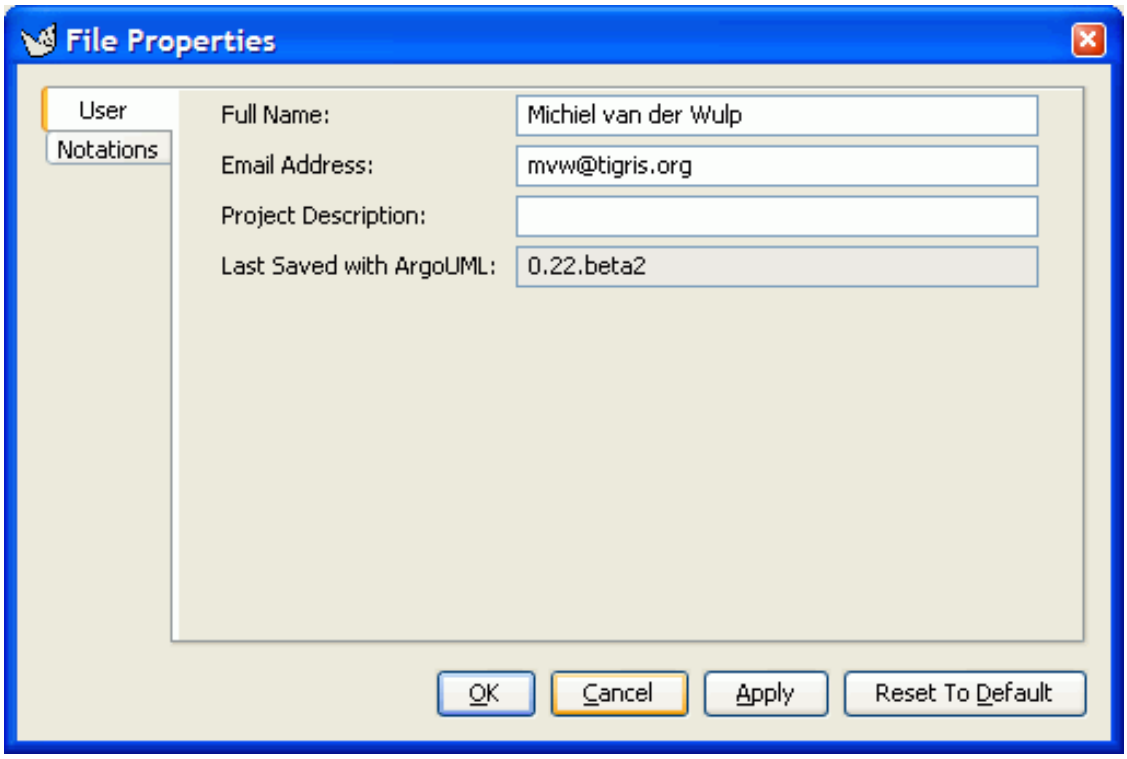
Besides UML, only Java is partly implemented in V0.22 of ArgoUML.

10.3.14. Properties

This menu entry brings up a dialog box, which allows the user to set various options of the currently loaded project.

All settings in this dialog are stored in the project-file together with the model.

Figure 10.9. The dialog for Properties - Notation: The User tab.

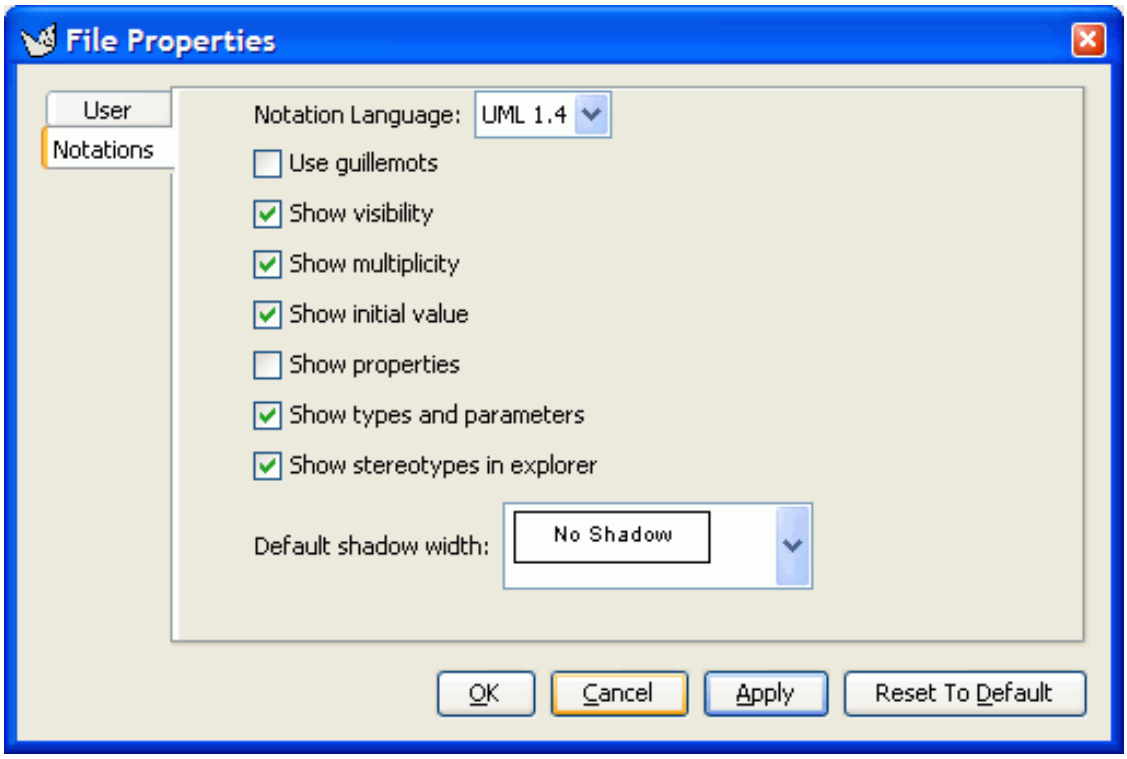


In the User tab, you are able to set the following fields:

- The first field contains the name of the author or responsible for the current project. By default the name and email of the creator are filled in, so probably you will never need to edit this, but it is possible.
- The Project Description field may contain any text that you need to describe the project. By default it is empty.
- The "Last saved with ArgoUML" field indicates the version of ArgoUML that was used to save this project (the last time it was saved). This may be usefull if multiple designers have different versions

of ArgoUML, which may not be backwards compatible all the time.

Figure 10.10. The dialog for Properties - Notation: The Notations tab.



In the Notations tab, you are able to set the following fields:

- The first field is a combobox that allows selection of the project's Notation language. By default, it lists UML and Java, but other languages may be added by plugins. See the chapter on Notation for more explanation: Section 12.11, “Notation”.
- `Use guillemots` (« ») for stereotypes (clear by default). By default ArgoUML uses pairs of *less than* and *greater than* (<< >>) characters for stereotypes. If this box is checked stereotypes on diagrams are shown between true guillemots (« »).

  This feature is presumably added to ArgoUML because guillemots are poorly supported by various fonts, and if they are present, then they are quite small and poorly visible.
- `Show visibility` (clear by default). If this is selected, then ArgoUML will show the visibility indicators in front of e.g. attributes in the diagram. In UML the notation is "+" for public, "-" for private, "#" for protected, and "~" for package. E.g. for an attribute, it may show: `+newAttr : int`.
- `Show multiplicity` (clear by default). If this is selected, then ArgoUML will show the multiplicity of e.g. attributes in the diagram. In UML notation, the multiplicity is shown between [], such as: `+newAttr [0..*] : int`. This setting has no impact on showing multiplicity near associationends.
- `Show initial value` (clear by default). If this is selected, then ArgoUML will show the initial

value of e.g. attributes in the diagram. In UML notation, the initial value is shown e.g. like this: `+newAttr : int = 1`.

- `Show properties` (clear by default). If this is selected, then ArgoUML will show various properties between braces {}. E.g. for an attribute, it may show: `+newAttr : int { frozen }`.
- `Show types and parameters` (set by default). When this checkbox is unmarked, attributes in classes are shown without type indication, and operations are shown without parameters. This feature may be usefull during the analysis phase of your project. If all checkmarks in the Notation Tab are unchecked, then e.g. for an attribute, ArgoUML may show: `newAttr`. And for an operation: `newOperation()`.
- `Show stereotypes in explorer` (clear by default). If this is selected, then ArgoUML will show stereotypes next to the icons of the modelelements in the Explorer, i.e. the tree structure at the left hand side.
- `Default shadow width` (set to 1 by default). ArgoUML is able to draw all elements on a diagram with a shadow, for esthetical reasons. Use this setting to adjust the size of the shadow, used when the modelelement is created. The details tab "Presentation" allows to set the shadow per modelelement, after they are created, but ArgoUML V0.22 does not retain this latter change after save and load.

10.3.15. Most Recent Used Files

ArgoUML remembers a few of the most recently saved files, and lists them here, to enable loading then in the most simple way.

The maximum number of files that is listed here, can be adjusted in the `Edit` -> `Settings...` menu. The list of files is stored in the `argo.user.properties` file in the user's home directory.

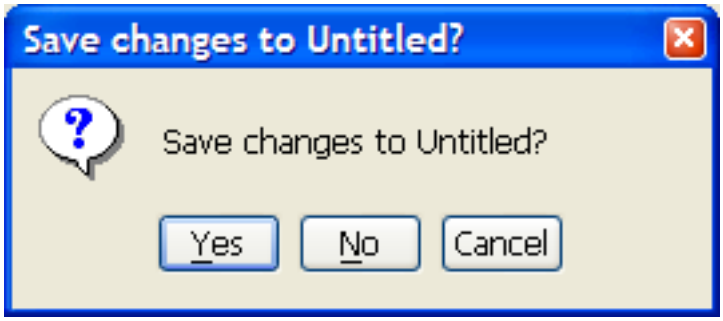
10.3.16. Exit

Shortcut Alt-F4.

This closes down ArgoUML. A warning message will pop-up if you have a project open with unsaved changes asking if you wish to save it. See Figure 10.11, “The save changes dialog.”. The options are:

- `Yes` (save the project and exit ArgoUML);
- `No` (do not save the project, but still exit ArgoUML); and
- `Cancel` (do not save the project and do not exit ArgoUML).
- The dialog box can also be closed by clicking in the close button in the window border. The effect is the same as selecting "Cancel".

Figure 10.11. The save changes dialog.



10.4. The Edit Menu

This menu provides support for selecting model elements on the editing pane; removal of model elements from diagrams and the model; and control of user settings.

10.4.1. Select

This sub-menu provides for selection of items on the editing menu. It has the following entries.

- `Select All` (shortcut Ctrl-A). Selects all model elements on the current pane or in the current field. The exact behaviour depends on the `current pane` (i.e. the last one you clicked in): explorer pane, editing pane, to-do pane, details pane. One rule applies in all cases though: the selection on the diagram (editing pane) and in the explorer are always synchronised.

  If the editing pane is the `current pane`: First everything in the explorer and on the current diagram is deselected, and then everything that is on the current diagram is selected (and if the same items apear in the explorer, then they are also there indicated as selected, because they are always synchronised).

  If the explorer pane is the `current pane`: All visible items in the explorer pane are selected, and non-visible items are deselected.

  If the to-do pane is the `current pane`: All visible items in the to-do pane are selected, and non-visible items are deselected. In fact, this works the same as for the explorer pane, because both are tree structures.

  If the details pane is the `current pane`: The function only works when the cursor is in certain fields, where selecting is possible, e.g. a Name field. In such a case, the Select All function extends the current selection to the whole field contents.

- `Navigate Back`. ArgoUML keeps a record of the model elements that you have been selecting while navigating the model. This button moves you back to the previous one selected. If there are no more previous model elements, the button is grayed out.

- `Navigate Forward`. ArgoUML keeps a record of the model elements that you have been selecting while navigating the model. This button moves you forward to the next one selected (after you have used the Navigate Back button to move back). If there are no more next model elements, the button is grayed out.

- `Invert Selection`. This inverts the current selection on the `current pane`. More exact: everything that was selected is de-selected and everything that was not selected within the current pane is selected.

10.4.2. Remove From Diagram

Shortcut Delete.

This removes the currently selected item(s) from the diagram, but not from the model.

The modelelement can be re-added to the diagram by button 2 click on the modelelement in the explorer, or by dragging it onto the diagram.

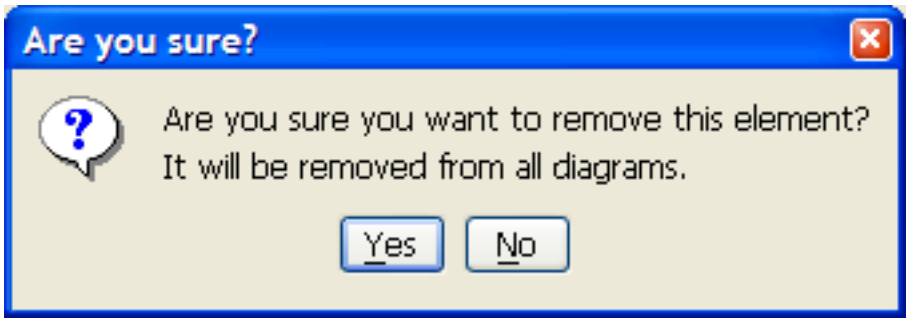
10.4.3. Delete From Model

Shortcut Ctrl-Delete.

This function deletes the selected item(s) from the model completely.

If the item to be deleted is also present on another diagram than the current one, the dialog box from figure x appears.

Figure 10.12. The dialog for confirmation of `Remove from Model`.



10.4.4. Configure Perspectives...

This menu-item invokes the same dialog as the button at the top of the explorer. See Section 11.5, “Configuring Perspectives”. for a complete description.

10.4.5. Settings...

This menu entry brings up a dialog box, which allows the user to set various options that control the behavior of ArgoUML (see Figure 10.13, “The dialog for `Settings - Preferences`.”).

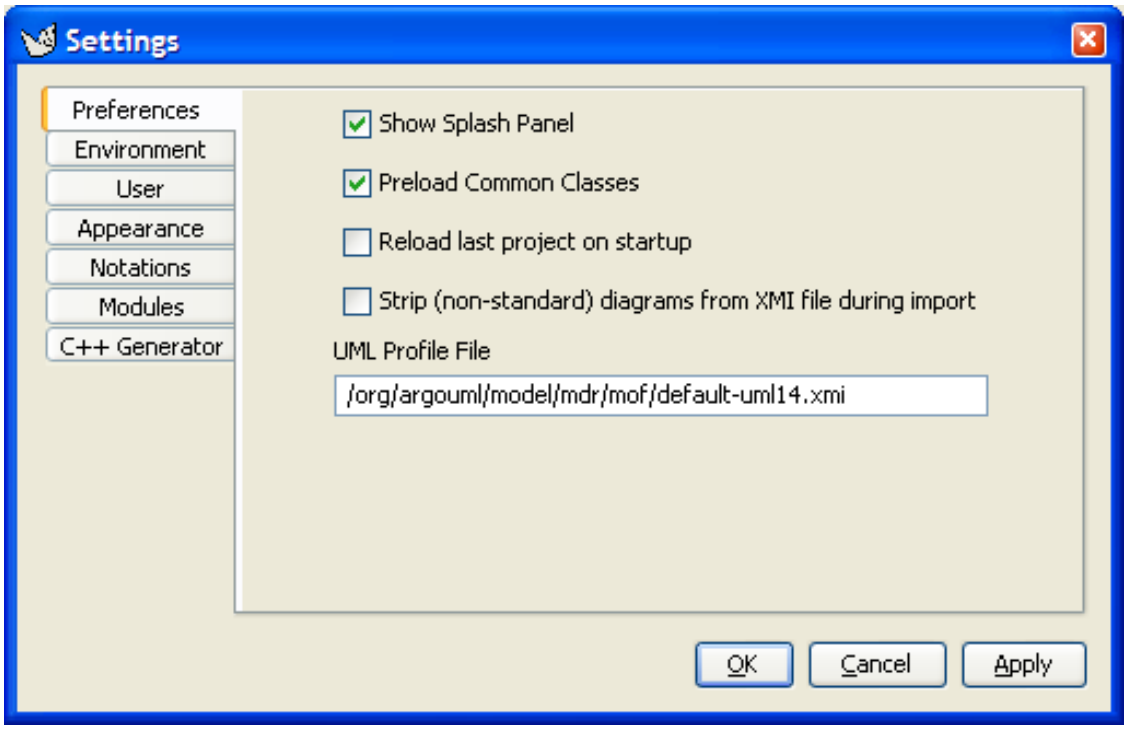
These settings are saved persistently for use by subsequent ArgoUML sessions.

ArgoUML has various user specific configurations that can be set in this dialog box, or directly on the various panes. Also the main window size and location is such a setting. Activating this menu entry causes the information to be saved in the file `argo.user.properties`. The location of this file is in the "users home directory", which is defined as `${user.home}` , and can be determined as described in Section 10.4.5.2, “Environment Tab” .

Tip

This is a text file, which you can edit to configure ArgoUML.

Figure 10.13. The dialog for `Settings - Preferences`.



The options that can be set up on the various tabs are described in the following sections. For each tab there are three buttons at the bottom of the dialog box.

- `OK`. Activating this button (button 1 click) applies the chosen settings and exits the dialog.
- `Cancel`. Selecting this button (button 1 click) exits the dialog without applying any settings changed since the last `Apply` (or since the dialog started if `Apply` has not been used).
- `Apply`. Selecting this button (button 1 click) applies the chosen settings and remains in the dialog.

Closing the dialog (with the close button in the top corner in the border of the window) causes the same effect as `Cancel`.

10.4.5.1. Preferences Tab

Selecting the `Preferences` tab (button 1 click on the tab) gives the following options as check boxes.

- `Show Splash Panel` (set by default). If enabled ArgoUML will show a small panel with a picture while starting up.

Tip

The splash panel can be seen by using the Help menu (see Section 10.11.2, “About ArgoUML”).

- `Preload Common Classes` (set by default). If enabled ArgoUML creates class objects of a number of classes internally during start up so that instantiation is quicker when they are needed.
- `Reload last saved project on startup` (clear by default). Check this item if you always work on the same project, and wish to load it automatically when you start up ArgoUML.
- `Strip (non-standard) diagrams from XMI file during import` (clear by default). Checking this item will tell ArgoUML to ignore the "Diagram" elements when importing XMI files.

  You only need to use this setting, if ArgoUML gives an error while importing your XMI file saying that it encountered unrecognized elements named "Diagram." Some versions of Poseidon are known to create this type of file by default although there's usually an export option to force them to create standard XMI files.
- `UML Profile file` (/org/argouml/model/mdr/mof/default-uml14.xmi by default).

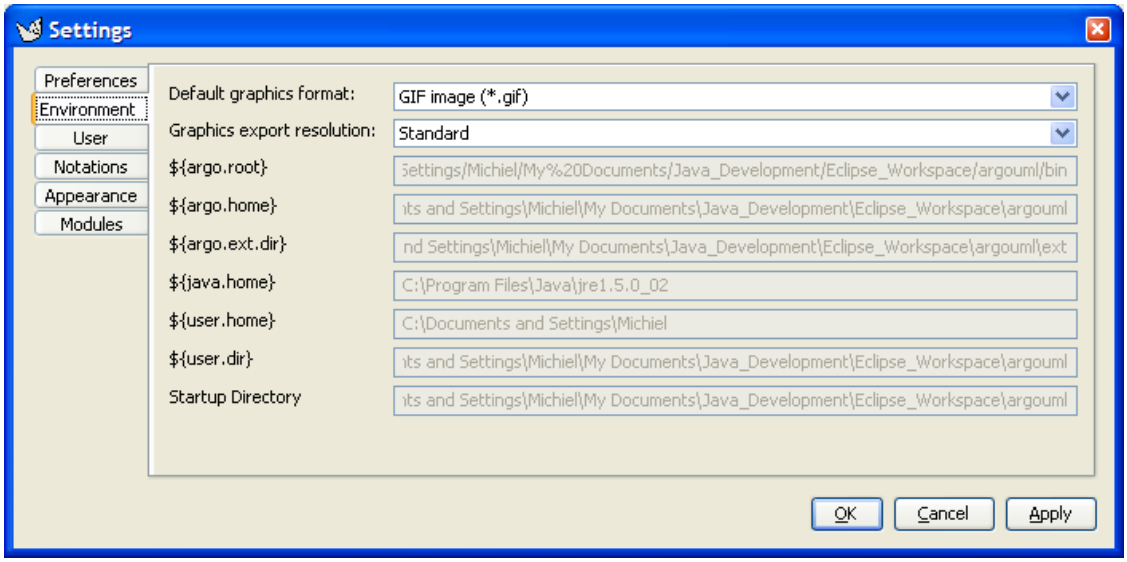
  This is a read-only field which shows the current profile being used by ArgoUML. If you specified an alternate profile at startup time or a plugin-module installed a different profile, it will display here.

  In the future this will be a settable field that allows you to select different profiles to match different modeling environments (Java, C++, AndroMDA, etc).

10.4.5.2. Environment Tab

Selecting the `Environment` tab (button 1 click on the tab) lists several environmental items. Note that none of the paths can be altered — these are just a matter of record.

Figure 10.14. The dialog for `Settings - Environment`.



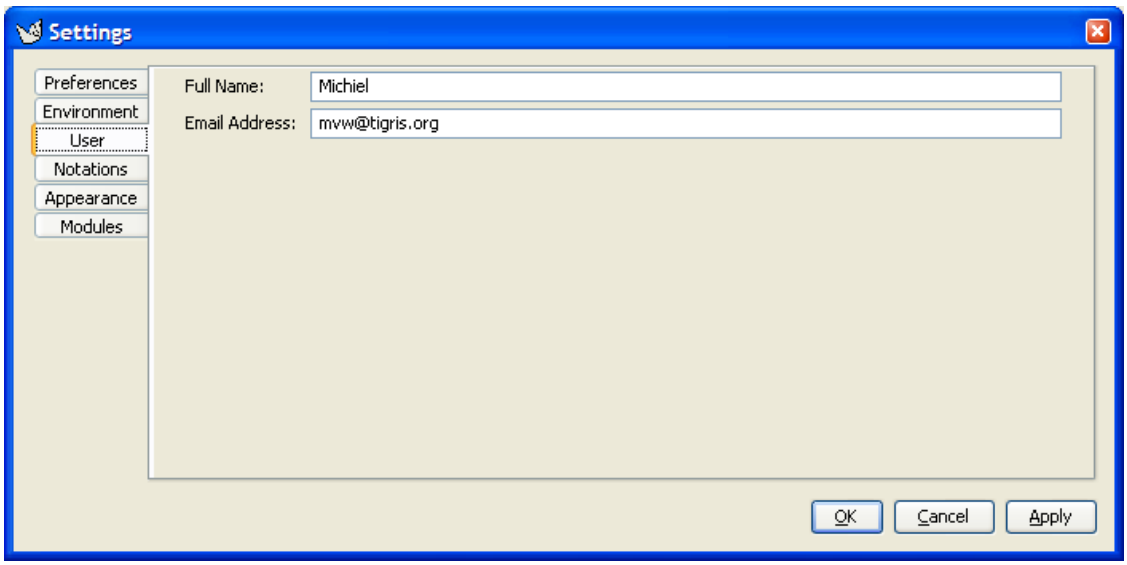
- `Default Graphics Format`. Here you can select the same graphics formats as in the menu Section 10.3.11, “Export Graphics...”. The chosen format is selected by default in the Export Graphics and Export All Graphics menu-items.

- `Graphics Export Resolution`. This allows you to artificially increase the resolution of produced graphics. The advised setting is "Standard". To be able to use "High" or "Extra High", you usually need to start the Java virtual machine with extra memory.
- `${argo.root}`. The full path to the ArgoUML program, i.e. the `argouml.jar` file.
- `${argo.home}`. The ArgoUML home directory which contains the "jar" files needed by ArgoUML.
- `${argo.ext.dir}`. The directory holding ArgoUML extensions—by default the `ext` subdirectory of the ArgoUML build directory.
- `${java.home}`. The home directory of the Java Runtime Environment (JRE).
- `${user.home}`. The user's home directory. Used for storing the `argo.user.properties` file.
- `${user.dir}`. The directory from which ArgoUML was started.
- `Startup Directory`. The directory in which ArgoUML starts file searches etc.

10.4.5.3. User Tab

This tab allows the user to record additional information of use to the system. There are two text boxes provided.

Figure 10.15. The dialog for `Settings - User`.



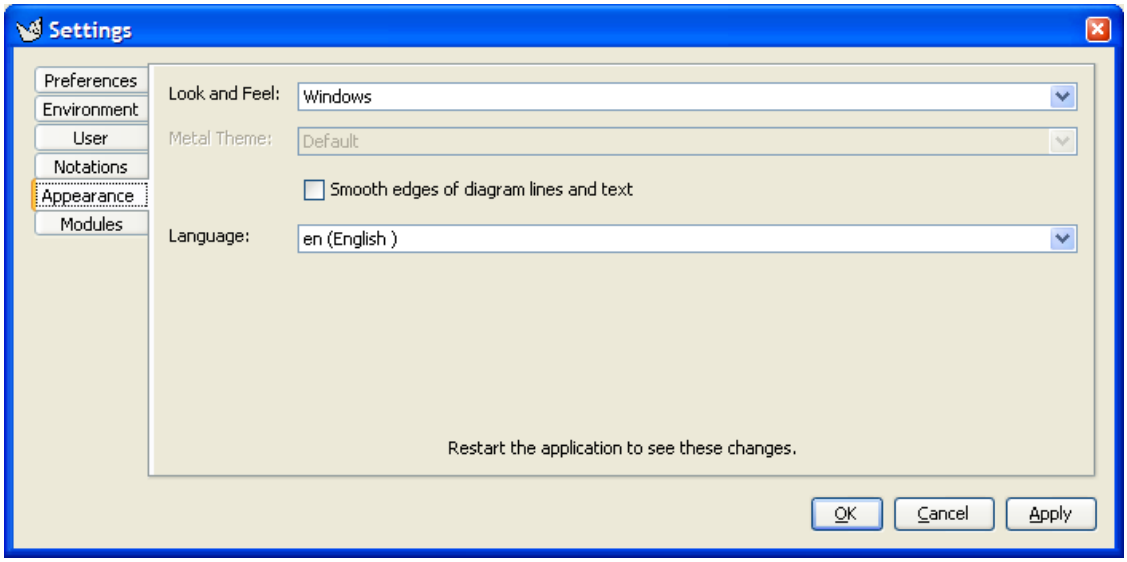
- `Full Name`. Allows the user to record her full name.
- `Email Address`. Allows the user to record his Email address.

This information is used when requesting to-do help by Email.

10.4.5.4. Appearance Tab

This tab allows the user to specify the LAF (Look And Feel) and theme, i.e. what the complete ArgoUML UI looks like. It comprises the following settings.

Figure 10.16. The dialog for `Settings - Appearance`.



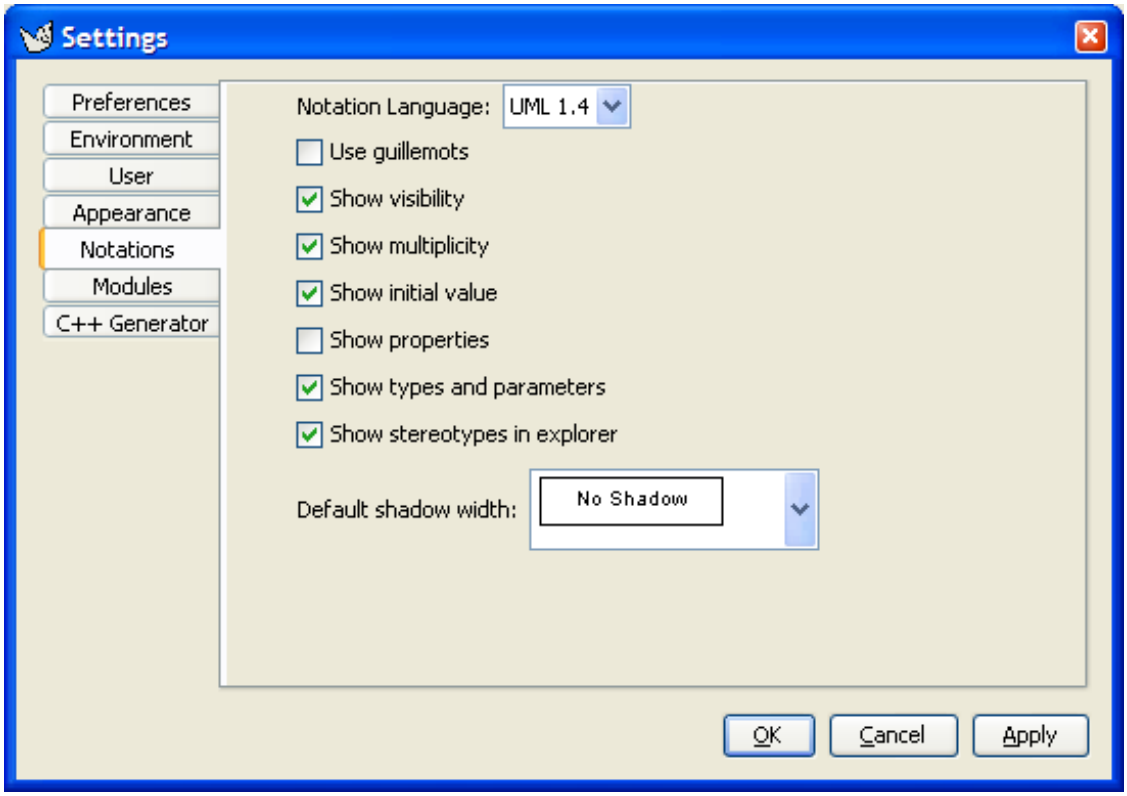
- `Look and Feel`. The choice made here influences the complete User Interface. It only becomes effective when ArgoUML is exited and restarted.
- `Metal Theme`. This item is downlighted if the Metal LAF is not chosen. The choice made here influences the complete User Interface. It only becomes effective when ArgoUML is exited and restarted.
- `Smooth edges of diagram lines and text`. This feature is known as “anti-aliasing” on certain platforms. It causes diagonal lines to look much less jagged, by making use of several shades of gray. This feature only works if the operating system supports it.

10.4.5.5. Notation Tab

This tab allows the user to specify certain notation settings, i.e. how things are shown on diagrams. It comprises the following check boxes.

All settings here, only define the defaults used for new projects. If you want to change the way the diagrams in your current project look, then see the File - Properties menu.

Figure 10.17. The dialog for `Settings - Notations`.



- `Notation Language` (UML 1.4 by default). This feature allows changing the default notation (i.e. language: UML, Java,...) used on the diagrams for new projects. Suppose that a designer indicates that the default notation of a project is Java. When he saves the project, the choice for Java is stored inside the project file. If someone else is viewing the diagram, he will see the Java notation, too. This person can select the UML notation in the File - Notation menu, and see all diagrams in UML language. See Section 10.3.13, “Notation”).

- `Use guillemots` (« ») for stereotypes (clear by default). By default ArgoUML uses pairs of *less than* and *greater than* (<< >>) characters for stereotypes. If this box is checked stereotypes on diagrams are shown between true guillemots (« »).

  This feature is presumably added to ArgoUML because guillemots are poorly supported by various fonts, and if they are present, then they are quite small and poorly visible.

  Independent of the way they are shown, when entering stereotypes, you can always type real guillemots (if your keyboard supports it) or their << >> equivalents.

- `Show visibility` (clear by default). If this is selected, then ArgoUML will show the visibility indicators in front of e.g. attributes in the diagram. In UML the notation is "+" for public, "-" for private, "#" for protected, and "~" for package. E.g. for an attribute, it may show: `+newAttr : int`.

- `Show multiplicity` (clear by default). If this is selected, then ArgoUML will show the multiplicity of e.g. attributes in the diagram. In UML notation, the multiplicity is shown between [], such as: `+newAttr [0..*] : int`. This setting has no impact on showing multiplicity near associationends.

- `Show initial value` (clear by default). If this is selected, then ArgoUML will show the initial value of e.g. attributes in the diagram. In UML notation, the initial value is shown e.g. like this:

`+newAttr : int = 1`.

- `Show properties` (clear by default). If this is selected, then ArgoUML will show various properties between braces {}. E.g. for an attribute, it may show: `+newAttr : int { frozen }`.
- `Show types and parameters` (set by default). When this checkbox is unmarked, attributes in classes are shown without type indication, and operations are shown without parameters. This feature may be usefull during the analysis phase of your project. If all checkmarks in the Notation Tab are unchecked, then e.g. for an attribute, ArgoUML may show: `newAttr`. And for an operation: `newOperation()`.
- `Show stereotypes in explorer` (clear by default). If this is selected, then ArgoUML will show stereotypes next to the icons of the modelelements in the Explorer, i.e. the tree structure at the left hand side.
- `Default shadow width` (set to 1 by default). ArgoUML is able to draw all elements on a diagram with a shadow. Use this setting to adjust the size of the shadow, used when the modelelement is created. The details tab "Presentation" allows to set the shadow per modelelement, after they are created.

10.4.5.6. Modules Tab

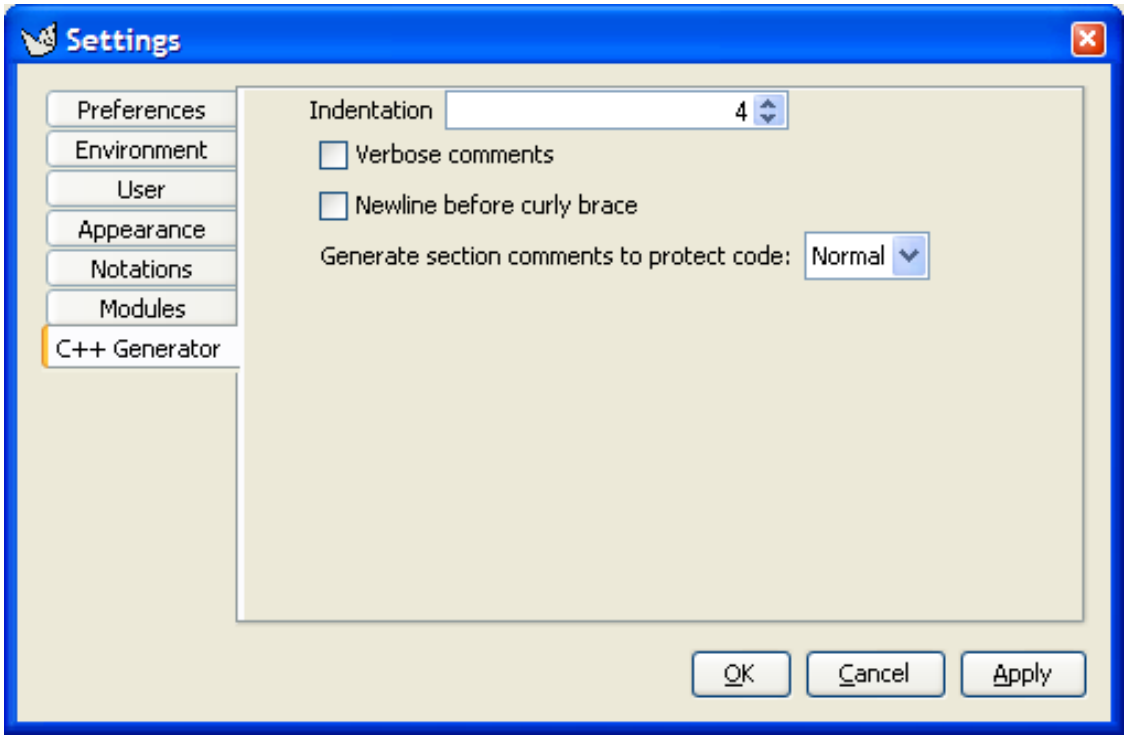
This tab shows a list of modules that are installed, which may be enabled or disabled. Since this is a new concept for ArgoUML, it currently contains a list of modules that can not be removed, and a button to test the concept. Pressing this button adds a useless menu-item on the Tools menu, nothing else.

Notice also that this is a "new" modules concept so the old Pluggable modules do not work this way, and are not listed.

10.4.5.7. Extra Tabs added by Plugins

A plug-in module has the possibility to add extra tabs. One example is C++; it adds the following tab.

Figure 10.18. The dialog for `Settings - C++`.



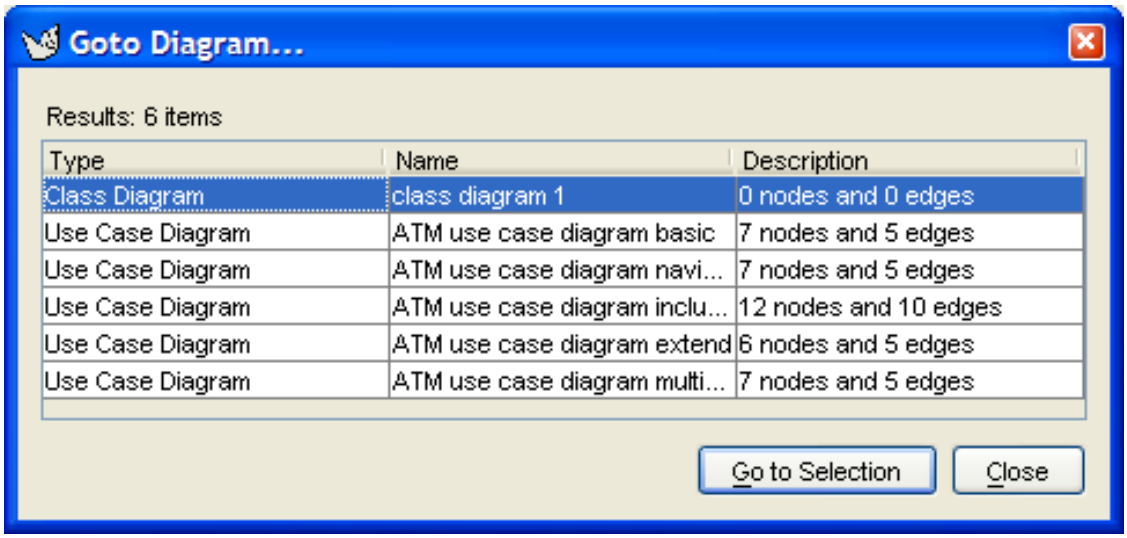
10.5. The View Menu

This menu is used for actions that affect how the various panes are viewed.

10.5.1. Goto Diagram...

This menu entry brings up a dialog box, describing all the diagrams in the current project under ArgoUML.

Figure 10.19. The dialog for `Goto Diagram...`



The dialog box contains a table with three columns and one row for each diagram in the current project. A scroll bar gives access if the table is too long for the box. Double button 1 click on any row will select that diagram in the editing pane. The three columns are as follows.

- `Type`. Lists the type of diagram.
- `Name`. Lists the name given to the diagram.
- `Description`. Shows how many nodes and edges there are on the diagrams. A node is a “2-D” model element and an edge is a connector model element.

This dialog box is not modal, which allows it to remain open while editing the model for easy navigation.

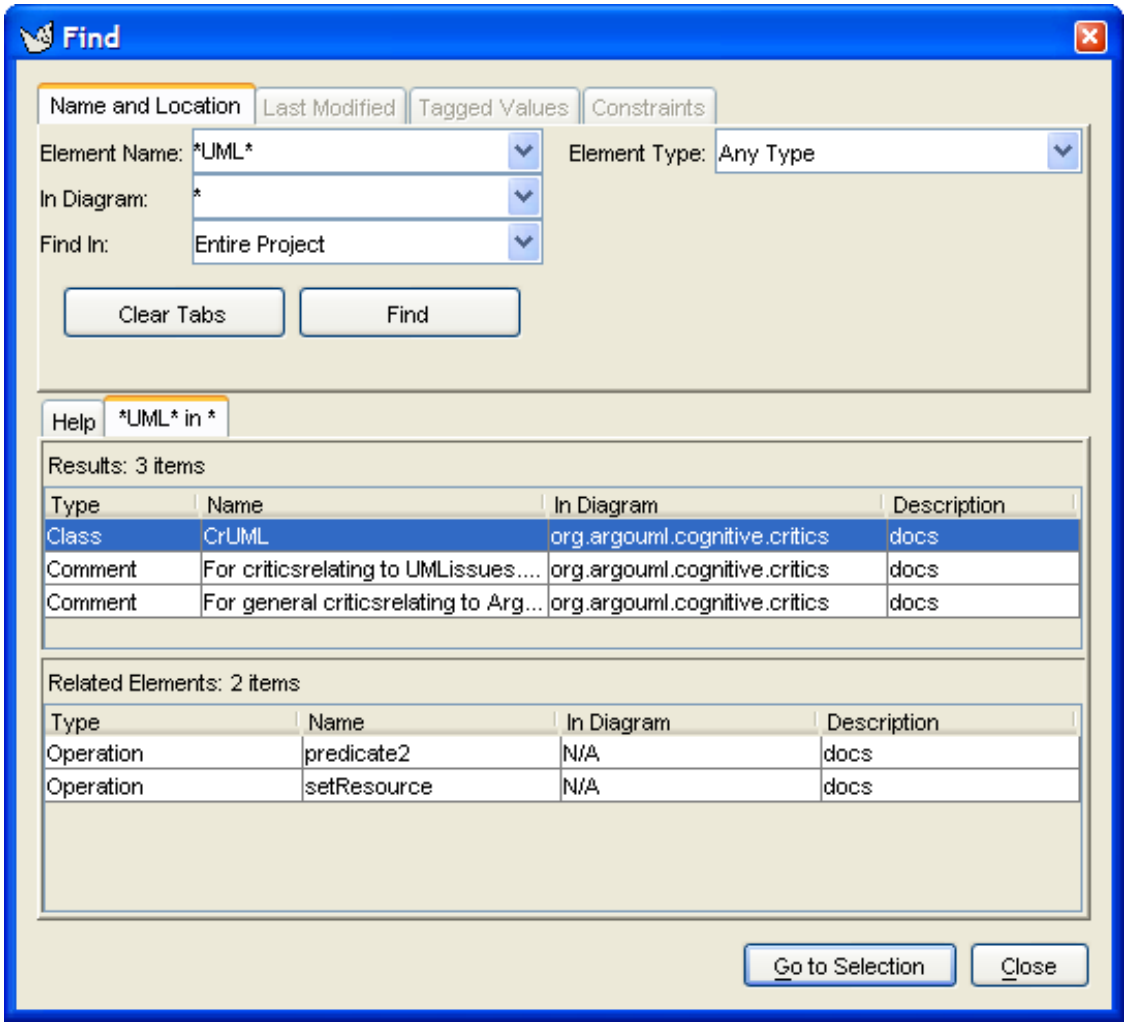
Warning

The V0.22 implementation of ArgoUML does not inmediately update the dialog box with changes made to diagrams: change of name, addition of diagrams, deletion of diagrams.

10.5.2. Find...

This menu entry brings up a non-modal dialog box for the ArgoUML search engine.

Figure 10.20. The dialog for `Find...`



At the top, the dialog box has four tabs labeled `Name and Location`, `Last Modified`, `Tagged Values` and `Constraints`. Of these all but the first are grayed out in the V0.22 version of ArgoUML (because they are not imlemented yet), so the first tab is always selected.

The `Name and Location` specifies the search to be made. It contains the following:

- A text box labeled `Element Name:` specifies the name of the model element to search for. Wild cards (`*`, `?`) may be used here. A drop down gives access to find expressions previously used.
- A text box labeled `In Diagram:` specifies which diagrams are to be searched. Again wild cards may be used. Both these two text boxes have a default entry of `*`, i.e. match anything.
- To the right of these two text boxes, a selector labeled `Element Type:` allows you to specify the UML metaclass for which you are searching.
- A selector labeled `Find in:` allows the search to be made over the entire project (the default) or as a sub-search over the results of a previous search. When opened, a list of all the search result tabs appears.
- Beneath these boxes is the button `Clear Tabs`. This clears the display of tabs with the results from previous searches (see below). This button is downlighted if there are no tabs but the `Help` tab.

- And finally, there is the button `Find`. This causes the search specified in the text boxes and selectors above to be executed. The results are displayed in a tab taking up the lower two thirds of the page.

The lower two thirds of the dialog comprises an initial tab (labeled `Help`) giving summary help, and further tabs displaying the results of searches. These search tabs are labeled with a summary of the search *element* `in` *diagram* and are divided horizontally in two halves.

Button 1 double clicking on these tabs removes the tab, and spawns a new window that contains the tab contents, i.e. the search results. This window can be moved and sized at will. This does not work for the help tab.

The top half is labeled `Search Results:` followed by a count of the number of items found. It comprises a table with one row for each model element and four columns. The width of the columns can be adjusted.

- `Type`. Lists the type of model element.
- `Name`. Lists the name given to the model element.
- `In Diagram`. Where the model element is visible on a diagram, this lists the name of the diagram, otherwise it shows `N/A`.
- `Description`. Contains a description of the model element. In ArgoUML V0.18 this seems to be restricted to the single entry `docs`.

Button 1 click on any row will give more information on that model element by showing related model elements in the bottom half (see below). Double click on any row describing a model element on a diagram and that item and diagram will be selected.

The bottom half of the tab is a table labeled `Related Elements:` and is a table with the same columns as the top half. When a model element has been selected in the top half, this table shows the details of any related elements.

Tip

Enlarging the dialog vertically shows that the "Related Items" part changes in size, but not the Search results part. However, between them is a divider line and when hovering over this line, the mouse pointer changes into a sizing icon, and the border between these 2 areas can be moved up or down to redistribute the space in the window.

Warning

This dialog box is not modal, which allows it to remain open while editing the model for easy navigation. But the V0.22 implementation of ArgoUML does not inmediately update the dialog box with changes made to the found model elements: change of model element name, change of diagram name. Deletion of a diagram does not stop the possibility to navigate to it.

10.5.3. Zoom

This entry brings up a sub-entry, which allows scaling the view of all diagrams to a factor of its normal size. This setting is not saved persistently.

The sub-menu items that can be selected are:

- `Zoom Out`. Shortcut (Ctrl-Minus). Gives more overview over the drawing.
- `Zoom Reset`. Returns to the default zoom ratio (i.e. 100%).
- `Zoom In`. Shortcut (Ctrl-Plus). Makes the items on the drawings bigger.

10.5.4. Adjust Grid

This allows selection of the grid representation on the screen between the following:

- `Lines 16`: full grid at 16 pixel spacing.
- `Lines 8`: full grid at 8 pixel spacing.
- `Dots 16`: dots at 16 pixel spacing (the default).
- `Dots 32`: dots at 32 pixel spacing.
- `None`: no grid of any form.

10.5.5. Adjust Snap

This allows selection of the spacing of grid snapping between the following:

- `Snap 4`: snap at 4 pixel spacing.
- `Snap 8`: snap at 8 pixel spacing (the default).
- `Snap 16`: snap at 16 pixel spacing.
- `Snap 32`: snap at 32 pixel spacing.

Note

There is no option to turn off snap to grid altogether

Note

If you wish to align existing elements to changed snap boundaries, you can use the `Arrange > Align To Grid Snap` menu (see Section 10.7.1, “Align”).

10.5.6. Page Breaks

This toggles whether page breaks are shown on the diagram (as white dotted lines).

Warning

This menu item does not work in ArgoUML V0.24.

10.5.7. XML Dump

This activates a window that shows the complete contents of the current project in XML format.

Although very useful for debugging ArgoUML, this menu function is hardly interesting to the common user.

10.6. The Create Menu

This menu provides for creating the various types of UML diagrams supported by ArgoUML.

10.6.1.  New Use Case Diagram

This menu entry creates a blank use case diagram, and selects that diagram in the editing pane. If a package is currently selected, then the use case diagram will be created within that package. This means that it will be shown within the package on the explorer hierarchy (under Package-centric view) and model elements created on the diagram will be created within the namespace of the package. This does not only apply to a package, but also to a class, interface, use case, etc.

Tip

This does not prevent model elements from other namespaces/packages appearing on the diagram. They can be added from the explorer using `Add to Diagram` from the button 2 pop-up menu.

10.6.2.  New Class Diagram

This menu entry creates a blank class diagram, and selects that diagram in the editing pane. If a package is currently selected, the class diagram will be created within that package. This means that it will be shown within the package on the explorer hierarchy (under Package-centric view) and model elements created on the diagram will be created within the namespace of the package. This does not only apply to a package, but also to a class, interface, use case, etc.

Tip

This does not prevent model elements from other namespaces/packages appearing on the diagram. They can be added from the explorer using `Add to Diagram` from the button 2 pop-up menu.

10.6.3.  New Sequence Diagram

This menu entry creates a blank sequence diagram, and selects that diagram in the editing pane. It also creates a `Collaboration` UML element, which is a container for the elements shown on the new diagram. If a class is currently selected, a sequence diagram and a collaboration will be created that represent the behaviour of that class. This means that the created elements will be shown within the class in the explorer hierarchy (under Package-centric view) and model elements created on the diagram will be created within the namespace of the collaboration. A sequence diagram may not only represent the behavior of a class, but also of any other classifier, such as interface, use case, etc. It is also possible to make

sequence diagrams for an operation.

10.6.4. New Collaboration Diagram

This menu entry creates a blank collaboration diagram, and selects that diagram. It also creates a `Collaboration` UML element, which is a container for the elements shown on the new diagram. If a package is selected when this menu item is activated, the collaboration diagram will be created within a collaboration within that package. This means that it will be shown within the collaboration within the package on the explorer hierarchy (under Package-centric view) and model elements created on the diagram will be created within the namespace of the collaboration within the package.

Tip

This does not prevent model elements from other namespaces/packages appearing on the diagram. They can added from the explorer by dragging or by using `Add to Diagram` from the button 2 pop-up menu.

10.6.5. New Statechart Diagram

This menu entry creates a blank statechart diagram associated with the currently selected class, and selects that diagram in the editing pane. It also creates a `Statemachine` UML element, which is a container for the elements shown on the new diagram.

Statechart diagrams are associated with a model element capable of dynamic behavior, such as a classifier or a behavioral feature, which provides the context for the state machine it represents. Suitable model elements are e.g. a class, an operation, and a use case. If such element is not selected at the time the `New Statechart Diagram` menu is activated, then an unattached statemachine is created. To obtain a well-formed UML model, you have to set the context of the statemachine on its details pane.

10.6.6. New Activity Diagram

This menu entry creates a blank activity diagram associated with the currently selected class, and selects that diagram in the editing pane. It also creates a `ActivityGraph` UML element, which is a container for the elements shown on the new diagram.

Activity diagrams are associated with a model element capable of dynamic behavior, such as packages, classifiers (including use cases) and behavioral features. Suitable model elements are e.g. a class, a use case, an operation, and a package. If such element is not selected at the time the `New Activity Diagram` menu is activated, then an unattached ActivityGraph is created. To obtain a well-formed UML model, you have to set the context of the ActivityGraph on its details pane.

10.6.7. New Deployment Diagram

This menu entry creates a blank deployment diagram, and selects that diagram in the editing pane.

Tip

Model elements from other namespaces/packages can be added from the explorer by dragging or by using `Add to Diagram` from the button 2 pop-up menu.

10.7. The Arrange Menu

This menu provides a range of functions to help in the alignment of model elements on diagrams within

the editing pane. In general the menu function invoked is applied to any model element or model elements currently selected in the editing pane.

10.7.1. Align

This sub-menu aligns the selected items. There are seven alignment options provided.

-  `Align Tops`. Aligns the selected model elements by their top edges.
-  `Align Bottoms`. Aligns the selected model elements by their bottom edges.
-  `Align Rights` (Shortcut Ctrl-R). Aligns the selected model elements by their right edges.
-  `Align Lefts` (Shortcut Ctrl-L). Aligns the selected model elements by their left edges.
-  `Align Horizontal Centers`. Aligns the selected model elements so their horizontal centers are in a vertical line.
-  `Align Vertical Centers`. Aligns the selected model elements so their vertical centers are in a horizontal line.
-  `Align To Grid`. Aligns the selected model elements so their top and right edges are on the grid snap boundary (see Section 10.5.5, “Adjust Snap”) edge.

Tip

The alignment is to the current *grid snap* setting, which may be smaller, larger or the same as the displayed grid. Since items are aligned to the grid snap boundary any way when you place them, this menu entry has no effect unless you have either changed the grid snap to a larger value or used one of the other `Arrange` menu entries to push items off their initial positions.

10.7.2. Distribute

This sub-menu distributes the selected items. There are four distribution options provided.

-  `Distribute Horizontal Spacing`. The leftmost and rightmost selected model elements are not moved. The others are adjusted horizontally until the horizontal space (i.e. from the right edge of the left model element to the left edge of the right model element) is the same for all of the selected items
-  `Distribute Horizontal Centers`. The leftmost and rightmost selected model elements are not moved. The others are adjusted horizontally until the distance between the horizontal centers of all the selected items is the same.
-  `Distribute Vertical Spacing`. The top and bottom selected model elements are not moved. The others are adjusted vertically until the vertical space (i.e. from the bottom edge of the

top model element to the top edge of the bottom model element) is the same for all of the selected items

-  `Distribute Vertical Centers`. The top and bottom selected model elements are not moved. The others are adjusted vertically until the distance between the vertical centers of all the selected items is the same.

10.7.3. Reorder

This sub-menu adjusts the ordering of overlapping items. There are four reorder options provided.

-  `Forward`. The selected model elements are moved one step forward in the ordering hierarchy with respect to other model elements they overlap.
-  `Backward`. The selected model elements are moved one step back in the ordering hierarchy with respect to other model elements they overlap.
-  `To Front`. The selected model elements are moved to the front of any other model elements they overlap.
-  `To Back`. The selected model elements are moved to the back of any other model elements they overlap.

10.7.4. Size To Fit Contents

This menu-item acts on all selected items on the current diagram. It resets all sizes of all model elements to its minimum size for which all text fits inside.

10.7.5. Layout

This menu-item provides an automatic diagram layout function, i.e. when activating this menu-item, all items on the current class diagram are rearranged according a certain layout algorithm.

This function currently only works for classdiagrams. For all other types of diagrams, the menu-item is downlighted.

10.8. The Generation Menu

This menu provides support for code generation from UML diagrams. The functionality is built around the structural information of class diagrams.

Note

Without any plugin modules installed, ArgoUML supports only code generation of Java. ArgoUML V0.20 supports the following languages by plugin: C#, C++, php4, php5.

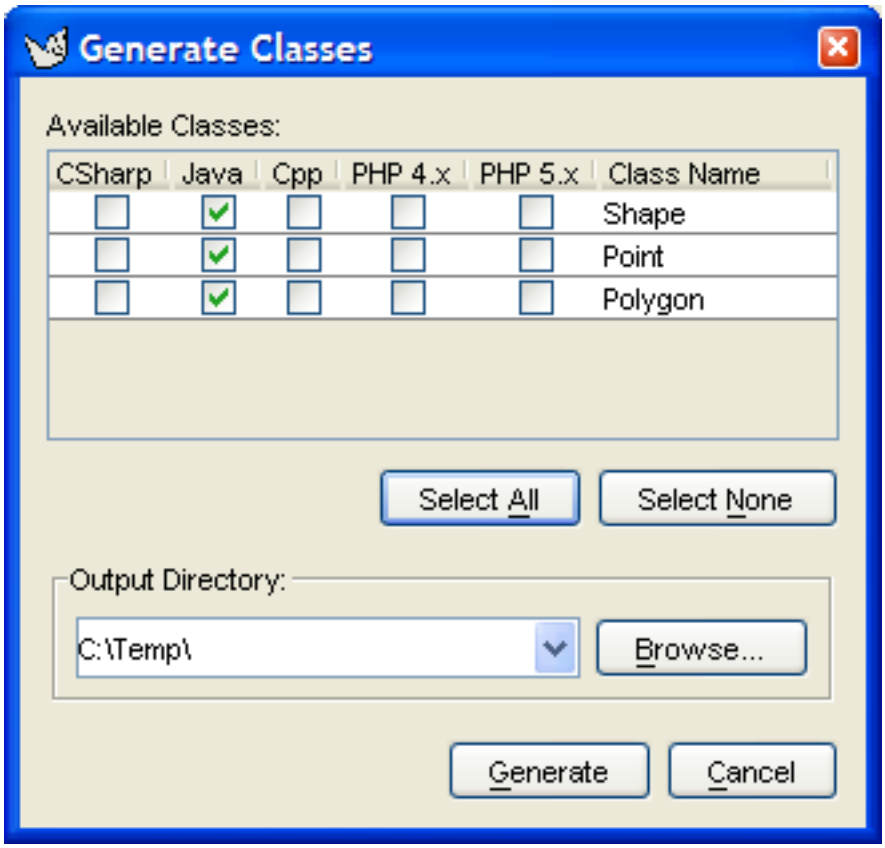
Warning

Code generation is still very much a work in progress. The current version of ArgoUML will generate a structural template for your code, but is not able to handle behavioral specifications to generate code for the dynamic behavior of the model.

10.8.1. Generate Selected Classes ...

This menu entry brings up a dialog box for the ArgoUML code generator (see Figure 10.21, “The dialog for `Generate Selected Classes...`”).

Figure 10.21. The dialog for `Generate Selected Classes...`



Below a label `Available Classes` the dialog box lists each of the selected classes by name with a check box to the left, for each language installed. All the checkboxes are initially unchecked. Checking any of these boxes will cause code generation for that class. Checking multiple languages for a class causes it to be generated in all these languages.

The buttons `Select All` and `Select None` may help when a lot of items have to be selected or deselected.

In the lower portion of the dialog box is an editable combo box labeled `Output Directory` to specify the directory in which code is generated. Within this directory, a top level directory will be created with the name of the model. Further sub-directories will be created to reflect the package/namespace hierarchy of the model. A drop down selector gives access to previously selected output directories.

Finally, at the bottom of the dialog box are two buttons, labeled `Generate` and `Cancel`. Button 1 click on the former will cause the code to be generated, button 1 click on the latter will cancel code generation.

10.8.2. Generate All Classes...

Shortcut F7.

This function behaves as `Generate Selected Classes...` (see Section 10.8.1, “Generate Selected Classes ...”) would with all classes in the current diagram selected.

10.8.3. Generate Code for Project... (To be Written)

10.8.4. Settings for Generate for Project... (To be Written)

10.9. The Critique Menu

This menu controls one of ArgoUML's unique features—the use of critics to guide the designer. The theory behind this is well described in Jason Robbins' PhD dissertation http://argouml.tigris.org/docs/robbins_dissertation/ [http://argouml.tigris.org/docs/robbins_dissertation/].

Note

A word about terminology: The *critics* are background processes, which evaluate the current model according to various “good” design criteria. There is one critic for every design criterion.

The output of a critic is a *critique*—a statement about some aspect of the model that does not appear to follow good design practice.

Finally a critique will generally suggest how the bad design issue it has identified can be rectified, by raising a *to-do item*.

Note

The critics run as asynchronous processes in parallel with the main ArgoUML tool. Changes typically take a second or two to propagate as the critics wake up.

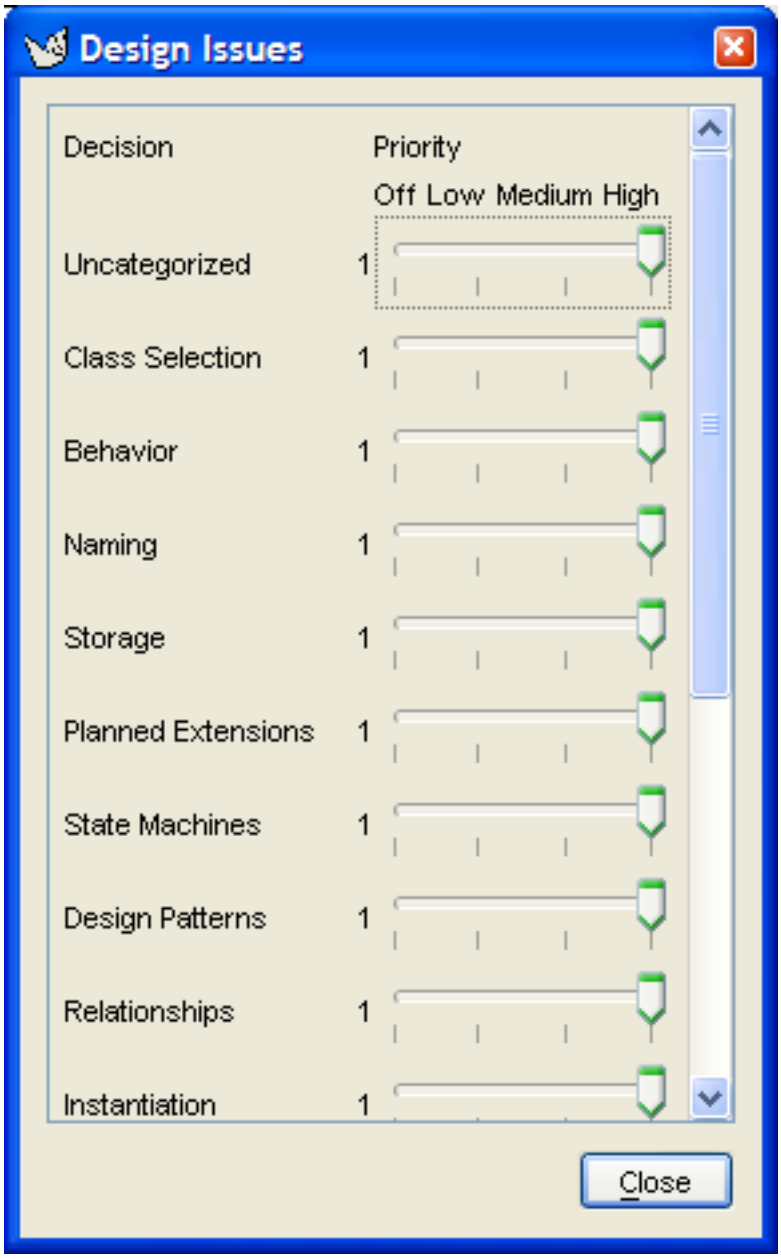
10.9.1. Toggle Auto-Critique

This is a check box, controlling whether the critics are running. By default it is checked. If unchecked, then all critics are disabled, and any to-do items generated by critics (the only others being those the designer has added by hand) are hidden in the to-do pane.

10.9.2. Design Issues...

This menu entry brings up a dialog box controlling how critics associated with a particular design area are to be handled (see Figure 10.22, “The dialog for `Design Issues...`.”).

Figure 10.22. The dialog for `Design Issues...`



ArgoUML categorizes critics according the the design issue they address. There are 16 such categories. The critics in each category are discussed in detail in the chapter on critics (Chapter 15, *The Critics*).

The sliders may be set for each category to control the critics that trigger for that category. Setting a slider to `Off` will disable all critics in that category, and remove all associated to-do items from the to-do pane.

Setting a slider to a higher priority value will enable all critics at or above that priority level within the design issue category (`Off` being the lowest priority).

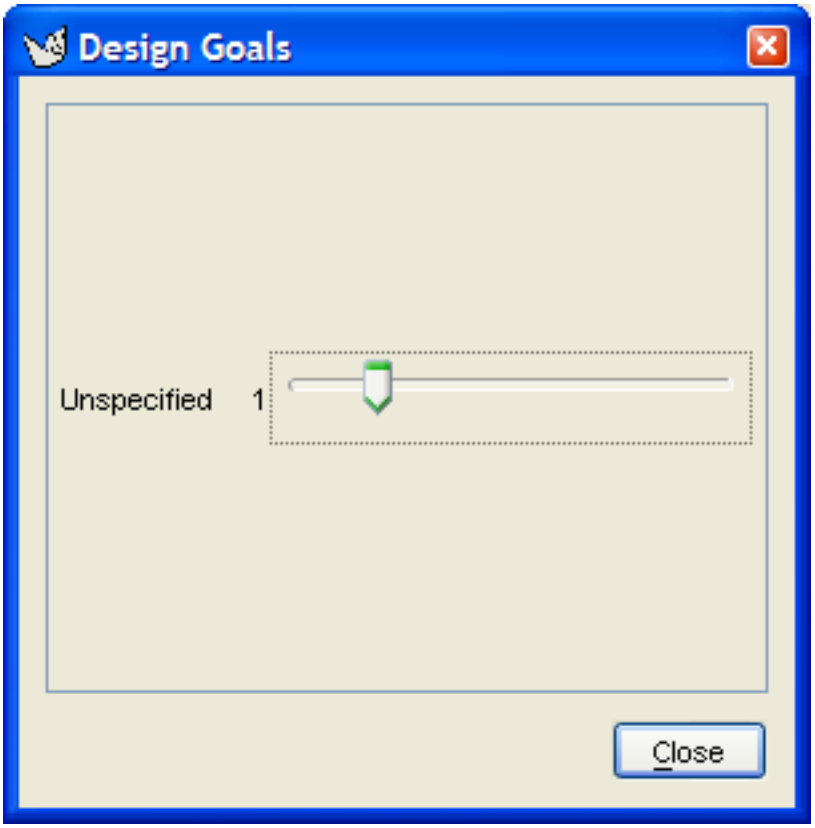
Note

The sliders are set by default to `High` for all design categories.

10.9.3. Design Goals...

This menu entry brings up a dialog box controlling how design goals are to be handled (see Figure 10.23, “The dialog for `Design Goals...`”).

Figure 10.23. The dialog for `Design Goals...`



ArgoUML has the concept that the designer will have a number of design goals to be achieved (for example good structural representation, detailed behavioral representation etc). Critics are associated with one or more goals.

This dialog allows the user to specify the priority of each design goal.

The sliders may be set for each design goal to control the critics that trigger for that goal. Setting a slider to zero will disable all critics in that goal, and remove all associated to-do items from the to-do pane.

Setting a slider to a higher value will enable all critics at or above that priority level within the design issue category (1 being the highest priority and 5 the lowest).

Tip

It may be useful to think of this function as very similar to `Design Issues...` (see Section 10.9.2, “Design Issues...”), but with grouping of critics according to the outcomes of OOA&D rather than grouping according to the structure of UML.

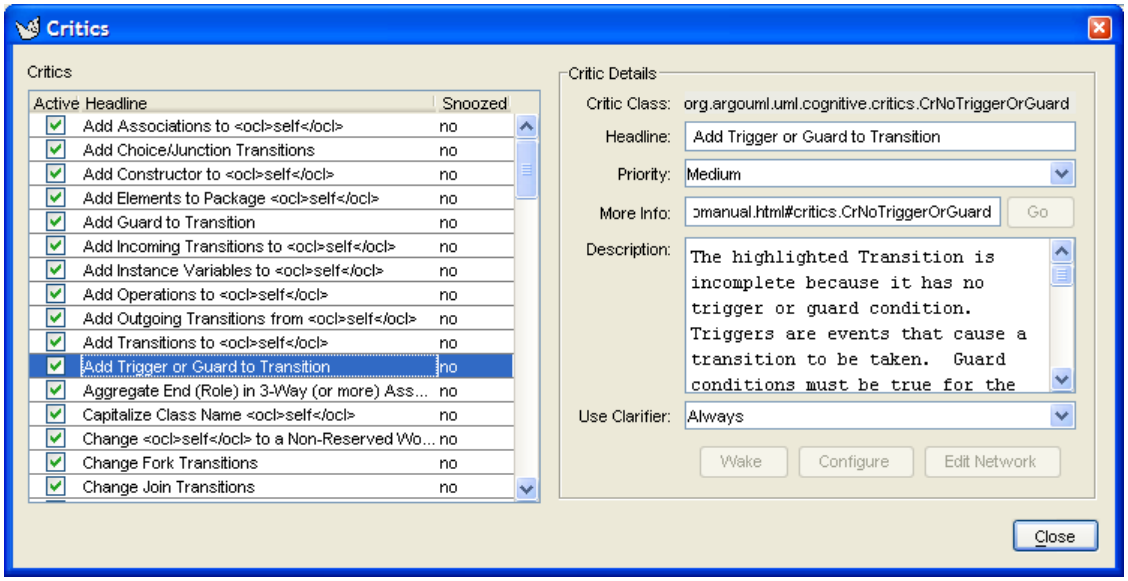
Warning

The V0.20 version of ArgoUML provides a single design goal, `Unspecified`, with its slider set by default to priority 1. However it contains no critics and so has no effect.

10.9.4. Browse Critics...

This menu entry brings up a dialog box controlling the individual critics (see Figure 10.24, “The dialog for `Browse Critics...`”).

Figure 10.24. The dialog for `Browse Critics...`



This dialog controls the behavior of individual critics. To the left is a list of all the critics, to enable them to be switched on or off individually. For each critic there are three columns, labeled `Active`, `Headline` and `Snoozed`. The first of these is a check box, which may be toggled with button 1 click. The second is the headline name of the critic, the third indicates if the critic has been snoozed from the to-do pane (see Chapter 14, *The To-Do Pane*. A critic is only really active if the box in the first column is checked *and* the critic has not been snoozed.

Any critic for which the box in the first column is unchecked is inactive and will not trigger. In addition any to-do items associated with that critic will be removed from the to-do pane.

The V0.20 version of ArgoUML has a total of 90 critics, a few of which are incompletely implemented. They are described in detail by design issue category in the chapter on critics (see Chapter 15, *The Critics*).

To the right of the list are a series of fields, titled `Critic Details`, giving detailed control over individual critics. Selecting a critic in the list on the left will populate the fields for that critic.

The first field on the right is titled `Critic Class:` and then the full name of the class in ArgoUML that implements the critic. This name can be used as unique identifier of the critique, e.g. in conversations about the critic.

The first field below this title is a text box labeled `Headline:` giving the complete headline of the critic (which may be truncated in the list on the left).

Note

In the headline you may see the text `<ocl>self</ocl>`, which will be replaced by the name of the model element in question when the critic is triggered.

The next field is a drop-down selector, labeled `Priority:`. The three options available are `High`, `Medium` and `Low` and specify the priority category of any to-do item generated by this critic. This does not alter the priority of the already existing todo items, only the newly generated ones. Changing the priority of a critic is not saved persistently.

The next field is labeled `MoreInfo:` and contains a URL pointing to further information with a button to the right labeled `Go` to navigate to that URL.

Warning

In the V0.20 release of ArgoUML there is no further information available, and the `Go` button is always grayed out and disabled.

The next field is labeled `Description:` and is a text area with a detailed explanation of what this critic means. If the text is too large for the area a scroll bar is provided to the right.

Note

In this text area you may see the text `<ocl>self</ocl>`, which will be replaced by the name of the model element in question when the critic is triggered.

The last field is a drop-down selector labeled `Use Clarifier`, with three options, `Always`, `If Only One` and `Never`.

Clarifiers are the icons and wavy red underlines drawn on the actual diagrams to indicate the artefact to which the critic refers. The original intention was to make the mapping from critics to clarifiers somewhat customizable.

For example one user might make a `Missing Name` critic show a red underline, another user might turn off the clarifier, or have it draw a wavy green underline or a blue questionmark. Critics with their clarifier's disabled would still produce feedback that is listed in the to-do pane.

Caution

In the V0.20 release of ArgoUML this selector has no function whatsoever. It is for future development.

Underneath the fields are three buttons in a horizontal row.

- `Wake`. It is possible to snooze a critic from the to-do pane (see Chapter 14, *The To-Do Pane*), which makes the critic inactive for a period. If the critic has been snoozed, this button is enabled and will

wake the critic back up again. Otherwise it is grayed out.

Tip

You can tell a snoozed critic, because in the list on the left it will be indicated in the third column.

- `Configure`. This button is for configuring the critic.

Caution

In the V0.20 version of ArgoUML this function is not implemented, and this button is always grayed out. It is for future development.

- `Edit Network`. Right now critics are implemented in java code. That means end-users cannot add new critics.

  The idea of a critic network is that they would be a state machine like diagram with several steps. Each step would express a condition which, collectively with the other steps associated with that critic, articulates the “rule” that the critic is providing. If the rule fires, then remaining steps would define the steps of the wizard to help the user fix the problem.

  The ideas behind this are discussed in Chapter 4 of Jason Robbins PhD disseration (http://argouml.tigris.org/docs/robbins_dissertation/diss4.html. In particular look at Figure 1-6 in this chapter and the related discussion.

  A suggested implementation is that the conditions could be written in OCL against the UML meta-model. A library of predefined conditions and steps would allow end-users to build new critics by combining those in novel ways.

Caution

In the V0.20 version of ArgoUML this function is not implemented, and this button is always grayed out. It is for future development.

Finally the bottom right of the dialog contains a button labeled `OK`. Button 1 click here dismisses the dialog.

10.10. The Tools Menu

This menu provides a generic menu attachment point for any plug-ins provided with ArgoUML. The standard system has no plug-in, and this menu entry is empty by default.

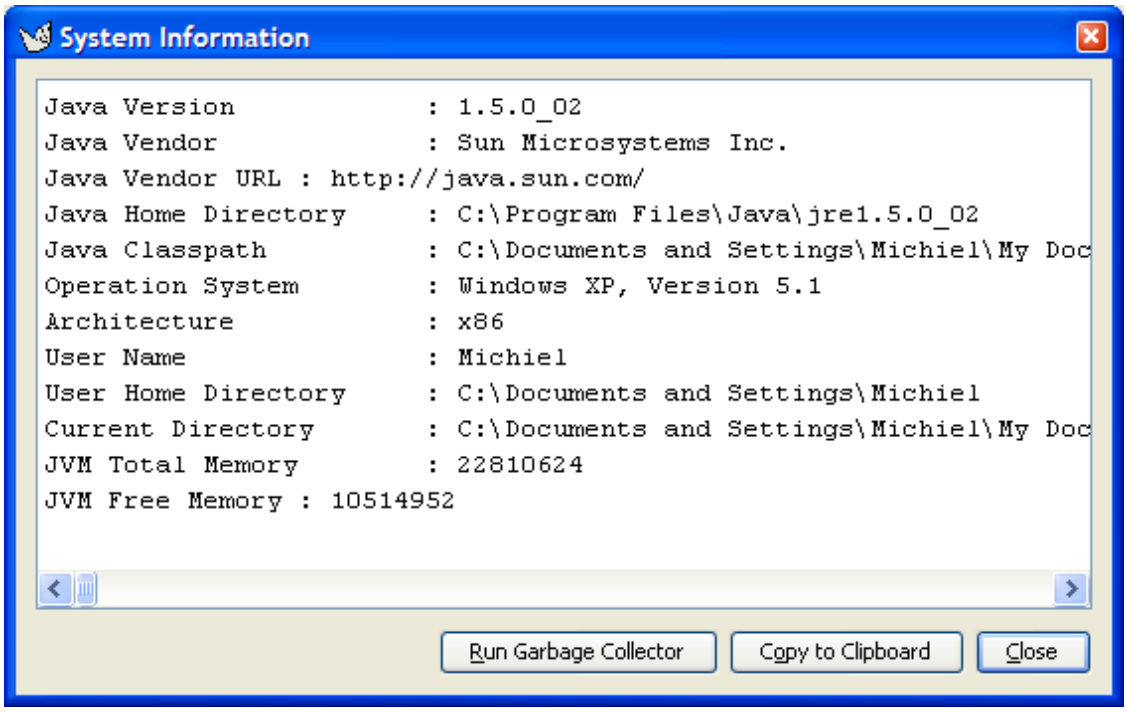
10.11. The Help Menu

This menu provides help on the use of ArgoUML. It has two entries.

10.11.1. System Information

This menu entry brings up the system information dialog, see Figure 10.25, “The dialog for `System Information`.”

Figure 10.25. The dialog for `System Information`.



Use this menu to describe the system that runs ArgoUML to the system manager or developer. Pressing the button `Run Garbage Collector` not only runs the Java gargage collector, but also refreshes the information shown. To facilitate copy and paste into (e.g.) an email, the button `Copy Information to System Clipboard` is foreseen. The `Cancel` button dismisses the dialog box.

10.11.2. About ArgoUML

This menu entry brings up the help window for ArgoUML (see Figure 10.26, “The help window for ArgoUML”).

Figure 10.26. The help window for ArgoUML

The window has six tabs, which are selected by button 1 click. By default the first tab (`Splash`) is shown.

- `Splash`. This displays the picture shown when ArgoUML starts up, and the current version number.
- `Version`. This provides version information on the various packages that make up ArgoUML, and some operating system and environment information.
- `Credits`. This details all those who have created ArgoUML, including contact details for the various module owners.
- `Contact Info`. This gives the major contact points for the ArgoUML project—the web site, and the developers mailing list.
- `Report bugs`. This gives information about how to deal with bugs in ArgoUML. It is important that all bugs are reported, and all cooperation is appreciated.
- `Legal`. A statement of the FreeBSD license which covers all the ArgoUML software.

Caution

The various documentation of the project are not all covered by FreeBSD (which is really meant for software). In particular this manual is covered by the OpenPub license

(see Appendix F, *Open Publication License*).

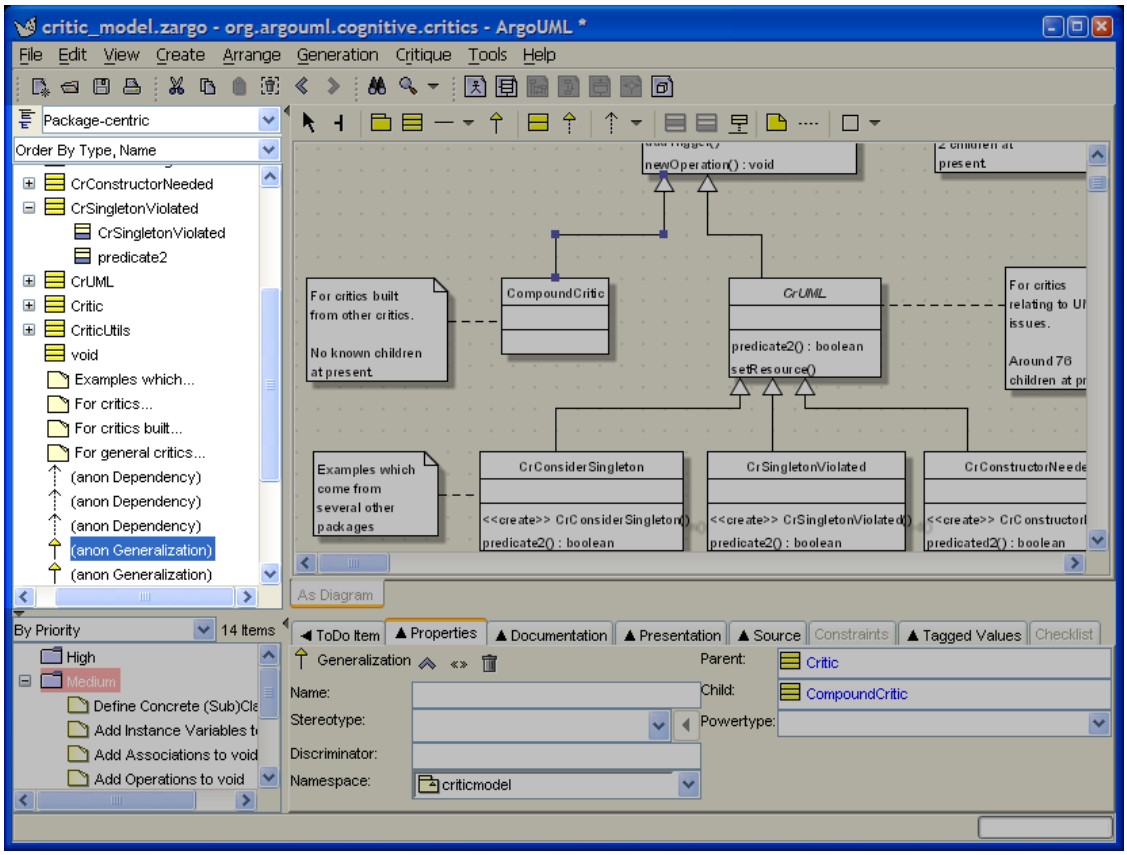
Chapter 11. The Explorer

The Explorer was previously called Navigation Pane/Tree or sometimes Navigator Pane/Tree.

11.1. Introduction

Figure 11.1, “Overview of the explorer” shows the ArgoUML window, with the explorer highlighted.

Figure 11.1. Overview of the explorer



The explorer allows the user to view the structure of the model from a number of predefined perspectives. It also allows the user to define their own perspetives for custom exploring of the model.

An important feature, related to the cognitive psychology ideas behind ArgoUML is that not all model elements are necessarily shown in all perspectives. Rather, the perspectives are used to implement hiding of uninteresting parts of the model.

11.2. Mouse Behavior in the Explorer

Behavior of the mouse in general, and the naming of the buttons is covered in the chapter on the overall user interface (see Chapter 8, *Introduction*).

11.2.1. Button 1 Click

Within the hierarchical display, elements which have sub-hierarchies are indicated by  when the hierarchy is hidden and  when the hierarchy is open.

Button 1 click over the name of any diagram model element will cause the diagram to be selected and displayed in the editing pane. Its details will also be displayed in the details pane.

Button 1 click over the name of any model element other than a diagram in the main area of the explorer will cause it to be selected, and its details shown in the details pane. If the model element is part of a diagram currently displayed in the editing pane, it will be highlighted there.

Note

If the model element is part of a diagram other than that currently displayed in the Editing Pane, there will be *no* change of diagram in the Editing Pane.

Where button 2 click has been used to bring up a context sensitive pop-up menu (see below), button 1 click is used to select the menu entry required. button 1 click outside the menu area will remove it.

11.2.2. Button 1 Double Click

This has the effect of a button 1 single click, and if the tree item was not a leaf, it will toggle the hierarchy open or close.

11.2.3. Button 1 Motion

Button 1 motion means that you pick up one or more modelelements, and drag them to a new location. Dropping the modelelement somewhere causes ArgoUML to execute some function that depends on where you drop the modelelements.

11.2.3.1. From Explorer to Explorer

Releasing the mouse button above a namespace, makes the modelelement owned by the namespace. In the Package-centric explorer perspective, this means a straigh-forward drag-and-drop function.

Use this drap and drop feature to easily move e.g. classes from one package into another.

11.2.3.2. From Explorer to Diagram

Dropping a modelelement on the diagram is the equivalent of the "Add to Diagram" function. Hence, if the diagram did not yet show this modelelement, it is added.

Use this drap and drop feature e.g. to easily create a diagram from imported XMI files. This because XMI files contain all the modelelements, but not any diagram information.

11.2.4. Button 2 Actions

When used in the the explorer, this will display a selection dependent pop-up menu. Menu entries are highlighted (but not selected) and sub-menus exposed by subsequent mouse motion (without any buttons). Menu entry selection is with button 1 or button 2.

11.2.5. Button 2 Double Click

This has no effect other than that of button 2 single click.

11.3. Keyboard Behavior in the Explorer

All keys active in a tree widget have their normal behaviour.

When a diagram is selected, pressing Ctrl-C will copy the diagram in GIF format to the system clipboard.

11.4. Perspective Selection

The model elements in the ArgoUML model may be configured for displaying in the tree by a number of perspectives. To this end, a drop-down at the top allows selection of the explorer perspective.

Below that, there is a drop-down to select the ordering of the atifacts within the hierarchy. The two possibilities are "Order by Type, Name" and "Order by Name". The former groups all items per type, and sorts them per group alphabetically on the name. The lattter simply sorts on name only.

The following explorer perspectives may be selected in the drop-down at the top:

- `Package-centric` (the default). The exploring hierarchy is organized by package hierarchy. The top level shows the model. Under this are all the top level packages in the model and all the model elements that are directly in the namespace of the model.

  Beneath each package are all the model elements that sit within the namespace of that package, including any further sub-packages (which in turn have their own sub-hierarchies).

- `Class-centric`. Shows classes in their package hierarchy as well as datatypes and use case diagram elements. Similar to the Package-centric view but it doesn't show connecting or associating elements.

- `Diagram-centric`. In this view the top level comprises all the diagrams in the model. Beneath each diagram is a flat listing of all the model elements on the diagram. Model elements that have sub-model elements that do not appear on the diagram have their own hierarchy (for example attributes and operations of classes).

- `Inheritance-centric`. In this view the top level shows the model. Beneath this are all model elements that have no generalization in the model. Those model elements that have specializations have a sub-hierarchy showing the specializations.

- `Class Associations`. In this view the top level shows the model. Beneath this are all diagrams and all classes. All classes that have associations have a hierarchy tracking through the associated classes.

- `Residence-centric`. In this view the model is shown at the top-level, with below it only Nodes, and below these only components that reside on the nodes, and below these components all elements that reside on the components.

- `State-centric`. In this view the top level shows all the state machines and all activity graphics associated with classes.

  Beneath each state machine is a hierarchy showing the statechart diagram and all of the states. Beneath each state is a list of the transitions in and out of the state.

  Beneath each activity graph is a hierarchy showing the activity diagram and all of the action states. Beneath each action state is a list of the transitions in and out of the action state.

- `Transitions-centric`. This is very similar to `State-centric` view, but under each state machine is listed the diagrams and all transitions on the diagram, with states being shown as sub-hierarchies under their connected transitions.

  Similarly under each activity graph is listed the diagrams and all transitions on the diagram, with action states being shown as sub-hierarchies under their connected transitions.

- `Composite-centric`. In this view, all modelelements are shown according their composition in the UML metamodel.

  This perspective shows far more modelelements then all others - it does not hide anything. Hence, this view is not so user-friendly, but very suited for the UML specialist.

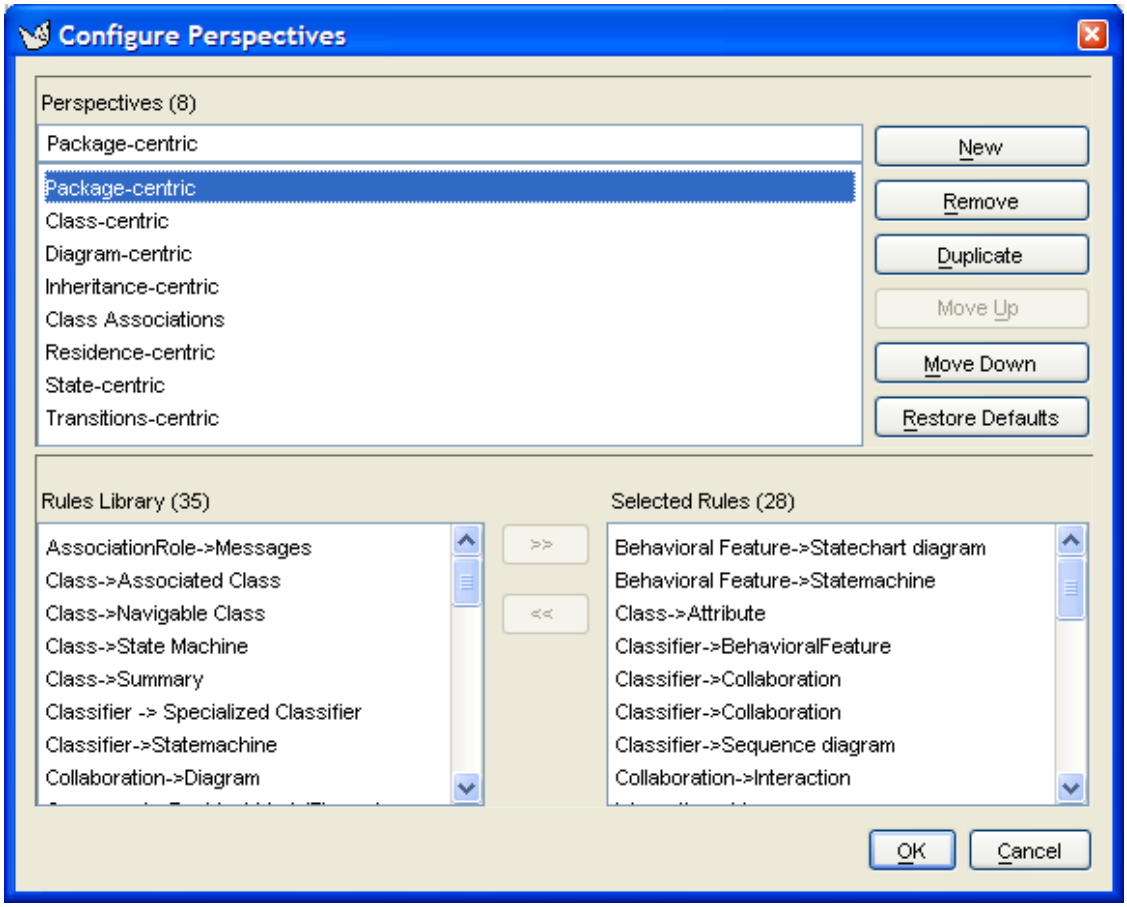
11.5. Configuring Perspectives

The explorer is designed to be user configurable, to allow the designer to view in his or her preferred way.

11.5.1. The Configure Perspectives dialog

button 1 click on the "Configure Perspectives" icon () at the top left of the explorer brings up the explorer perspectives dialog (see Figure 11.2, “The Configure Perspectives dialog box”).

Figure 11.2. The Configure Perspectives dialog box



The top half of the dialog contains a list of all the currently defined perspectives and to the right a series of buttons stacked vertically. Button 1 click can be used to select a perspective. You can select only one perspective at a time.

Selecting a perspective reveals a text field above the list, where the name of the perspective can be edited.

The lower half of the dialog contains two list areas. The one on the left, labeled `Rules Library`, contains the list of available rules that may be used to create the perspective. The one on the right, labeled `Selected Rules` contains the actual rules chosen for the perspective that has been selected in the list of perspectives at the top. In both lists, you can select only one rule at a time.

Separating the two areas in the lower half of the dialog are buttons labeled >> and <<. The first of these transfers the rule selected in the library on the left to the list of rules on the right—i.e. it adds a rule to the perspective. The second one transfers the rule selected on the right to the library list on the left—i.e. it removes a rule from the perspective.

If you hover the mouse over the horizontal line that separates the two halves of the dialog, then you see it change shape, to indicate that you can grab this line and drag it up or down.

All three titles of the lists show the number of items in the list. ArgoUML V0.24 has 9 default perspectives, and 72 rules in the library to build perspectives from.

The buttons at the top right are explained as follows:

- `New`. This creates a new perspective from scratch with no rules selected, with an automatically generated name.
- `Remove`. This removes the selected perspective.
- `Duplicate`. This creates a copy the selected perspective so it can be used as the basis of a new perspective. The new one is named "Copy of" followed by the original name.
- `Move Up`. This moves the selected perspective one place up in the list. This button is downlighted for the topmost perspective.
- `Move Down`. This moves the selected perspective one place down in the list. This button is downlighted for the last perspective.
- `Restore Defaults`. This restores all perspectives and their selected rules to the build-in defaults of ArgoUML.

At the very bottom right is a button labeled `OK` to be used when all changes are complete. button 1 click on this button will close the dialog window. The changes are saved when you exit ArgoUML in the `argo.user.properties` file.

Then there is the `Cancel` button, which cancels all changes made in the dialog. Pressing the dialog close icon (usually at the top right corner) has the same effect as pressing the cancel button.

11.6. Context Sensitive Menu

Button 2 Click over any selected model element in the main area of the explorer will cause a pop-up menu to appear.

11.6.1. Create Diagram

This entry on the pop-up menu opens a choice of submenus, one for each diagram type.

The namespace of the new diagram will be based on the selected modelelement.

11.6.2. Copy Diagram to Clipboard as Image

This entry on the pop-up menu creates a graphical file, in the default graphical format, and puts it on the clipboard of your PC. The graphics can inmediately be pasted into e.g. a requirements document in OpenOffice.Org.

The graphics format and its resolution are determined by ArgoUML's default setting: Select in the menu `Edit`, then `Settings...`, then the tab `Environment`. The PNG and GIF formats are advised, and the resolution `Standard`.

Tip

Some applications (such as Doors from Telelogic) require the background color of the generated graphics to be adapted (else the image is empty). This can be done with a tool like IrfanView; it is as easy as clicking its paste button, and then its copy button.

11.6.3. Add to Diagram

This entry on the pop-up menu appears for any model element that could be added to the diagram in the

editing pane.

The item can be placed in a diagram by moving the cursor to the editing pane or a spawned editing pane window (where it will appear as a cross) and clicking button 1.

Caution

This menu entry only appears as not grayed out, if the diagram in the editor pane allows to contain the model element, and the model element is not present yet in the diagram. ArgoUML will not let you place more than one copy of any particular model element on a diagram.

11.6.4. Delete From Model

This entry on the pop-up menu appears for any model element that could be deleted from the model.

Warning

This deletes the model element from the model completely, not just from the diagram. To remove the model element just from the diagram, use the edit menu (see Section 10.4.2, “ Remove From Diagram”).

Caution

You can delete a diagram from the model. Depending on the type of diagram, that might delete all model elements shown on the diagram. To illustrate the differences, consider the following examples:

- Deleting a class diagram does not delete any model element drawn on it. All model elements that were shown on the diagram remain present in the model. This because a class diagram does not "map" on any model element according the UML standard V1.4.
- Deleting a statechart diagram also deletes the statemachine it represents, and hence also all the model elements owned by the statemachine. This because a statechart diagram does "map" into a StateMachine according the UML standard V1.4.

11.6.5. Set Source Path... (To be written)

This entry on the pop-up menu ...

11.6.6. Add Package

This entry on the pop-up menu is available whenever a model element is selected that may contain a package, e.g. a package. After activating this menu the model element will own a new package.

11.6.7. Add All Classes in Namespace

This entry on the pop-up menu is available for Class Diagrams only. Activating this menu-item will add all classes in the current namespace to the diagram. They will be located at the top left

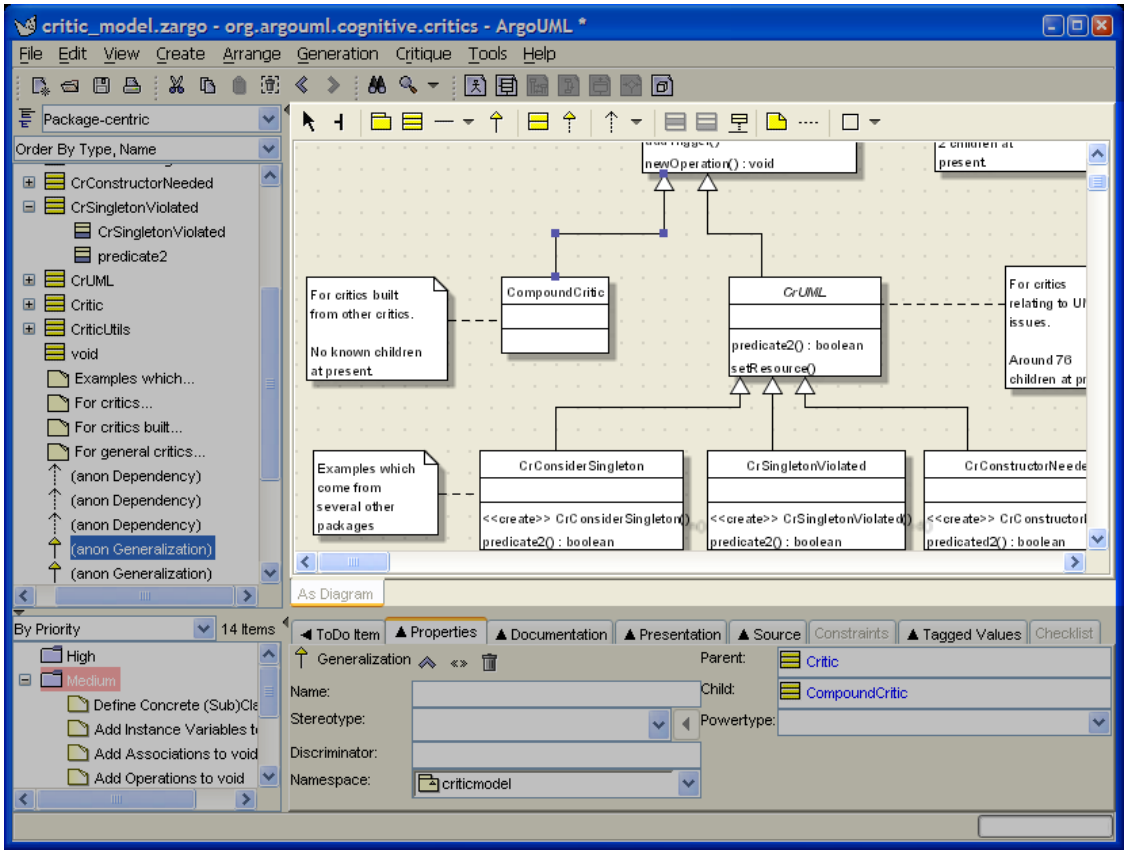
corner—obviously a perfect occasion to use the “Arrange->Layout” function in the menu.

Chapter 12. The Editing Pane

12.1. Introduction

Figure 12.1, “Overview of the editing pane” shows the ArgoUML window with the editing pane highlighted.

Figure 12.1. Overview of the editing pane



This is where all the diagrams are drawn. In earlier versions of ArgoUML this pane went under a variety of names. You may encounter “drawing pane”, “diagram pane” or “multi-editor pane” in other documentation that is still being updated.

The pane has a tool bar at the top, and a single tab labeled `As Diagram` at the bottom, which has no function in the 0.20 version of ArgoUML. The main area shows the currently selected diagram, of which the name is shown in the window title bar.

12.2. Mouse Behavior in the Editing Pane

Behavior of the mouse in general, and the naming of the buttons is covered in the chapter on the overall user interface (see Chapter 8, *Introduction*).

12.2.1. Button 1 Click

In the tool bar of the editing pane, button 1 click is used to select a tool for creating a new model element and adding it to the diagram (see double clicking for creating multiple model elements). For most tools, adding a new model element to the diagram is achieved by moving the mouse into the editing area and clicking again.

In the main editing area button 1 click is used to select an individual model element.

Many model elements (e.g. actor, class) show special handles when selected and the mouse hovers over them. These are called “Selection Action Buttons”, see Section 12.6, “Selection Action Buttons”. They appear at the sides, top and bottom, and indicate a relationship type. Clicking on a Selection Action Button creates a new related model element, with the relation of the type that was indicated. If the shift key is pressed when hovering the mouse over a selected model element, sometimes different handles are shown, which stand for different relation types.

Where button 2 click has been used to bring up a context sensitive pop-up menu (see below), button 1 click is used to select the menu entry required. The pop-up menu will be removed by any button 1 click outside of the menu area.

There are various more detailed effects, which are discussed under the descriptions of the various tools (see Section 12.4, “The tool bar”).

12.2.2. Button 1 Double Click

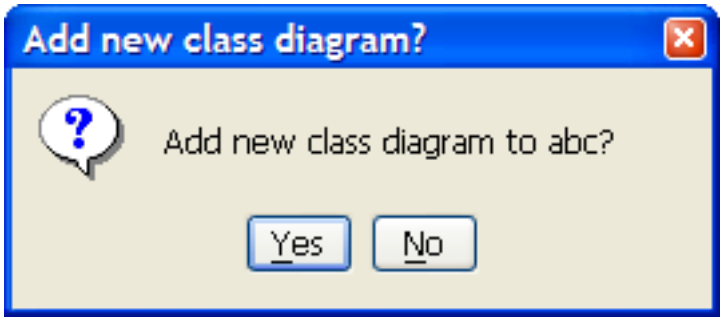
When used on the tool bar with a tool to add a model element, the selected model element will be added multiple times to the drawing area, once for each further button click, until the tool is again selected or another tool chosen.

When used within the drawing area on a model element that has sub-components, double click will select the sub-component for editing (creating it if necessary).

For example double clicking over an operation compartment of a class will select the operation. Or create one if there is none yet.

A special use is with package model elements on the class diagram. A double click on a package will navigate to the class diagram associated with a package (the first created if there is more than one), or will offer to create one for you if there is none. See Figure 12.2, “The dialog for adding a new class diagram”

Figure 12.2. The dialog for adding a new class diagram



12.2.3. Button 1 Motion

Where the model element being added is some form of connector its termination point is shown with button 1 up over the terminating model element. button 1 click may be used in the space between model elements to create articulation points in the connector. This is particularly useful where connectors must loopback on themselves.

Over graphical model elements button 1 motion will move the model element to a new position.

Graphical model elements that are selected show handles at the corners or ends, and these can be used for re-sizing.

Some model elements (e.g. actor, class) show special handles (called “Selection Action Buttons”, see Section 12.6, “Selection Action Buttons”) at the sides, top and bottom, which can be dragged to form types of relationship with other model elements.

Where the model element is some form of connector between other items, button 1 motion other than at a handle will cause a new handle to be created, allowing the connector to be articulated at that point. This only works when the connecting line is not straight angled. Such new handles can be removed by moving them to the end of the connector.

There are various more detailed effects, which are discussed under the descriptions of the various tools (see Section 12.4, “The tool bar”).

12.2.4. Shift and Ctrl modifiers with Button 1

Where multiple selections are to be made, the CTRL key is used with button 1 to *add* unselected model elements to the current selection. Where a model element is already selected, it is removed from the current selection.

Clicking Button 1 while the SHIFT key is pressed, invokes the broom tool, which causes the selected model elements (and any others swept up with them) to be moved with the broom tool (see Section 12.4.1, “Layout Tools”).

12.2.5. Alt with Button 1 motion

Button 1 down anywhere in the diagram while the ALT key is pressed, allows to scroll the canvas in all directions with button 1 motion.

12.2.6. Button 2 Actions

When used over model elements in the the editing pane, this will display a context dependent pop-up menu. Menu entries are highlighted (but not selected) and sub-menus exposed by subsequent mouse motion (without any buttons). Menu entry selection is with button 1 or button 2. See Section 12.10, “Pop-Up Menus” for details of the specific pop-up menus.

In case multiple elements are selected, the pop-up menu only appears if all the items are of the same kind. In this case, the functions apply to all selected elements.

12.2.7. Button 2 Double Click

This has no effect other than that of button 2 single click.

12.2.8. Button 2 Motion

This is used to select items in a context sensitive menu popped up by use of button 2 click.

12.3. Keyboard Behavior in the Editing Pane

Many keyboard shortcuts can be used when the editing pane is active, mainly to change the selection or to move across the model elements.

12.3.1. Nudging a model element

You can nudge a fig by selecting an element and using arrow keys. Keeping the shift or the alt key pressed will produce a wider movement.

12.3.2. Moving across the model elements

You can select the model element nearer to the selected one by using the arrow keys while clicking the right button of the mouse. You can also select the next model element using the tab key, or the previous using the ctrl+tab keys.

12.4. The tool bar

The toolbar at the top of the editing pane provides the main functions of the pane. The default tool is the `Select` tool (). In general button 1 click on any tool selects a tool for one use, before reverting to the default tool, and button 1 double click selects a tool for repeated use.

The tools fall into four categories.

- *Layout tools.* Provide assistance in laying out model elements on the diagram.
- *Annotation tools.* Used to annotate model elements on the diagram.
- *Drawing tools.* Used to add general graphic objects to diagrams.
- *Diagram specific tools.* Used to add UML model elements specific to a particular diagram type to the diagram.

Some of the tools that are generally not all used so often, are combined in a dropdown, to take less space on the toolbar. See e.g. Figure 12.3, “The drawing tools selector.”. Press the symbol at the right of the tool to pop it open. These drop-down tools remember their last used tool persistently. This means that when ArgoUML starts, they show the last tool that was activated the previous time ArgoUML was run.

12.4.1. Layout Tools

The following two tools are provided in all diagrams in this category.

-  `Select`. This tool provides for general selection of model elements on the diagram. Button 1 click will select a model element. CTRL with button 1 can be used to select (or deselect) multiple model elements. Button 1 motion will move selected 2D items or add and move a new control point on a link. Button 1 motion on a selected component's control point will stretch that component's shape.
-  *Broom.* Button 1 motion with this tool provide a “broom” which will sweep all model elements along. This is a very shortcut way of lining things up.

The Broom can also be invoked by using SHIFT with button 1 motion when the `Select` tool is in use.

The Broom is discussed at length in its own chapter, see Section 12.5, “The Broom”

Tip

Additional control of model element layout is provided through the `Arrange` menu (see Section 10.7, “The Arrange Menu”).

12.4.2. Annotation Tools

The annotation tool `Comment` () is used to add a comment to a selected UML model element.

Caution

Unlike most other tools you use the `Select` tool to select a model element, and then button 1 click on `Comment` to create the comment. If no element is selected when the comment tool is clicked, then the comment is created and put at the left top corner.

The comment is created alongside the selected model element, empty by default. The text can be selected with button 1 double-click and edited from the keyboard.

The UML standard allows comments to be attached to any model element.

You can link any comment to aditional elements using the `CommentLink` ( ) tool.

12.4.3. Drawing Tools

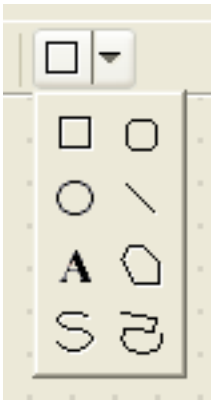
These are a series of tools for providing graphical additions to diagrams. Although they are not UML model elements, the UML standard provides for such decoration to improve the readability of diagrams.

Tip

These drawing tools provide a useful way to partially support some of the UML features (such as general purpose notes) that are missing from the current release of ArgoUML.

Eight tools are provided, all grouped into one drop-down widget. See Figure 12.3, “The drawing tools selector.”. Button 1 click on the diagram will place an instance of the graphical item of the same size as the last one placed. The size can be controlled by button 1 motion during placement. One side or end of the element will be at button 1 down, the other side or end at button 1 up. In general after they are placed on the diagram, graphical elements can be dragged with the `Select` tool and button 1 and re-sized by button 1 motion on the handles after they have been selected.

Figure 12.3. The drawing tools selector.



-  `Rectangle`. Provides a rectangle.
-  `Rounded Rectangle`. Provides a rectangle with rounded corners. There is no control over the degree of rounding.
-  `Circle`. Provides a circle.
-  `Line`. Provides a line.
-  `Text`. Provides a text box. The text is entered by selecting the box and typing. Text is centered horizontally and after typing, the box will shrink to the size of the text. However it can be re-sized by dragging on the corners.
-  `Polygon`. Provides a polygon. The points of the polygon are selected by button 1 click and the polygon closed with button 1 double click (which will link the final point to the first point).
-  `Spline`. Provide an open spline. The control points of the spline are selected with button 1 and the last point selected with button 1 double click.
-  `Ink`. Provide a polyline. The points are provided by button 1 motion.

12.4.4. Use Case Diagram Specific Tools

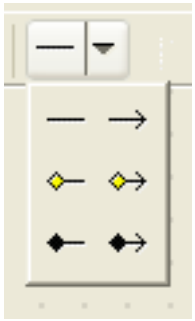
Several tools are provided specific to UML model elements on use case diagrams. The detailed properties of these model elements are described in the section on use case diagram model elements (see Chapter 17, *Use Case Diagram Model Element Reference*).

-  `Actor`. Add an actor to the diagram. For convenience, when the mouse is over a selected actor it displays two handles to left and right which may be dragged to form association relationships.
-  `Use Case`. Add a use case to the diagram. For convenience, when the mouse is over a selec-

ted use case it displays two handles to left and right which may be dragged to form association relationships and two handles top and bottom which may be dragged to form generalization and specialization relationships respectively.

- `Association`. Add an association between two model elements selected using button 1 motion (from the first model element to the second). There are 6 types of association offered here, see Figure 12.4, “The association tool selector.”: `association`, `aggregation` and `composition`, and all these three can be `bidirectional` or `unidirectional`.

Figure 12.4. The association tool selector.



- `Dependency`. Add a dependency between two model elements selected using button 1 motion (from the dependent model element).
- `Generalization`. Add a generalization between two model elements selected using button 1 motion (from the child to the parent).
- `Extend`. Add an extend relationship between two model elements selected using button 1 motion (from the extended to the extending use case).
- `Include`. Add an include relationship between two model elements selected using button 1 motion (from the including to the included use case).
- `Add Extension Point`. Add an extension point to a selected use case. The extension point is given the default name `newEP` and location `loc`. Where the extension point compartment is displayed, the extension point may be edited by button 1 double click and using the keyboard, or by selecting with button 1 click (after the use case has been selected) and using the property tab. Otherwise it may be edited through its property tab, selected through the property tab of the owning use case.

Note

This tool is grayed out except when a use case is selected.

12.4.5. Class Diagram Specific Tools

Several tools are provided specific to UML model elements on class diagrams. The detailed properties of these model elements are described in the section on class diagram model elements (see Chapter 18, *Class Diagram Model Element Reference*).

-  `Package`. Add a package to the diagram.
-  `Class`. Add a class to the diagram. For convenience, when the mouse is over a selected class it displays two handles to left and right which may be clicked or dragged to form association relationships (or composition in case SHIFT has been pressed) and two handles top and bottom which may be dragged or clicked to form generalization and specialization relationships respectively.
- `Association`. Add an association between two model elements selected using button 1 motion (from the first model element to the second). There are 2 types of association offered here, `bidirectional` or `unidirectional`.
-  `Aggregation`. Add an aggregation between two model elements selected using button 1 motion (from the first model element to the second). There are 2 types of aggregation offered here, `bidirectional` or `unidirectional`.
-  `Composition`. Add an composition between two model elements selected using button 1 motion (from the first model element to the second). There are 2 types of composition offered here, `bidirectional` or `unidirectional`.
-  `Association-end`. Add another end to an already existing association using button 1 (from the association middle to a class, or vice versa). This is the way to create so-calld N-ary associations.
-  `Generalization`. Add a generalization between two model elements selected using button 1 (from the child to the parent).
-  `Interface`. Add an interface to the diagram. For convenience, when the mouse is over a selected interface it displays a handle at the bottom which may be dragged to form a realization relationship (the target being the realizing class).
-  `Realization`. Add a realization between a class and an interface selected using button 1 motion (from the realizing class to the realized interface).
-  `Dependency`. Add a dependency between two model elements selected using button 1 motion (from the dependent model element). There are also 2 special types of dependency offered here, `Permission` () and `Usage` (). A `Permission` is created by default with stereotype `Import`, and is used to import elements from one package into another.
-  `Attribute`. Add a new attribute to the currently selected class. The attribute is given the default name `newAttr` of type `int` and may be edited by button 1 double click and using the keyboard, or by selecting with button 1 click (after the class has been selected) and using the property tab.

Note

This tool is grayed out except when a class is selected.

-  `Operation`. Add a new operation to the currently selected class or interface. The operation is given the default name `newOperation` with no arguments and return type `void` and may be edited by button 1 double click and using the keyboard, or by selecting with button 1 click (after the class has been selected) and using the property tab.

Note

This tool is grayed out except when a class or interface is selected.

-  `Association Class`. Add a new association class between two model elements selected using button 1 motion (from the first model element to the second).
-  `Datatype`. Add a datatype to the diagram. For convenience, when the mouse is over a selected datatype it displays handles at the top and at the bottom which may be clicked or dragged to form a generalization relationship (the target being another datatype). There are 2 other elements available here,  `Enumeration` and  `Stereotype`. These two have similar handles, except the one at the top of a stereotype: when clicked, it creates a metaclass, connected by a dependency marked with «stereotype». This eases the creation of "stereotype declaration" diagrams - see the literature on the subject.

12.4.6. Sequence Diagram Specific Tools

Seven tools are provided specific to UML model elements on sequence diagrams. The detailed properties of these model elements are described in the section on sequence diagram model elements (see Chapter 19, *Sequence Diagram Model Element Reference*).

-  `ClassifierRole`. Add a classifierrole to the diagram.
-  `Message with Call Action`. Add a call message between two classifierroles selected using button 1 motion (from the originating classifierrole to the receiving classifierrole).
-  `Message with Return Action`. Add a return message between two classifierroles selected using button 1 motion (from the originating classifierrole to the receiving classifierrole).
-  `Message with Create Action`. Add a create message between two classifierroles selected using button 1 motion (from the originating classifierrole to the receiving classifierrole).
-  `Message with Destroy Action`. Add a destroy message between two classifierroles selected using button 1 motion (from the originating classifierrole to the receiving classifierrole).
-  `Add Vertical Space to Diagram`. Add vertical space to a diagram by moving all mes-

sages below this down. Click the mouse at the point where you want the space to be added and drag down the screen vertically the distance which matches the height of the space you'd like to have added.

-  `Remove Vertical Space in Diagram`. Remove vertical space from diagram and move all elements below up vertically. Click and drag the mouse vertically over the space that you want deleted.

12.4.7. Collaboration Diagram Specific Tools

Three tools are provided specific to UML model elements on collaboration diagrams. The detailed properties of these model elements are described in the section on collaboration diagram model elements (see Chapter 21, *Collaboration Diagram Model Element Reference*).

-  `Classifier Role`. Add a classifier role to the diagram.
- `Association Role`. Add an association role between two classifier roles selected using button 1 motion (from the originating classifier role to the receiving classifier role). There are 6 types of association roles offered here, see Figure 12.4, “The association tool selector.”: `association`, `aggregation` and `composition`, and all these three can be `bidirectional` or `unidirectional`.
-  `Generalization`. Add a generalization between two model elements selected using button 1 (from the child to the parent).
-  `Dependency`. Add a dependency between two model elements selected using button 1 motion (from the dependent model element).
-  `Add Message`. Add a message to the selected association role.

Note

This tool is grayed out except when an association role is selected.

12.4.8. Statechart Diagram Specific Tools

Eleven tools are provided specific to UML model elements on statechart diagrams. The detailed properties of these model elements are described in the section on statechart diagram model elements (see Chapter 20, *Statechart Diagram Model Element Reference*).

-  `Simple State`. Add a simple state to the diagram.
-  `Composite State`. Add a composite state to the diagram. All model elements that are subsequently placed on the diagram on top of the composite state will form part of that composite state.

-  `Transition`. Add a transition between two states selected using button 1 motion (from the originating state to the receiving state).
-  `Synch State`. Add a synchstate to the diagram.
-  `Submachine State`. Add a submachinestate to the diagram.
-  `Stub State`. Add a stubstate to the diagram.
-  `Initial`. Add an initial pseudostate to the diagram.

  ## Caution

  There is nothing to stop you adding more than one initial state to a diagram or composite state. However to do so is meaningless, and one of the critics will complain.

-  `Final State`. Add a final state to the diagram.
- `Junction`. Add a junction pseudostate to the diagram.

  ## Caution

  A well formed junction should have at least one incoming transition and at least one outgoing. ArgoUML does not enforce this, but an ArgoUML critic will complain about any junction that does not follow this rule.

-  `Choice`. Add a choice pseudostate to the diagram.

  ## Caution

  A well formed choice should have at least one incoming transition and at least one outgoing. ArgoUML does not enforce this, but an ArgoUML critic will complain about any choice that does not follow this rule.

-  `Fork`. Add a fork pseudostate to the diagram.

  ## Caution

  A well formed fork should have exactly one incoming transition and two or more outgoing. ArgoUML does not enforce this, but an ArgoUML critic will complain about any fork that does not follow this rule.

-  `Join`. Add a join pseudostate to the diagram.

  ## Caution

A well formed join should have exactly one outgoing transition and two or more incoming. ArgoUML does not enforce this, but an ArgoUML critic will complain about any join that does not follow this rule.

-  `Shallow History`. Add a shallow history pseudostate to the diagram.
-  `Deep History`. Add a deep history pseudostate to the diagram.
-  `Call Event`. Add a Call Event as trigger to a transition. There are 4 types of events offered here: `Call Event`, `Change Event`, `Signal Event` and `Time Event`.
-  `Guard`. Add a guard to a transition.
-  `Call Action`. Add a call action (i.e. the effect) to a transition. There are 7 types of actions offered here: `Call Action`, `Create Action`, `Destroy Action`, `Return Action`, `Send Action`, `Terminate Action`, `Uninterpreted Action` and `Action Sequence`.

12.4.9. Activity Diagram Specific Tools

Seven tools are provided specific to UML model elements on activity diagrams. The detailed properties of these model elements are described in the section on activity diagram model elements (see Chapter 22, *Activity Diagram Model Element Reference*).

-  `Action State`. Add an action state to the diagram.
-  `Transition`. Add a transition between two action states selected using button 1 motion (from the originating action state to the receiving action state).
-  `Initial`. Add an initial pseudostate to the diagram.

Caution

There is nothing to stop you adding more than one initial state to a diagram. However to do so is meaningless, and one of the critics will complain.

-  `Final State`. Add a final state to the diagram.
- `Junction`. Add a junction (decision) pseudostate to the diagram.

Caution

A well formed junction should have one incoming transition and two or more outgoing. ArgoUML does not enforce this, but an ArgoUML critic will complain about any junction that does not follow this rule.

-  `Fork`. Add a fork pseudostate to the diagram.

  ### Caution

  A well formed fork should have one incoming transition and two or more outgoing. ArgoUML does not enforce this, but an ArgoUML critic will complain about any fork that does not follow this rule.

-  `Join`. Add a join pseudostate to the diagram.

  ### Caution

  A well formed join should have one outgoing transition and two or more incoming. ArgoUML does not enforce this, but an ArgoUML critic will complain about any join that does not follow this rule.

-  `CallState`. Add a callstate to the diagram. A call state is an action state that calls a single operation. Hence, the name of the operation being called is put in the symbol, along with the name of the classifier that hosts the operation in parentheses under it.

-  `ObjectFlowState`. Add a objectflowstate to the diagram. An objectflowstate is an object that is input to or output from an action.

12.4.10. Deployment Diagram Specific Tools

Ten tools are provided specific to UML model elements on deployment diagrams. The detailed properties of these model elements are described in the section on deployment diagram model elements (see Chapter 23, *Deployment Diagram Model Element Reference*).

Note

Remember that ArgoUML's deployment diagrams are also used for component diagrams.

-  `Node`. Add a node to the diagram. For convenience, when the mouse is over a selected node it displays four handles to left, right, top and bottom which may be dragged to form association relationships.

-  `Node Instance`. Add a node instance to the diagram. For convenience, when the mouse is over a selected node instance it displays four handles to left, right, top and bottom which may be dragged to form link relationships.

-  `Component`. Add a component to the diagram. For convenience, when the mouse is over a selected component it displays four handles to left, right, top and bottom which may be dragged to form dependency relationships.

-  `Component Instance`. Add a component instance to the diagram. For convenience, when

the mouse is over a selected component instance it displays four handles to left, right, top and bottom which may be dragged to form dependency relationships.

-  `Generalization`. Add a generalization between two model elements selected using button 1 (from the child to the parent).
-  `Realization`. Add a realization between a class and an interface selected using button 1 motion (from the realizing class to the realized interface).
-  `Dependency`. Add a dependency between two model elements selected using button 1 motion (from the dependent model element).
- `Association`. Add an association between two model elements (node, component, class or interface) selected using button 1 motion (from the first model element to the second model element). There are 6 types of association offered here, see Figure 12.4, “The association tool selector.”: `association`, `aggregation` and `composition`, and all these three can be `bidirectional` or `unidirectional`.

  Caution

  The constraint that associations between classes and interfaces must not be navigable *from* the interface still applies on deployment diagrams.

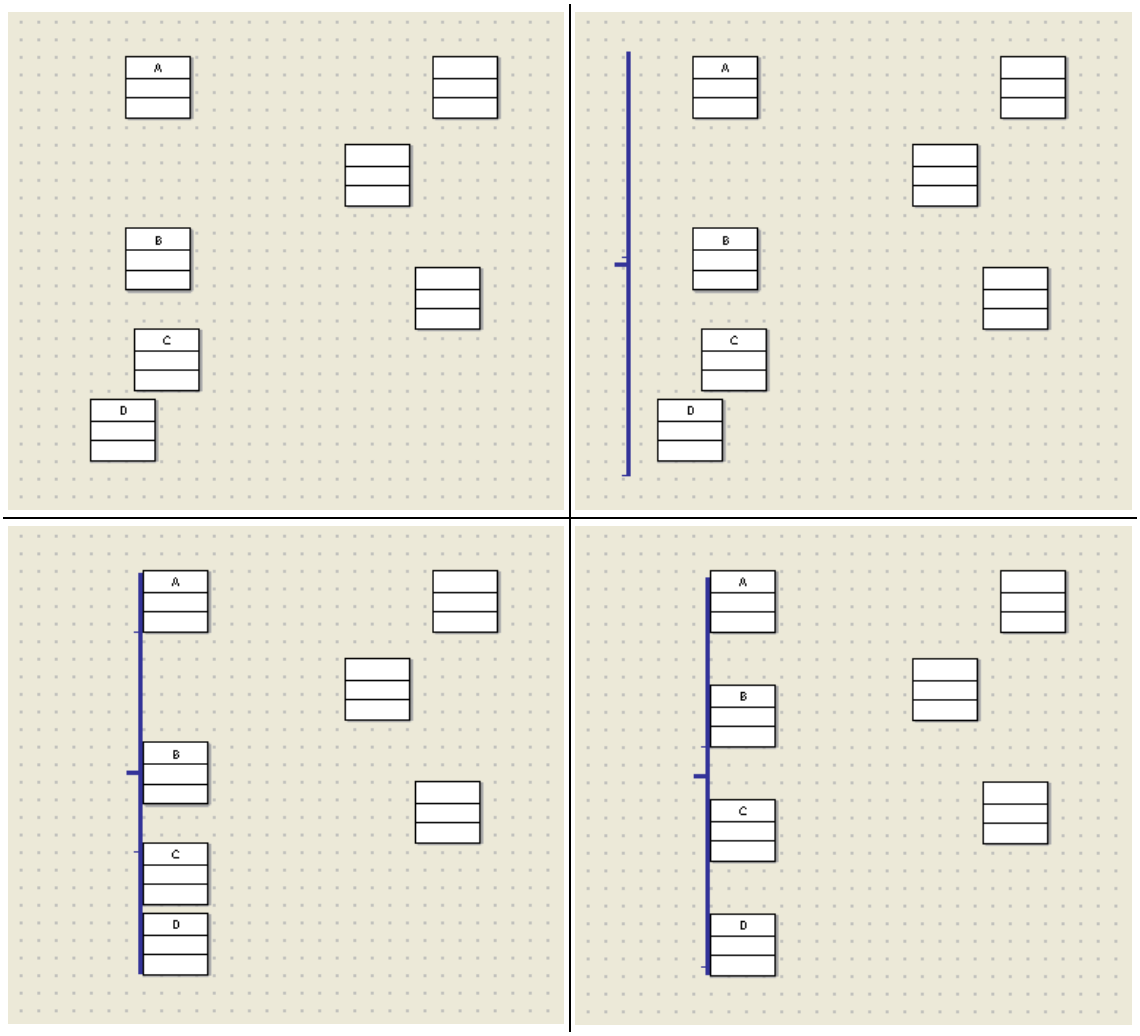
-  `Object`. Add an object to the diagram. For convenience, when the mouse is over a selected object it displays four handles to left, right, top and bottom, which may be dragged to form link relationships.
-  `Link`. Add a link between two model elements (node instance, component instance or object) selected using button 1 motion.

12.5. The Broom

ArgoUML's broom alignment tool is specialized to support the needs of designers in achieving the kind of alignment used in UML diagrams. It is common for designers to roughly align objects as they are created or by using simple movement commands. The broom is an easy way to precisely align objects that are already roughly aligned. Furthermore, the broom's distribution options are suited to the needs of UML designers: making related objects appear evenly spaced, packing objects to save diagram space, and spreading objects out to make room for new objects. The broom also makes it easy to change from horizontal to vertical alignment or from left-alignment to right-alignment.

The T-shaped icon in ArgoUML's diagram toolbar invokes the broom alignment tool. When the mouse button 1 is pressed while in broom-mode, the designer's initial mouse movement orients the broom to face in one of four directions: north, south, east, or west. After that, mouse drag events cause the broom to advance in the chosen direction, withdraw, or grow in a lateral direction. Like a real-world push broom, the broom tool pushes diagram elements that come in contact with it. This has the effect of aligning objects along the face of the broom and provides immediate visual feedback (see the figure below). Unlike a real-world broom, moving backwards allows diagram elements to return to their original position. Growing the broom makes it possible to align objects that are not near each other. When the mouse button is released, the broom disappears and the moved objects are selected to make it easy to manipulate them further.

Figure 12.5. The Broom.

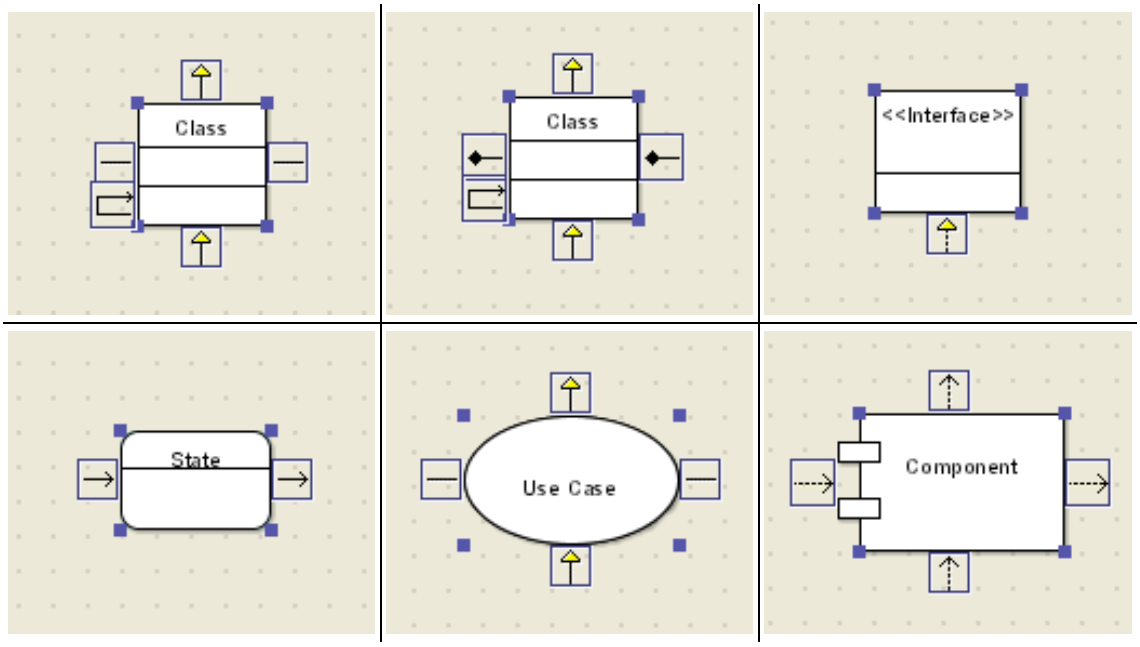


If the designer presses the space bar while using the broom, objects on the face of the broom are distributed (i.e., spaced evenly). ArgoUML's broom supports three distribution modes: objects can be spaced evenly across the space that they use, objects can be packed together with only a small gap between them, or objects can be distributed evenly over the entire length of the broom's face. Repeatedly pressing the space bar cycles among these three distribution modes and displays a brief message indicating the operation just performed: Space evenly, Pack tightly, Spread out and Original.

12.6. Selection Action Buttons

When the user selects a model element in a UML diagram, several handles are drawn on it to indicate that it is selected and to provide user interface affordances to resize the node. ArgoUML also displays some “selection-action buttons” around the selected model element. See the figure below for some examples of the handles and “selection-action buttons”. The two figures for a class differ because for creating the second one, the shift key has been depressed.

Figure 12.6. Some examples of “Selection Action Buttons”.



Selection-action buttons offer common operations on the selected object. For example, a class node has a button at 12-o'clock for adding a superclass, one at 6-o'clock for adding a subclass, and buttons at 3-o'clock and 9-o'clock for adding associations. These buttons support a "click or drag" interaction: a single click creates a new related class at a default position relative to the original class and creates a generalization or association; a drag from the button to an existing class creates only the generalization or association; and, a drag to an empty space in the diagram creates a new class at the mouse position and the generalization or association. ArgoUML provides some automated layout support so that clicking the subclass button will position the new classes so that they do not overlap.

Selection-action buttons are transparent. They have a visibly recognizable rectangular shape and size and they contain an icon that is the same as the icon used for the corresponding type of design element on the standard toolbar. However, these icons are unfilled line drawings with many transparent pixels. This allows selection-action buttons to be overlaid onto the drawing area without overly obscuring the diagram itself. Also, the buttons are only drawn when the mouse is over the selected model element; if any part of the diagram is obscured, the mouse can simply be moved away to get a clearer view of the diagram.

12.7. Clarifiers

A key feature of ArgoUML are the critics, which run in parallel with the main ArgoUML tool. When they find a problem, they typically raise a to-do item, and also highlight the problem on the editing pane. The graphical techniques used for highlighting are called *Clarifiers*

- Note icon (). Displayed at the top left of a model element indicates a critic of that model element. Moving the mouse over the icon will pop up the critic headline.
- Colored wavy line ( ). Used for critics specific to sub-components of graphical

model elements. For example to underline attributes with a problem within a class.

- Solid colored line ( ). Not seen in ordinary editing, but used when a to-do item is highlighted from the to-do pane (see Chapter 14, *The To-Do Pane*) by button 1 double click. The solid line is used to show all the model elements affected by the critic, for example all stimuli that are out of order.

12.8. The Drawing Grid

The editing pane is provided with a background grid which can be set in various styles or turned off altogether through the menu (see Section 10.5.4, “Adjust Grid”).

Whatever grid is actually displayed, placement of items on the diagram is always controlled by the setting for grid snap, which ranges from 4 to 32 pixels (see Section 10.5.5, “Adjust Snap”).

12.9. The Diagram Tab

At the bottom of the editing pane is a small tab labeled as `As Diagram`. The concept is that a UML diagram can be displayed in a number of ways, for example as a graphical diagram or as a table. Each representation would have its own tab and be selected by button 1 click on the tab.

Earlier versions of ArgoUML did implement a tabular representation, but the current release only supports a diagram representation, so this tab does not have any function.

12.10. Pop-Up Menus

Within the editing pane, button 2 click over a model element will bring up a pop-up menu with a variable number of main entries, many with a sub-menu.

12.10.1. Critiques

This sub-menu gives list of all the critics that have triggered for this model element. Selection of a menu entry causes that entry to be highlighted in the to-do pane and its detailed explanation to be placed in the `ToDoItem` tab of the details pane. A solid colored line indicates the offending element.

12.10.2. Ordering

This menu controls the ordering of overlapping model elements on the diagram. It is equivalent to the `Reorder` sub-menu of the `Arrange` menu (see Section 10.7.3, “Reorder”). There are four entries.

-  `Forward`. The selected model elements are moved one step forward in the ordering hierarchy with respect to other model elements they overlap.
-  `Backward`. The selected model elements are moved one step back in the ordering hierarchy with respect to other model elements they overlap.
-  `To Front`. The selected model elements are moved to the front of any other model elements they overlap.
-

 `To Back`. The selected model elements are moved to the back of any other model elements they overlap.

12.10.3. Add

This sub-menu only appears for model elements that can have notes attached (class, interface, object, state, pseudostate) or have operations or attributes added (class, interface). There are at most three entries.

-  `New Attribute`. Only appears where the selected model element is a class. Creates a new attribute on the model element.
-  `New Operation`. Only appears where the selected model element is a class or interface. Creates a new operation on the model element.
-  `New Comment`. Attaches a new comment to the selected model element.
- `Add All Relations`. Only appears where the selected model element is a class or interface. Makes all relations visible that exist in the model and that are connected to the selected model element.
- `Remove all Relations`. Only appears where the selected model element is a class or interface. Removes all connected relations from the diagram (without removing them from the model).

12.10.4. Show

This sub-menu only appears with certain model elements. It is completely context dependent. There are many possible entries, depending on the selected model element and its state.

- `Hide Extension Point Compartment`. Only appears when the extension point compartment of a use case is displayed. Hides the compartment.
- `Show Extension Point Compartment`. Only appears when the extension point compartment of a use case is hidden. Displays the compartment.
- `Hide All Compartments`. Only appears when both attribute and operation compartments are displayed on a class or object. Hides both compartments.
- `Show All Compartments`. Only appears when both attribute and operation compartments are hidden on a class or object. Displays both compartments.
- `Hide Attribute Compartment`. Only appears when the attribute compartment of a class or object is displayed. Hides the compartment.
- `Show Attribute Compartment`. Only appears when the attribute compartment of a class or object is hidden. Displays the compartment.
- `Hide Operation Compartment`. Only appears when the operation compartment of a class or object is displayed. Hides the compartment.
- `Show Operation Compartment`. Only appears when the operation compartment of a class or

object is hidden. Displays the compartment.

- `Hide Enumeration Literal Compartment`. Only appears when the enumeration literal compartment of an enumeration is displayed. Hides the compartment.
- `Show Enumeration Literal Compartment`. Only appears when the enumeration literal compartment of an enumeration is hidden. Displays the compartment.
- `Show All Edges`. Only appears on a class. Displays all associations (to shown model elements) that are not shown yet. This is the same function as the "add to Diagram" on the asociation in the explorer context menu. currently.
- `Hide All Edges`. Only appears on a class. Hides all associations. This is the same function as “Remove from Diagram” on all the associations of this class.
- `Hide Stereotype`. Only appears when the Stereotype of a package is displayed. Hides the stereotype.
- `Show Stereotype`. Only appears when the Stereotype of a package is hidden. Displays the stereotype.
- `Hide Visibility`. Only appears when the visibility of a package is displayed. Hides the visibility.
- `Show Visibility`. Only appears when the visibility of a package is hidden. Displays the visibility.

12.10.5. Modifiers

This sub-menu only appears with class, interface, package and use case model elements. It is used to set or clear the values of the various modifiers available.

- `Abstract`. Set for an abstract model element.
- `Leaf`. Set for a final model element, i.e. one with no sub-model elements.
- `Root`. Set for a root model element, i.e. one with no super-model elements.
- `Active`. Set for a model element with dynamic behavior.

Note

This really ought to be set automatically for model elements with state machines or activity diagrams.

12.10.6. Multiplicity

This sub-menu only appears with association model elements, when clicking at one end of the association. It is used to control the multiplicity at the end of the association nearest the mouse click point. There are only four entries, a sub-set of the range of multiplicities that are available through the property sheet of a association end (see Section 17.6, “Association End”).

- `1`

- `0..1`
- `1..*`
- `0..*`

12.10.7. Aggregation

This sub-menu only appears with association model elements, when clicking at one end of the association. It is used to control the aggregation at the end of the association nearest the mouse click point. There are three entries.

- `none`. Remove any aggregation.
- `aggregate`. Make this end a shared aggregation (loosely known as an “aggregation”).
- `composite`. Make this end a composite aggregation (loosely known as a “composition”).

Caution

UML requires that an end with a composition relationship must have a multiplicity of 1 (the default).

12.10.8. Navigability

This sub-menu only appears with association model elements, when clicking at one end of the association. It is used to control the navigability of the association. There are three entries.

- `bidirectional`. Make the association navigable in both directions.
- `<class1> to <class2>`. Make the association navigable only from <class1> to <class2>. In other words <class1> can reference <class2> but not the other way round.
- `<class2> to <class1>`. Make the association navigable only from <class2> to <class1>. In other words <class2> can reference <class1> but not the other way round.

Note

UML does permit an association to be non-navigable in both directions. ArgoUML will allow this, but you will have to set each of the association ends navigation property, reached from the property tab of the association - and the diagram does not show any arrows in this case.

This is considered bad design practice (it will trigger a critic in ArgoUML), so is only of theoretical interest.

Note

UML does not permit navigability from an interface to a class. ArgoUML does not prevent this.

12.11. Notation

Notation is the textual representation on the diagram of a modelelement or its properties.

12.11.1. Notation Languages

ArgoUML supports showing notation in different languages. By default, all text is shown in UML notation, but the menus contain an item to select between Java and UML. With plugin modules, it is even possible to select other languages, such as C++.

Figure 12.7, “A class in UML notation” shows a class in UML notation, while Figure 12.8, “A class in Java notation” shows the same class in Java notation.

Figure 12.7. A class in UML notation

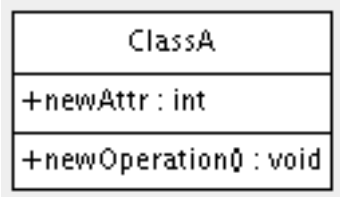
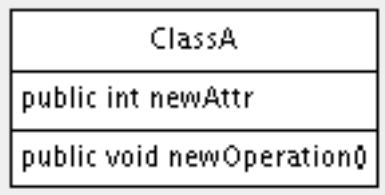


Figure 12.8. A class in Java notation



12.11.2. Notation Editing on the diagram

Most text shown on a diagram may be edited by double-clicking button 1 on the text. This causes a edit box to be shown, with the previous text selected, ready for amending.

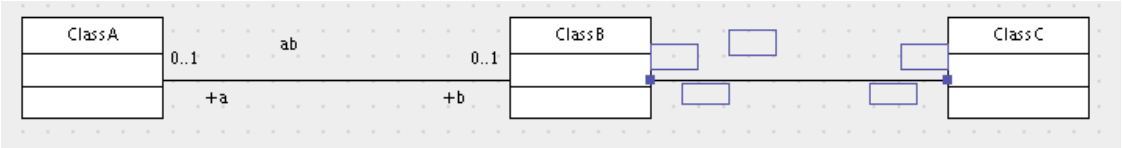
Also, the status bar of ArgoUML (i.e. the small area at the bottom of the ArgoUML window), shows an help text that indicates the syntax of the text to be entered. Text entry can be concluded by pressing F2, or for single-line fields, by pressing the enter key. Additionally, editing can be concluded by clicking somewhere in the diagram outside the edit area.

Editing notation on the diagram is a very powerful way to enter a lot of model-information in a very compact way. It is e.g. possible to create an operation, its stereotype, all parameters and their types, and operation properties (visibility, concurrency), all at once by typing:

```
+Order(customerID : int,items : List) : void {sequential}
```

An association (e.g. between two classes) is showing many texts close to its middle and ends, so it deserves some extra explanation. Figure 12.9, “A couple of associations with adornments” shows two associations to clarify the following:

Figure 12.9. A couple of associations with adornments



The association on the right shows that invisible fields where text can be entered become visible once the modelelement is selected. The fields are indicated by blue rectangles - double-click on them with mouse button 1 to start editing.

The visibility (the +, -, # or ~) is shown together with the association-end name, but it is not shown for an unnamed association end.

Likewise, the multiplicity is not shown if it is 1, unless the setting `Show "1" multiplicities` in the menu `File=>Properties` is checked.

The example figure does not demonstrate this, but stereotypes of an association are shown on the diagram, but are not editable. And stereotypes of association-ends are shown together with the association-end name.

12.11.3. Notation Parsing

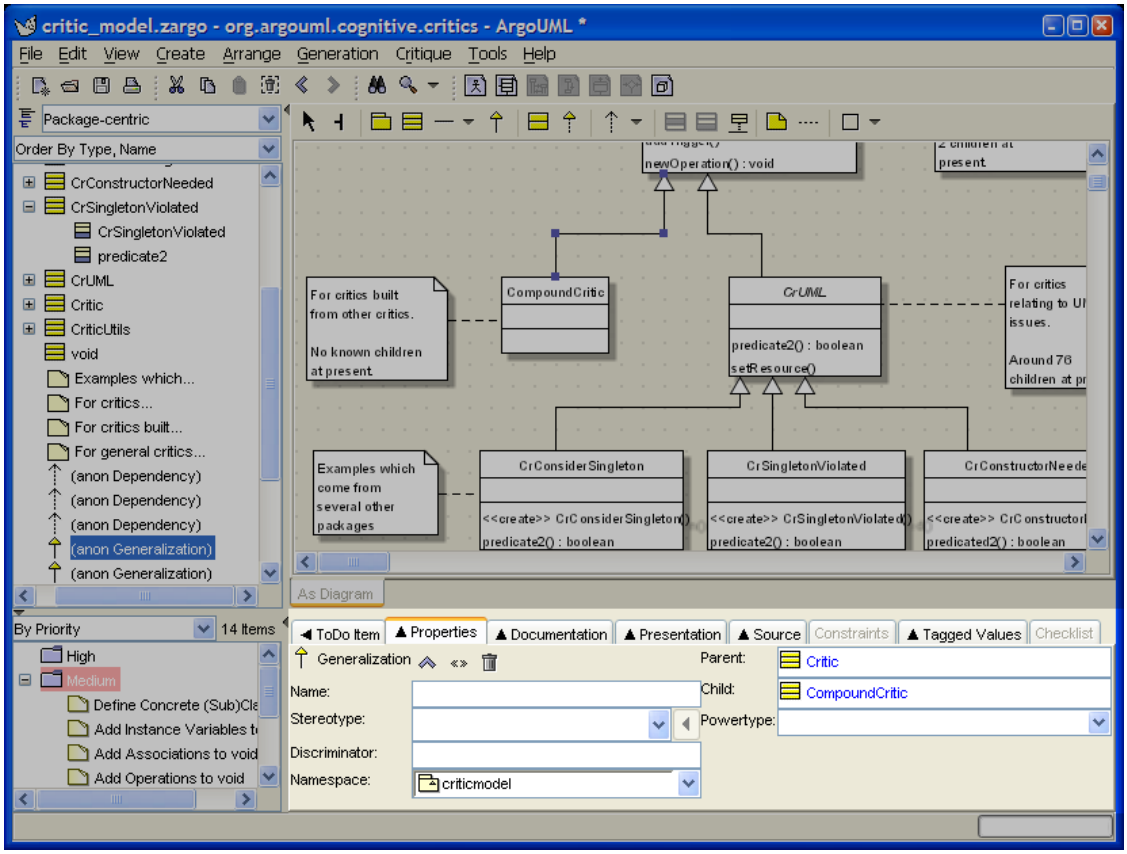
(to be written)

Chapter 13. The Details Pane

13.1. Introduction

Figure 13.1, “Overview of the details pane” shows the ArgoUML window, with the details pane highlighted.

Figure 13.1. Overview of the details pane



For any model element within the system, this pane is where all its associated data is viewed and entered.

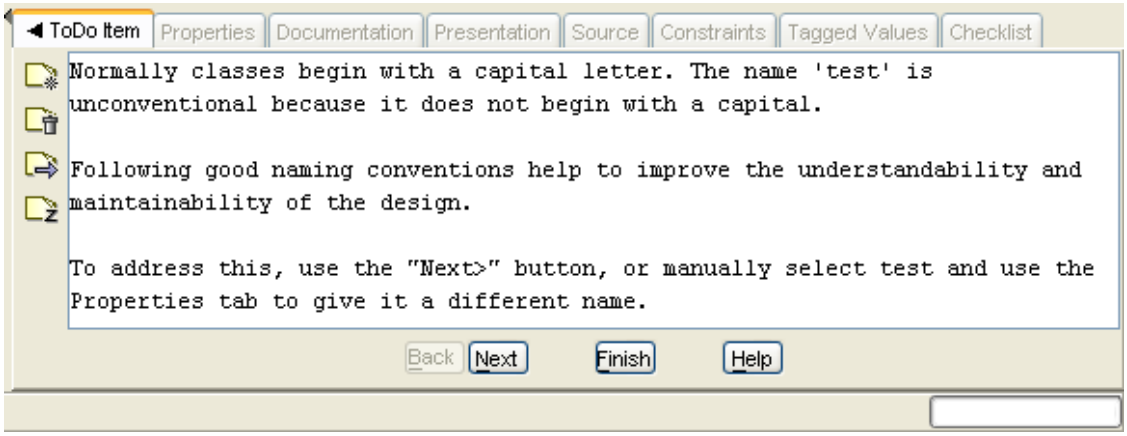
The Pane has a series of tabs at the top, which are selected by button 1 click The body of a tab is a menu of items to be checked, selected or entered specific to the particular tab selected.

Of these, the `Properties` Tab is by far the most complex, with a different presentation for each model element within the system. The detailed descriptions of the properties tab for each model element are the subject of separate chapters covering the model elements that may appear on the various diagrams (see Chapter 16, *Top Level Model Element Reference* through Chapter 23, *Deployment Diagram Model Element Reference*).

13.2. To Do Item Tab

This tab provides control over the various to-do items created by the user, or raised automatically by the ArgoUML critics (discussed in more detail in the section on the `Critique` menu—see Section 10.9, “The Critique Menu”). Figure 13.2, “Example of the `To Do Item` tab on the properties pane” shows a typical pane. The to-do item is selected with button 1 in the to-do pane (see Chapter 14, *The To-Do Pane*) or by using the `Critiques` context sensitive pop-up menu on the editing pane.

Figure 13.2. Example of the `To Do Item` tab on the properties pane

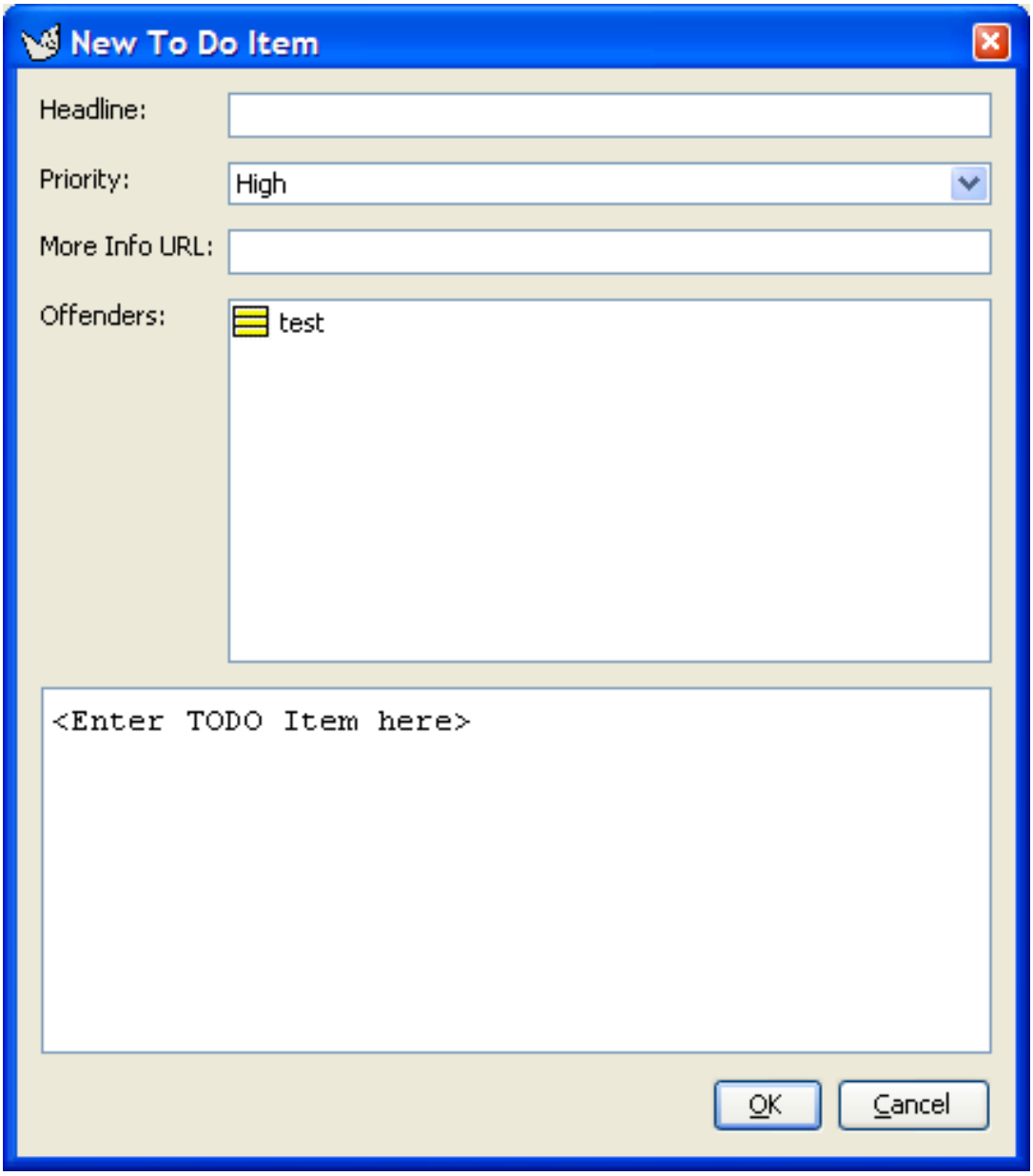


Customization of the critics behaviour is possible through the `Browse critics...` menu (see Section 10.9.4, “Browse Critics...”).

The body of the tab describes the problem found by the critic and outlines how it can be fixed. To the left are four buttons.

- `New To Do Item...` This launches a dialog box (see Figure 13.3, “Dialog box for `New To Do Item` ”), which allows you to create your own to-do item, with its own headline (which appears in the to-do pane), priority for the to-do pane, reference URL and detailed description for further information.

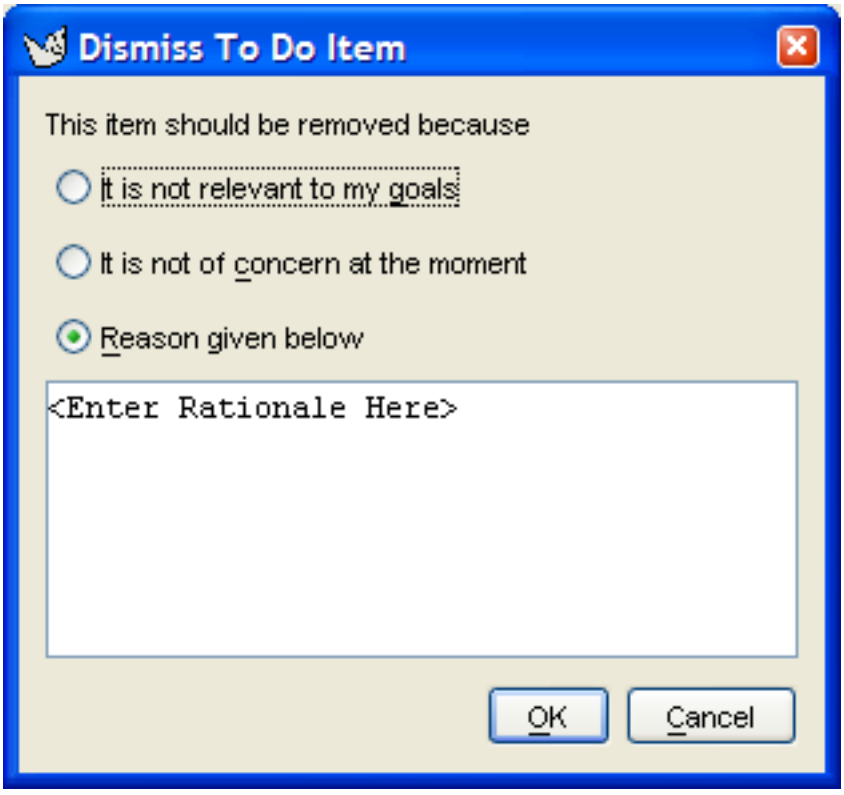
Figure 13.3. Dialog box for `New To Do Item`



- `Resolve Item...` This pops up a dialog allowing the user to resolve the selected to-do item (see Figure 13.4, “Dialog box for `Resolve Item` ”). This is an important dialog, because it allows you to deal with to-do items in ways other than the recommendation of the to-do item (which is the whole point of their being advisory).

  This dialog box is intended to be used for the following reasons: deleting todo items that were manually created, preventing a single critic to trigger on a single object, and dismissing categories of todo items by lowering design concerns or design goals.

Figure 13.4. Dialog box for `Resolve Item`



At the top are three radio-buttons, of which by default the last is selected, labeled 1) `It is not relevant to my goals`, 2) `It is not of concern at the moment`, and 3) `Reason given below`. If you choose the third of these you should enter a reason in the main text box.

Tip

If you wish to resolve a to-do item (that is generated by a critic) by following its recommendations, just make the recommended changes and the to-do item will disappear of its own accord. There is no need to use this dialog.

Warning

The V0.20 version of ArgoUML implementation is incomplete: The reason given is not stored when the project is saved. And there is no way to retrieve todo items that were resolved. So, it is not usefull to give a reason at all.

When a todo item generated by a critic is resolved, then there is no way to undo this (unless by re-creating the object that triggered the critic).

-  `Snooze Critic` This suspends the activity of the critic that generated the current to-do item. The to-do item (and all others generated by the critic) will disappear from the to-do pane.

  The critic will wake up after a period of time. Initially this period is 10 minutes, but it doubles on each successive application of the Snooze button. The critic can be awakened explicitly through the `Critique > Browse Critics...` menu (see Section 10.9.4, “Browse Critics...”).

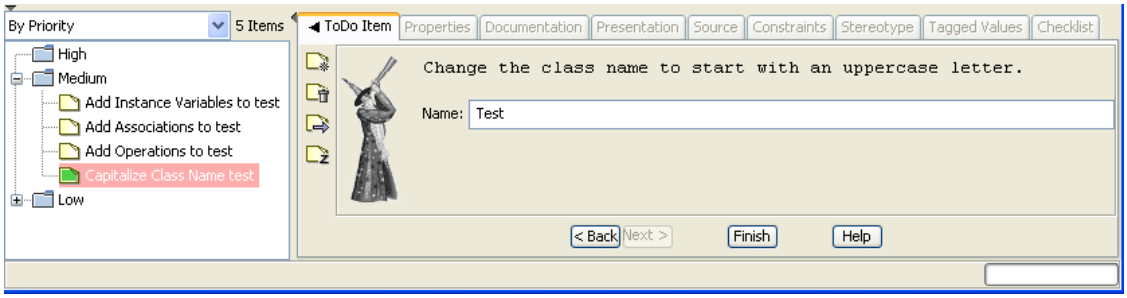
Tip

Some common critics can fire the whole time as you build a big diagram. Some users find it useful to snooze these critics until the diagram has been completed.

13.2.1. Wizards

Some of the more common critics have a “wizard” available to help in fixing the problem. The wizard comprises a series of pages (one or more) in the `ToDo Item` tab that step you through the changes. Start the wizard by clicking the `Next>` button.

Figure 13.5. Example of a `Wizard`



The wizard is driven through the first three buttons at the bottom of the `ToDo Item` tab.

- `<Back`. This will take you back to the previous step in the wizard. Grayed out if this is the first step.
- `Next>`. This will take you back to the next step in the wizard. Grayed out if this is the last step.
- `Finish`. This will commit the changes you have made through the wizard in previous steps, and/or use the defaults for all next steps.

Note

Not all to-do items have wizards. If there is no wizard all three buttons will remain grayed out.

The ArgoUML wizards are *non-modal*, i.e. once started, you may select other todo items, or do some other actions, and all the while the wizard will remeber where it was, so if you return to the todo item, the wizard will indicate the same step it was on when you left it.

13.2.2. The Help Button

There is one remaining button at the bottom of the `To Do Item` tab, labeled `Help`. This will fire up a browser to a URL with further help.

13.3. Properties Tab

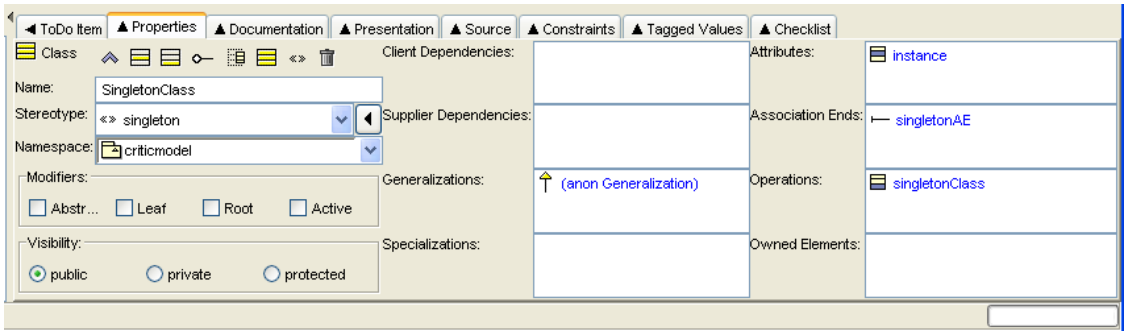
Through this tab, the properties of model elements selected in the explorer or editing pane may be set. The properties of an model element may be displayed in one of the following ways:

1. Selection of the model element in the explorer or editing panes, followed by selection of the properties tab in the details pane; or

2. Navigation buttons cause different model elements to be selected. I.e. the `Go Up` button on the properties tab, the `Navigate Back` and `Navigate Forward` buttons in the main tool bar, and the various menu-items under `Edit - Select`.

Figure 13.6, "A typical `Properties` tab on the details pane" shows a typical properties tab for a model element in ArgoUML (in this case a class).

Figure 13.6. A typical `Properties` tab on the details pane



At the top left is the icon and name of the type of model element (i.e. the UML metaclass, not the actual name of this particular model element). In this example the property tab is for a class.

To the right of this is a toolbar of icons relevant to this property tab. The first one is always navigation `Go up`. The last is always `Delete` to delete the selected model element from the model. The ones in between depend on the model element.

The remainder of the tab comprises fields, laid out in two or three columns. Each field has a label to its left. The fields may be text boxes, text areas, drop down selectors, radio boxes and check boxes. In most (but not all cases) the values can be changed. In the case of text boxes this is sometimes by just typing the required value.

However for many text boxes and text areas, data entry is via a context sensitive pop-up menu (using button 2 click), which offers options to add a new entry, delete an entry or move entries up and down (in text areas with multiple entries).

The first field is almost always a text field `Name`, where the name of the specific model element can be entered. The remaining fields vary depending on the model element selected.

The detailed property sheets for all ArgoUML model elements are discussed in separate chapters for each of the diagram types (use case diagram (Chapter 17, *Use Case Diagram Model Element Reference*, class diagram (Chapter 18, *Class Diagram Model Element Reference*, sequence diagram (Chapter 19, *Sequence Diagram Model Element Reference*, statechart diagram (Chapter 20, *Statechart Diagram Model Element Reference*, collaboration diagram (Chapter 21, *Collaboration Diagram Model Element Reference*, activity diagram (Chapter 22, *Activity Diagram Model Element Reference*, deployment diagram (Chapter 23, *Deployment Diagram Model Element Reference*). Property sheets for model elements that are common to all diagram types have their own chapter (Chapter 16, *Top Level Model Ele-*

ment Reference).

Caution

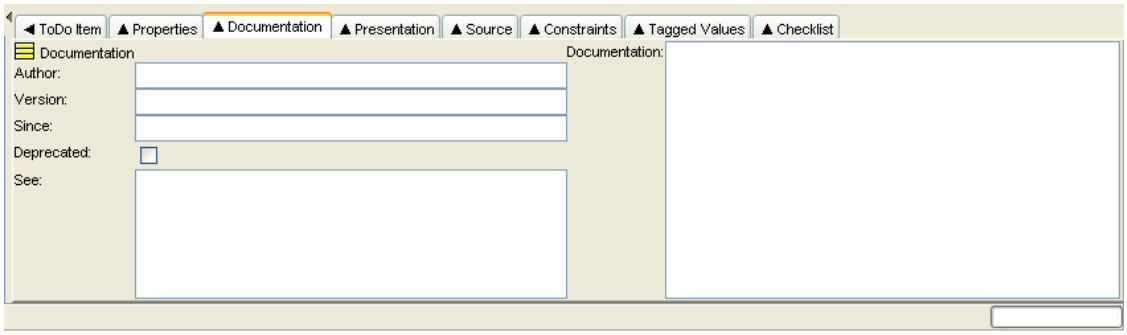
ArgoUML will always try to squeeze all fields on to the property sheet. If the size of the property tab is too small, it may become unusable. The solution is to either enlarge the property tab by enlarging the main window, or by moving the dividers to left and top.

13.4. Documentation Tab

Within the UML 1.4 standard, all model elements are children of the `Element` metaclass. The `Element` metaclass defines a tagged value `documentation` for comment, description or explanation of the element to which it is attached. Since this tagged value applies to every model element, it is given its own tab in the details pane, rather than being part of the `Tagged Values` tab.

Figure 13.7, “A typical `Documentation` tab on the details pane” shows a typical documentation tab for a model element in ArgoUML.

Figure 13.7. A typical `Documentation` tab on the details pane



As you can see, many more fields have been added to the Documentation field alone. The other fields similarly store their information under tagged values: `author`, `version`, `since`, `deprecated`, `see`.

The fields on this tab are the same for all model elements.

Since UML comments are a kind of documentation, they are also shown on this tab, with name and body.

- `Author:` A text box for the author of the documentation.
- `Version:` A text box for the version of the documentation.
- `Since:` A text box to show how long the documentation has been valid.
- `Deprecated:` A check box to indicate whether this model element is deprecated (i.e. planned for removal in future versions of the design model).
- `See:` Pointers to documentation outside the system.
- `Documentation:` Literal text of any documentation.

- `Comment Name:` The names of all comments attached to the modelelement.
- `Body:` The bodies of all comments attached to this modelelement.

Tip

ArgoUML is not primarily a documentation system. For model elements that require heavy documentation, notably use cases, the use of the `See:` field to point to external documents is more practical.

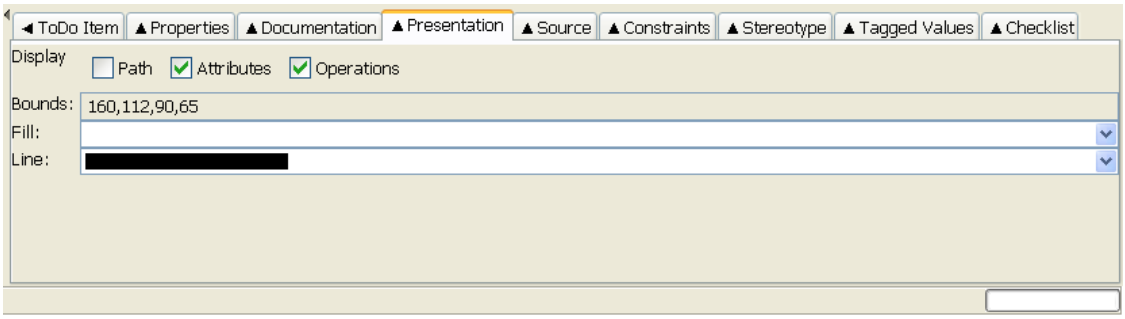
13.5. Presentation Tab

This tab provides some limited control over the graphical representation of model elements in the diagram in the editing pane.

Model elements that do not have any specific direct graphical representation on the screen (beyond their textual description) do not have style tabs of their own. For example the style sheet of an operation on a class will be downlighted.

Style sheets vary a little from model element to model element, but Figure 13.8, “A typical `Presentation` tab on the details pane” shows a typical style tab for a model element in ArgoUML (in this case a class).

Figure 13.8. A typical `Presentation` tab on the details pane



There may be further fields in some cases, e.g. for a package, but most fields are common to many model elements.

- `Path` This checkbox allow to display or hide the path in front of the name of the modelelement. It is shown in UML notation with `::` seperators. E.g. the ArgoUML Main class would be shown as: `org::argouml::application::Main`.
- `Attributes` This checkbox allows to hide or show the attributes compartment of a class.
- `Operation` This checkbox allows to hide or show the operations compartment of a class or interface.
- `Stereotype` This checkbox allows to reveal or hide the stereotypes of a package, shown above the name.
- `Visibility` This checkbox allows to hide the visibility of a package. The visibility is shown in

UML notation as +, -, # or ~.

- `Extension Points` This checkbox allows to reveal or hide the extensions points compartment of a usecase.

- `Bounds:` This defines the corners of the bounding box for a 2D model element. It comprises four numbers separated by commas. These four numbers are respectively: i) the X coordinate of the upper left corner of the box; ii) the Y coordinate of the upper left corner of the box; iii) the width of the box; and iv) the height of the box. All units are pixels on the editing pane.

  This field has no effect on 1D model elements that link other model elements (associations, generalizations etc), since their position is constrained by their connectedness. In this case the field is downlighted.

- `Fill:` This drop-down selector specifies the fill color for 2D model elements. It is not present for line model elements. Selecting `No Fill` makes the model element transparant. Selecting `Custom` allows to create other colors then the ones listed. It causes the color selector dialog box to appear, see Figure 13.9, “The `Custom Fill/Line Color` dialog box”.

- `Line:` This drop-down selector specifies the line color for model elements. Selecting `No Fill` makes the model element transparant. Selecting `Custom` allows to create other colors then the ones listed. It causes the color selector dialog box to appear, see Figure 13.9, “The `Custom Fill/Line Color` dialog box”.

Figure 13.9. The `Custom Fill/Line Color` dialog box

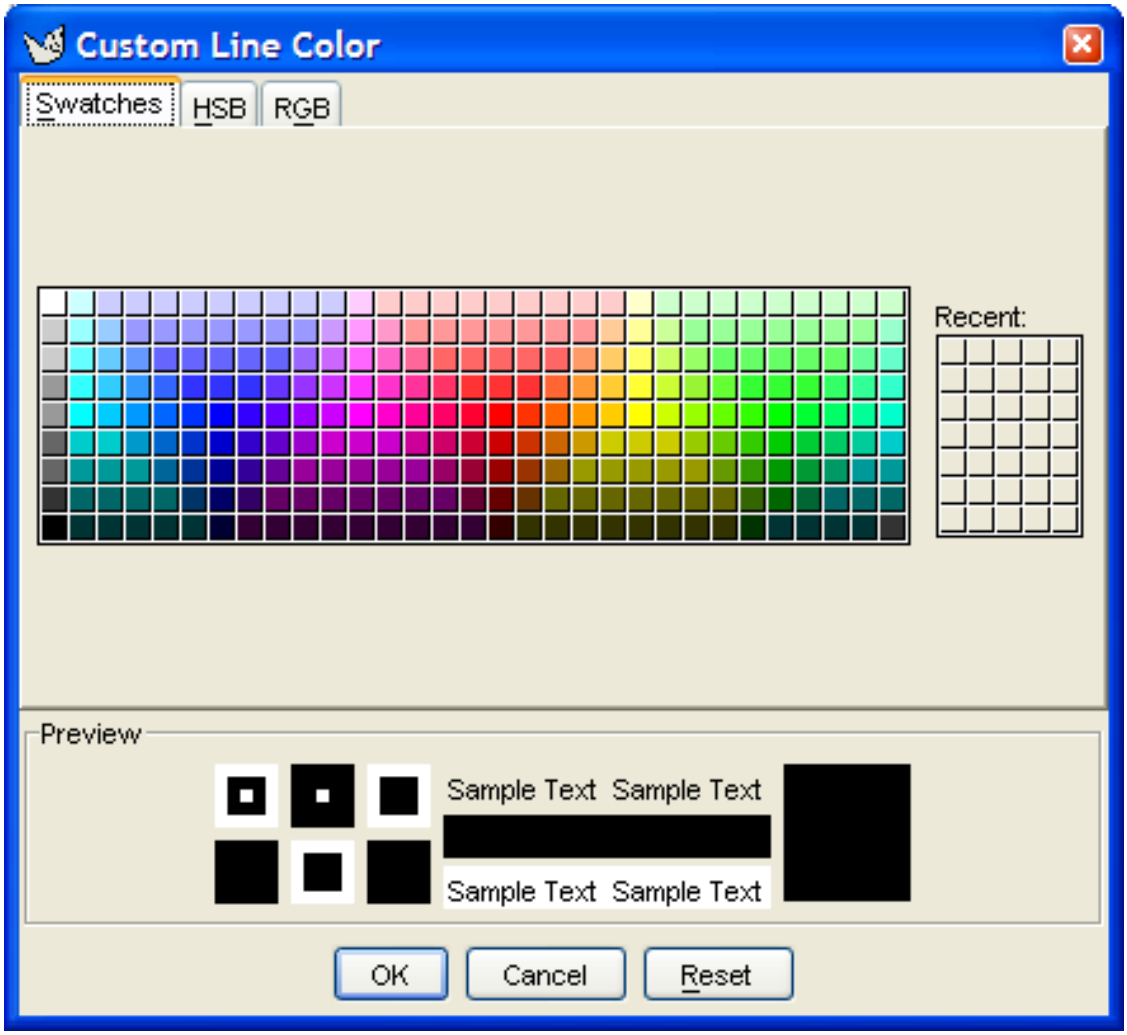


Figure 13.10. The `Custom Fill/Line Color` dialog box

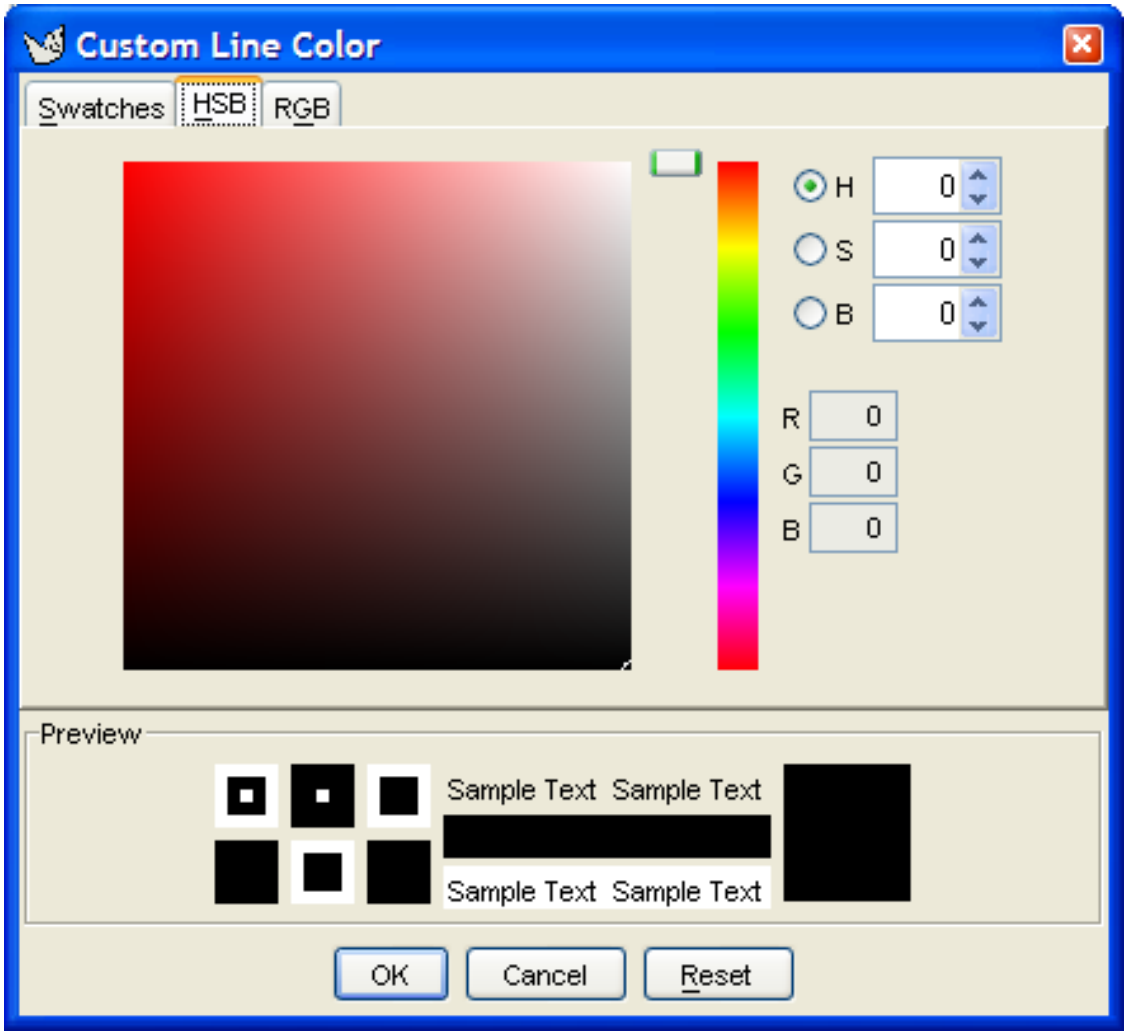
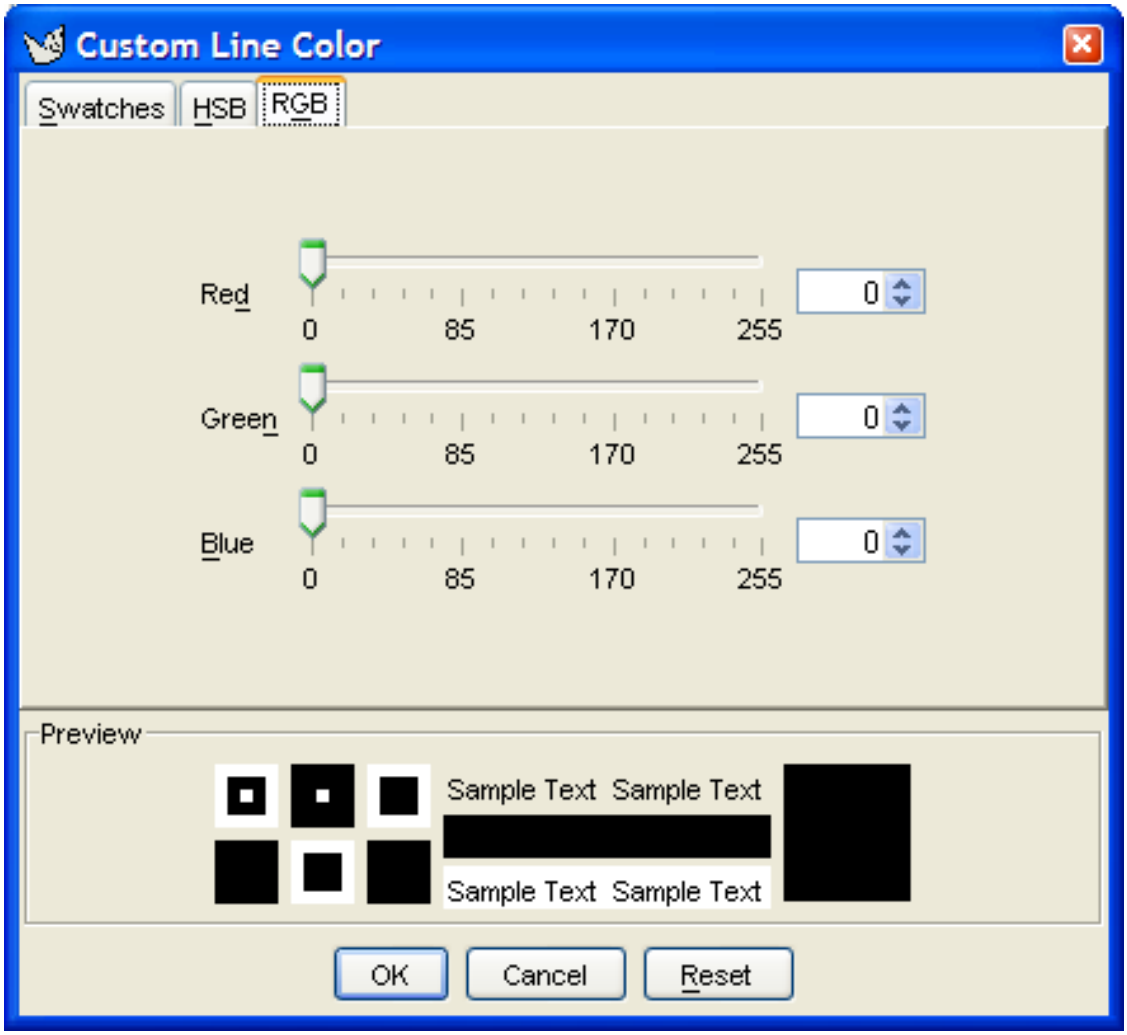


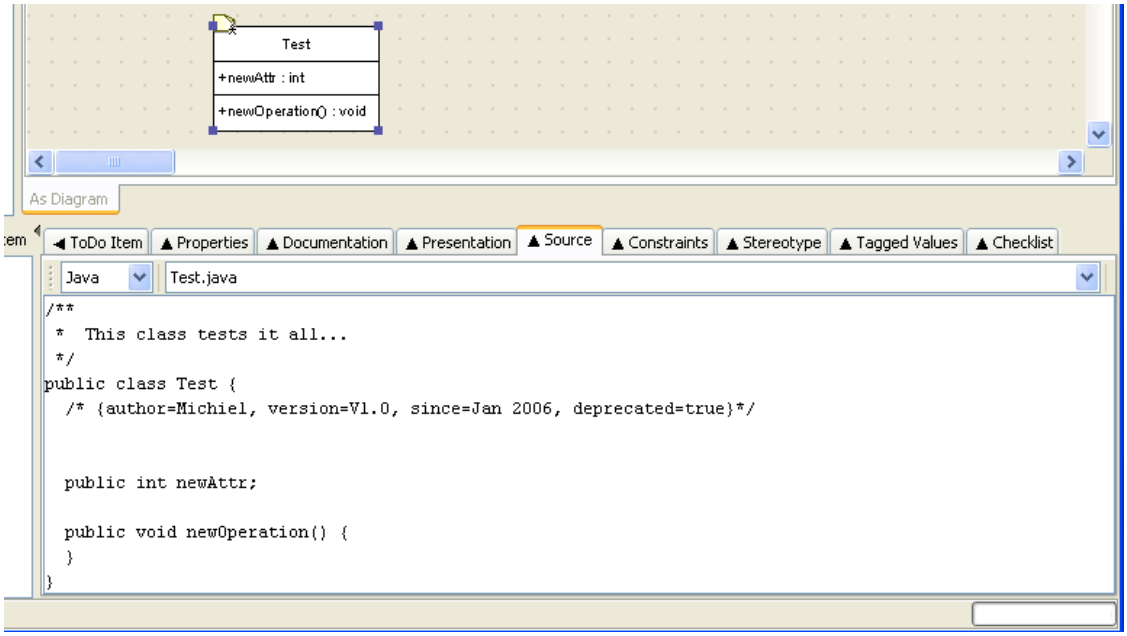
Figure 13.11. The `Custom Fill/Line Color` dialog box



13.6. Source Tab

This tab shows the source code that will be generated for this model element, in the selected language. ArgoUML generates the code e.g. for classes and interfaces. The code shown here, may be saved in the indicated files with the aid of the functions in the Generation menu.

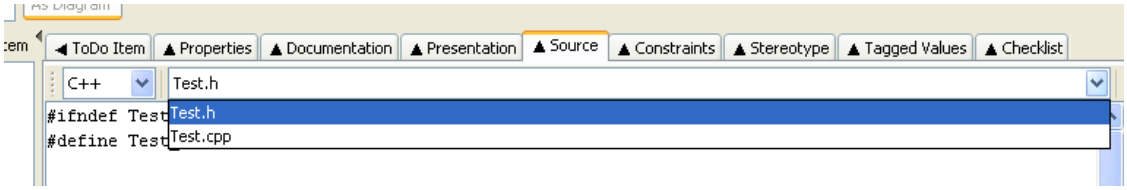
Figure 13.12. The Source Tab of a class.



Any code you add will be lost - that is not the intention of ArgoUML - use an IDE instead.

The dropdown at the right allows selection of the output file. This function is not very useful for languages that generate all code for a class within one file, but serves its purpose for e.g. C++, where a .h and .cpp file are generated. See the figure below.

Figure 13.13. A C++ example.



13.7. Constraints Tab

Constraints are one of the extension mechanisms provided for UML. ArgoUML is equipped with a powerful constraint editor based on the *Object Constraint Language (OCL)* defined in the UML 1.4 standard.

Caution

The OCL editor implementation for ArgoUML V0.24 doesn't support OCL constraints for elements other than Classes and Features.

This is something of a general restriction of OCL. Although the UML specification claims that there may be a constraint for every model element, the OCL specification only defines classes/interfaces and operations as allowable contexts.

It is not before OCL 2.0 that a more general definition of allowable contexts is introduced. The key issue is that for each context definition you need to define what is the contextual-

> Classifier, i.e., the classifier that will be associated with the self keyword. The creators of the OCL specification claim that this is not an issue for the OCL specification, but rather for UML or some integration task force. Conversely, it seems that the UML specification people seem to expect this to be defined in the OCL specification (which is why we did a first step in that direction in OCL 2.0).
>
> So, to cut a long story short, it appeared that the simplest solution for ArgoUML at the moment would be to enable the OCL property panel only for those model elements for which there actually exists a definition of the contextualClassifier in OCL 1.4. These are (s. above) Class/Interface and Feature.

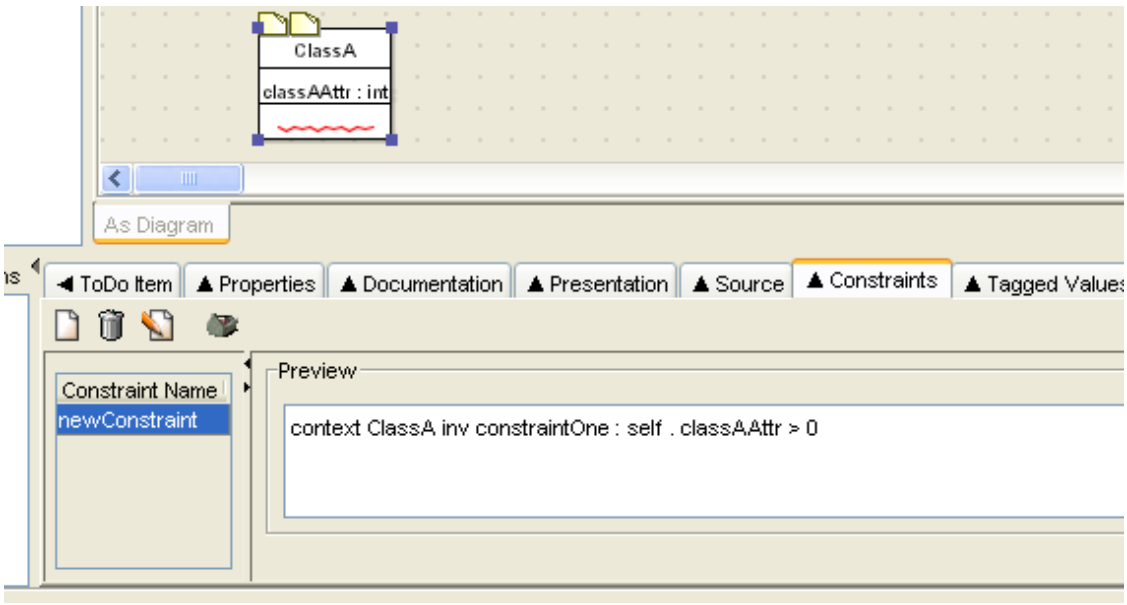
The standard pre-defines a small number of constraints (for example the `xor` constraint over a set of associations indicating that only one may be manifest for any particular instance).

The standard also envisages a number of circumstances where general purpose constraints may be useful:

- To specify invariants on classes and types in the class model;
- To specify type invariants for stereotypes;
- To describe pre-conditions and post-conditions on operations and methods;
- To describe guards;
- As a navigation language; and
- To specify constraints on operations.

Figure 13.14, “A typical `Constraints` tab on the details pane” shows a typical constraint tab for a model element in ArgoUML (in this case a class).

Figure 13.14. A typical `Constraints` tab on the details pane



Along the top of the tab are a series of icons.

-  `New Constraint`. This creates a new constraint and launches the constraint editor in the `Constraints tab` for that new constraint (see Section 13.7.1, “The Constraint Editor”). The new constraint is created with a context declaration for the currently selected model element.

  ## Warning

  It seems logical, that when a new constraint is created, it needs to be edited. But ArgoUML V0.24 fails to start the OCL editor upon creation; you have to do this by primo selecting the new constraint first, secundo rename it, and tertio press the `Edit Constraint` button. It is essental for successfully creating a constraint to follow these 4 steps accurately: create, select, rename, edit. The step to rename is necessary, because the validity check will refuse the constraint if its name differs from the name mentioned in the constraint text. For the same reason, renaming a constraint afterwards is impossible.

-  `Delete Constraint`. The constraint currently selected in the `Constraint Name` box (see below) is deleted.

  ## Caution

  In V0.24 of ArgoUML this button is not downlighted when it is not functional, i.e. when no constraint is selected.

-  `Edit Constraint`. This launches the constraint editor in the `Constraints` tab (see Section 13.7.1, “The Constraint Editor”). The editor is invoked on the constraint currently selected in the `Constraint Name` box.

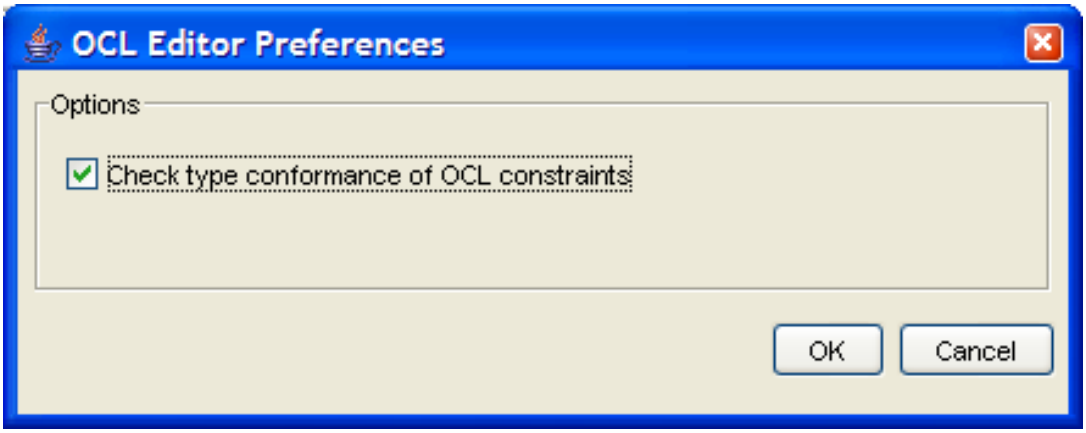
  

  ## Caution

  In V0.24 of ArgoUML this button is not downlighted when it is not functional, i.e. when no constraint is selected.

-  `Configure Constraint Editor`. This a dialog to configure options in the constraint editor (see Figure 13.15, “Dialog box for configuring constraints”).

Figure 13.15. Dialog box for configuring constraints



The dialog box has a check box for the following option.

- `Check type conformance of OCL constraints`. OCL is strictly typed. At the early stages of design it may be helpful to disable type checking, rather than follow through all the detailed specification needed to get type consistency.

At the bottom are two buttons, labeled `OK` (to accept the option changes) and `Cancel` (to discard the changes).

The main body of the constraints tab comprises two boxes, a smaller to the left and a larger one to the right. The two are separated by two small arrow buttons which control the size of the boxes.

- `Shrink Left`. Button 1 click on this icon shrinks the box on the left. Its effect may be reversed by use of the `Shrink Right` button (see below).
- `Shrink Right`. Button 1 click on this icon shrinks the box on the right. Its effect may be reversed by use of the `Shrink Left` button (see above).

Finer control can be achieved by using button 1 motion to drag the dividing bar to left and right.

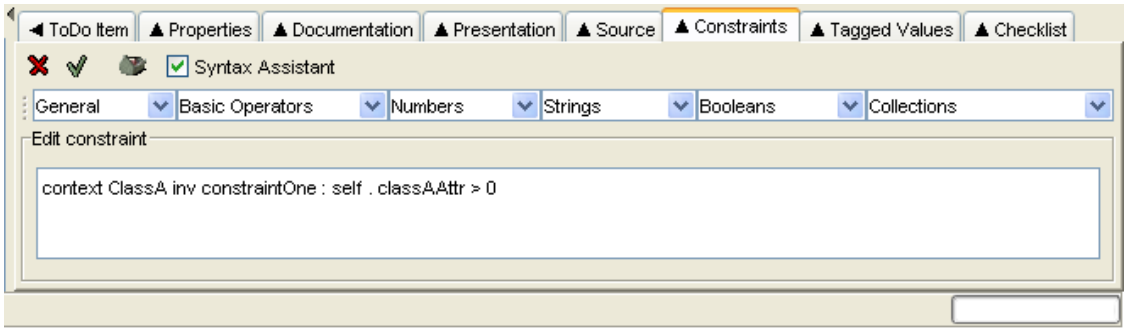
The box on the left is titled `Constraint Name` and lists all the constraints (if any) so far defined for the selected model element. A constraint may be selected by button 1 click.

The box on the right is labeled `Preview` and contains the text of the constraint. This box only shows some contents if a constraint is selected. Where a constraint is too large for the box, a scroll bar is provided to the right.

13.7.1. The Constraint Editor

This is invoked through the use of the `Edit Constraint` button on the main `Constraints` tab. The constraint editor takes up the whole tab (see Figure 13.16, “Dialog box for configuring constraints”).

Figure 13.16. Dialog box for configuring constraints



Along the top of the tab are a series of icons.

-  `Cancel Edit Constraint`. This exits the constraint editor without saving any changes and returns to the main `Constraints` tab.
-  `Check OCL Syntax`. This button invokes a full syntax check of the OCL written in the editor. If the syntax is valid, the constraint is saved, and control returns to the main `Constraints` tab. If the syntax is not valid, a dialog box explains the problem.

  ## Warning

  Whether type checking is included should be configurable with the `Configure Constraint Editor` button (see below). But ArgoUML V0.20 does always check, and refuses to accept any constraint with the slightest error.

-  `Configure Constraint Editor`. This a dialog to configure options in the constraint editor. It is also available in the main `Constraints` tab and is discussed in detail there (see Section 13.7, “Constraints Tab”).

To the right of the toolbar is a check box labeled `Syntax Assistant` (unchecked by default), which will enable the syntax assistant in the constraint editor.

If the syntax assistant is enabled, six drop down menus are provided in a row immediately below the toolbar. These provide standard templates for OCL that, when selected, will be inserted into the constraint being edited.

The syntax assistant can be made floating in a separate window by button 1 motion on the small divider area to the left of the row of drop-down menus.

- `General`. General OCL constructors. Entries: `inv` (inserts an invariant); `pre` (inserts a pre-condition); `post` (inserts a post-condition); `self` (inserts a self-reference); `@pre` (inserts a reference to a value at the start of an operation); and `result` (inserts a reference to a previous result).
- `Basic Operators`. Relational operators and parentheses. Entries: `=`; `<>`; `<`; `>`; `<=`; `>=`; and `()`.
- `Numbers`. Arithmetic operators and functions. Entries: `+`; `-`; `*`; `/`; `mod`; `div`; `abs`; `max`; `min`; `round`; and `floor`.
- `Strings`. String functions. Entries: `concat`; `size`; `toLower`; `toUpper`; and `substring`.

- `Booleans`. Logical functions. Entries: `or`; `and`; `xor`; `not`; `implies`; and `if then else`.
- `Collections`. Operators and functions on collections—bags, sets and sequences. The large number of functions are organized into sub-groups.
  - `General`. Functions that apply to all types of collection. Entries: `Collection {}` (insert a new collection); `Set {}` (insert a a new set); `Bag {}` (insert a new bag); `Sequence {}` (insert a new sequence); `size`; `count`; `isEmpty`; `notEmpty`; `includes`; `includesAll`; `iterate`; `exists`; `forAll`; `collect`; `select`; `reject`; `union`; `intersection`; `including`; `excluding`; and `sum`.
  - `Sets`. Operators and functions that apply only to sets. Entries: `-` (set difference); and `symmetricDifference`.
  - `Sequences`. Functions that apply to sequences. Entries: `first`; `last`; `at`; `append`; `prepend`; and `subSequence`.

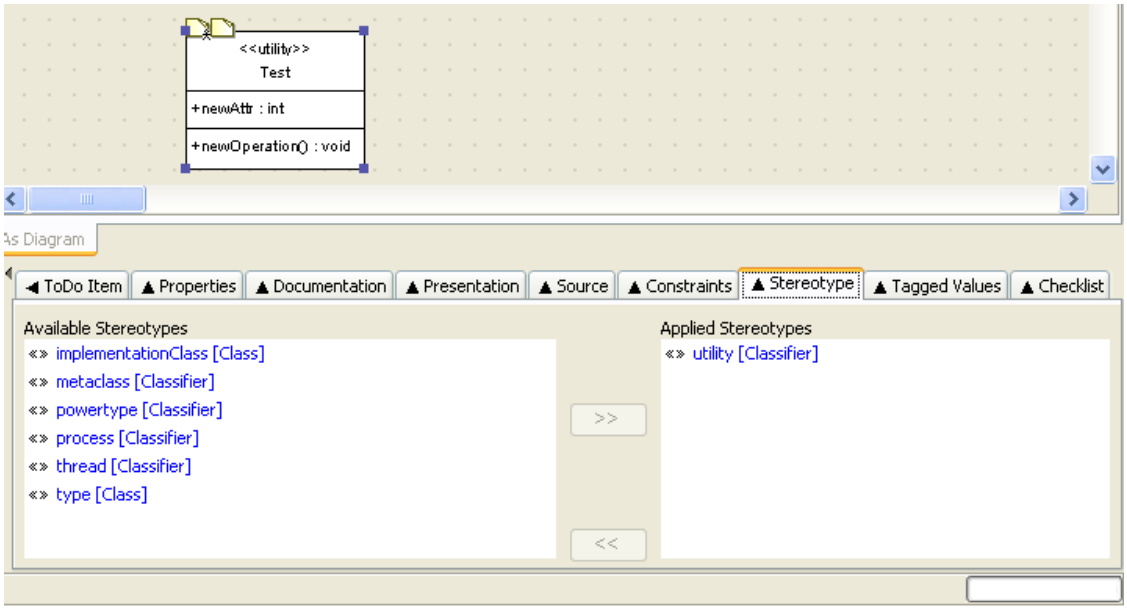
The remainder of the tab comprises a writable text area containing the text to be edited. The mouse buttons have their standard behavior within an editable text area (see Section 8.2, “General Mouse Behavior in ArgoUML”).

In addition, cut, copy and paste operations may be invoked through the keyboard shortcuts `Ctrl-X`, `Ctrl-C` and `Ctrl-V` respectively.

13.8. Stereotype Tab

This tab shows the available and applied stereotypes for the currently selected modelelement. It consists of 2 panels and 2 buttons. The buttons allow to move the stereotypes from one list to the other.

Figure 13.17. An example of a stereotype tab for a class.



In the lists, between `[]` the baseclass of the stereotypes is shown. E.g. in the figure above, the `thread [Classifier]` stereotype may be applied to all types of classifiers, such as Class, UseCase,...

13.9. Tagged Values Tab

Tagged values are another extension mechanism provided by UML. The user can define name-value pairs to be associated with model elements which define properties of that model element. The names are known as *tags*. UML pre-defines a number of tags that are useful for many of its model elements.

Note

The tag `documentation` is defined for the top UML metaclass, `Element` and is so available to all model elements. In ArgoUML documentation values are provided through the `Documentation` tab, rather than by using the `Tagged Values` tab.

The `Tagged Values` tab in ArgoUML comprises a two column table, with a combo-box on the left to select the tagdefinition and an editable box on the right for the associated value. There is always at least one empty row available for any new tag.

The button at the top of this tab allows creation of a new tagdefinition. After clicking this button, go to the properties tab first to set the name of the new tagdefinition.

The mouse buttons have their standard behavior within the editable value area (see Section 8.2, “General Mouse Behavior in ArgoUML”). In addition, when in the value field, cut, copy and paste operations may be invoked through the keyboard shortcuts `Ctrl-X`, `Ctrl-C` and `Ctrl-V` respectively.

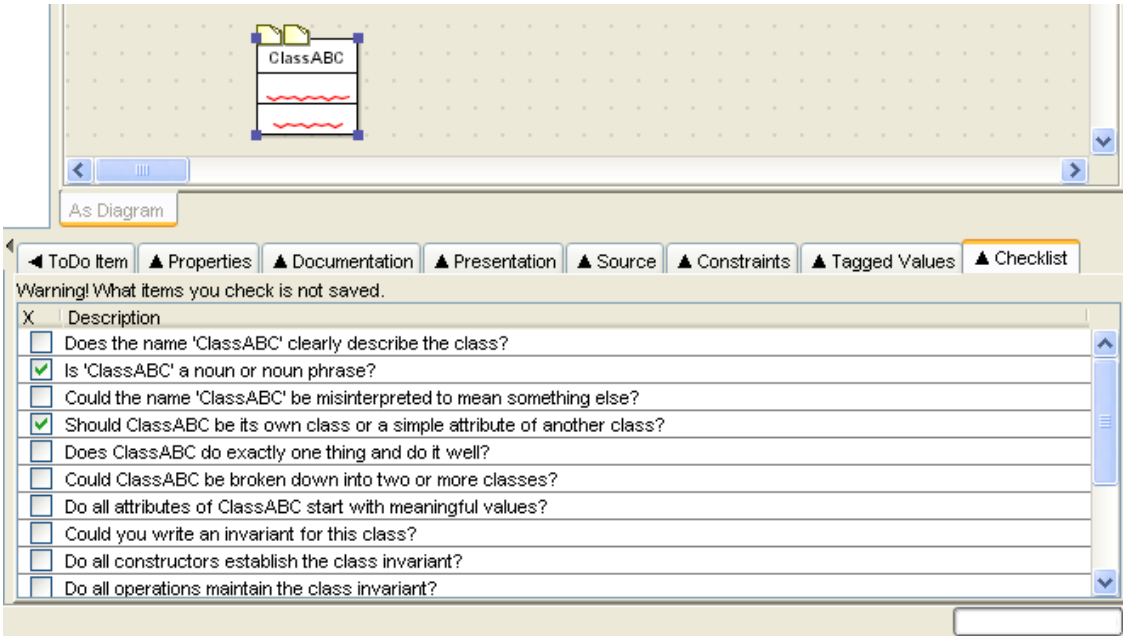
13.10. Checklist Tab

Conducting design reviews and inspections is one of the most effective ways of detecting errors during software development. A design review typically consists of a small number of designers, implementers, or other project stakeholders holding a meeting to review a software development artifact. Many development organizations have developed checklists of common design problems for use in design review meetings. Recent research indicated that reviewers inspecting code without meeting, makeing use of these checklists, are just as effective as design review meetings.

Hence, a checklist feature has been added to ArgoUML, that is much in the spirit of design review checklists. However, ArgoUML's checklists are integrated into the design tool user interface and the design task.

A software designer using ArgoUML can see a review checklist for any design element. The “Checklist” tab presents a list of check-off items that is appropriate to the currently selected design element. For example, when a class is selected in a design diagram, the checklist tab shows items that prompt critical thinking about classes. See the figure below. Designers may check off items as they consider them. Checked items are kept in the list to show what has already been considered, while unchecked items prompt the designer to consider new design issues. ArgoUML supplies many different checklists with many possible items.

Figure 13.18. An example of a checklist for a class.



Caution

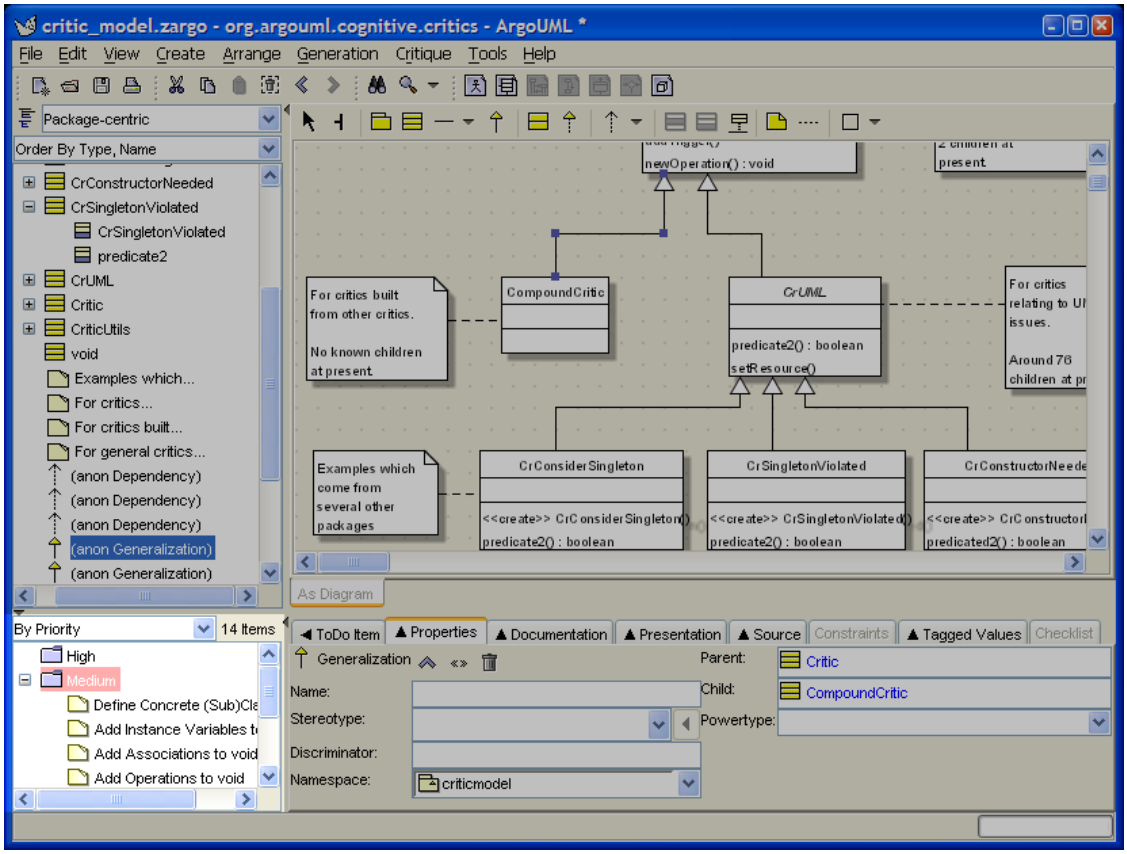
In the V0.22 release of ArgoUML, this tab is not completely implemented. E.g. the checks are not saved.

Chapter 14. The To-Do Pane

14.1. Introduction

Figure 14.1, “Overview of the to-do pane” shows the ArgoUML window with the to-do pane highlighted.

Figure 14.1. Overview of the to-do pane



This pane provides access to the advice that comes from the critics processes running within ArgoUML.

A selector box at the top allows a choice of how the data is presented, a button allows the display of the hierarchy to be changed, and there is an indicator of the number of to-do items identified.

More information on critics can be found in the discussion of the `Critique` menu (see Section 10.9, “The Critique Menu”).

14.2. Mouse Behavior in the To-Do Pane

Behavior of the mouse in general, and the naming of the buttons is covered in the chapter on the overall user interface (see Chapter 8, *Introduction*).

14.2.1. Button 1 Click

This action is generally used to select an item for subsequent operations.

Within the hierarchical display, elements which have sub-hierarchies may be indicated by  when the hierarchy is hidden and  when the hierarchy is open.

When these icons are displayed, the display of the hierarchy is toggled by button 1 click on these icons.

Button 1 click over the headline of any to-do item will cause its details to be shown in the `To Do Item` tab of the details pane. That tab is automatically selected if it is not currently visible.

14.2.2. Button 1 Double Click

When applied to the folder icon alongside a hierarchy category, this will cause the display of that hierarchy to be toggled.

When applied to a headline, button 1 double click will show the diagram for the model element to which the to-do item applies in the editing pane and select the model element on the diagram using an appropriate clarifier (the model element may be highlighted, underlined with a wavy line or surrounded by a colored box as appropriate).

14.2.3. Button 2 Actions

There are no button 2 functions in the to-do pane.

14.2.4. Button 2 Double Click

There are no button 2 functions in the to-do pane.

14.3. Presentation Selection

At the top of the pane is a drop-down selector controlling how the to-do items are presented. The to-do items may be presented in six different ways. This setting is not stored persistently, i.e. it is on its default vallue when ArgoUML is started.

- `By Priority`. This is the default setting. The to-do items are organized into three hierarchies by priority: `High`, `Medium` and `Low`. The priority associated with the to-do items generated by a particular critic may be altered through the `Critique > Browse Critics...` menu (see Section 10.9.4, “Browse Critics...”).
- `By Decision`. The to-do items are organized into 17 hierarchies by design issue: `Uncategorized`, `Class Selection`, `Behavior`, `Naming`, `Storage`, `Inheritance`, `Containment`, `Planned Extensions`, `State Machines`, `Design Patterns`, `Relationships`, `Instantiation`, `Modularity`, `Expected Usage`, `Methods`, `Code Generation` and `Stereotypes`. The details of the critics in each category are discussed in Section 10.9.2, “Design Issues...”.
- `By Goal`. ArgoUML has a concept that critics may be grouped according to the user goals they affect. This presentation groups the to-do items into hierarchies by goal.

Caution

In the current release of ArgoUML there is only one goal, `Unspecified` and all to-do items will appear under this heading.

- `By Offender`. The to-do items are organized into a hierarchy according to the model element that caused the problem. Todo items that were manually created with the "New ToDo item" button (i.e. not by a critic), are not listed here.
- `By Poster`. The to-do items are organized into a hierarchy according to which critic generated the to-do item. The class name of the critic is listed instead of just its headline name since the former is guaranteed to be a unique name.
- `By Knowledge Type`. ArgoUML has the concept that a critic reflects a deficiency in a category of knowledge. This presentation option groups the critics according to their knowledge category: `Designer's`, `Correctness`, `Completeness`, `Consistency`, `Syntax`, `Semantics`, `Optimization`, `Presentational`, `Organizational`, `Experiencial` and `Tool`. The former category (Designer's) contains the manually entered todo items.

14.4. Item Count

To the right of the flat/hierarchical button is a count of the number of to-do items currently found. It will be highlighted in yellow when the number of to-do items grows above 50 todo items, and red when above 100.

Chapter 15. The Critics

15.1. Introduction

The key feature that distinguishes ArgoUML from other UML CASE tools is its use of concepts from cognitive psychology. The theory behind this is well described in Jason Robbins' PhD dissertation http://argouml.tigris.org/docs/robbins_dissertation/ [http://argouml.tigris.org/docs/robbins_dissertation/].

Critics are one of the main ways in which these ideas are implemented. Running in the background they offer advice to the designer which may be accepted or ignored. A key point is that they do not *impose* a decision on the designer.

Note

The critics are asynchronous processes that run in parallel with the main ArgoUML tool. Changes typically take a second or two to propagate as the critics wake up.

15.1.1. Terminology

The *critics* are background processes, which evaluate the current model according to various “good” design criteria. There is one critic for every design criterion.

The output of a critic is a *critique*—a statement about some aspect of the model that does not appear to follow good design practice.

Finally a critique will generally suggest how the bad design issue it has identified can be rectified, by raising a *to-do item*.

15.1.2. Design Issues

ArgoUML categorizes critics according the the design issue they address (some critics may be in more than one category). At present there are 16 such categories.

Within this manual the descriptions of critics are grouped in sections by design issue.

15.2. Uncategorized

These are critics that do not fit into any other category.

ArgoUML has no critics in this category. Maybe some will be added in later versions.

15.3. Class Selection

These are critics concerning how classes are chosen and used.

ArgoUML has the following critics in this category.

15.3.1. Wrap DataType

DataTypes are not full classes within UML 1.4. They can only have enumeration literals as values, and only support `query` operations (that is operations that do not change the DataType's state).

DataTypes cannot be associated with classes, unless the DataType is part of a composite (black diamond) aggregation. Such an association relects the tight binding of a collection of DataType instances to a class instance. In effect such a DataType is an attribute of the class with multiplicity.

Good OOA&D depends on careful choices about which entities to represent as full objects and which to represent as attributes of objects.

There are two options to fix this problem.

- Replace the DataType with a full class.
- or change the association aggregation to composite relationship at the DataType end.

15.3.2. Reduce Classes in diagram <diagram>

Suggestion to improve readability by having fewer classes on a diagram. If one class diagram has too many classes it may become very difficult for humans to understand. Defining an understandable set of class diagrams is an important part of your design.

The Wizard of this critic allows setting of the treshold, i.e. the maximum number of classes allowed before this critic fires.

Caution

This number is not stored persistently, and there is no way to reduce it after it has been set higher, except by creating more classes until the critic fires again. Restarting ArgoUML resets this number to its default: 20.

15.3.3. Clean Up Diagram

Suggestion that the diagram could be improved by moving model elements that are overlapping.

15.4. Naming

These are critics concerning the naming of model elements. The current version of ArgoUML has 18 critics in this category.

15.4.1. Resolve Association Name Conflict

Suggestion that two association names in the same namespace have the same name. This is not permitted in UML.

15.4.2. Revise Attribute Names to Avoid Conflict

Suggestion that two attribute names of a class have the same name. This is not permitted in UML.

Note

The problem may be caused by inheritance of an attribute through a generalization relationship.

15.4.3. Change Names or Signatures in a model element

Two operations in <model element> have the same signature. This means their name is the same, and the list of parameters has the same type.

Where there are conflicting signatures, correct code cannot be generated for mainstream OO languages. It also leads to very unclear semantics of the design.

In comparing signatures, this critic considers:

1. the name;
2. the list of in, out and in-out parameter types *in order*; and

Only if these all match in both type and order, will the signatures be considered as the same.

This follows the line of Java/C++ in ignoring the return parameters for the signature. This *may* be unsatisfactory for some functional OO languages.

Note

Some purists would argue that the comparison should really differentiate between in, out and in-out parameters. However no practical programming language can do this when resolving an overloaded method invocation, so this critics lumps them all together.

15.4.4. Duplicate End (Role) Names for an Association

The specified association has two (or more) ends (roles) with the same name. One of the well-formedness rules in UML 1.4 for associations, is that all end (role) names must be unique.

This ensures that there can be unambiguous reference to the ends of the association.

To fix this, manually select the association and change the names of one or more of the offending ends (roles) using the button 2 pop-up menu or the property sheet.

15.4.5. Role name conflicts with member

A suggestions that good design avoids role names for associations that clash with attributes or operations of the source class. Roles may be realized in the code as attributes or operations, causing code generation problems.

15.4.6. Choose a Name (Classes and Interfaces)

The class or interface concerned has been given no name (it will appear in the model as `Unnamed`). Suggestion that good design requires that all interfaces and classes are named.

15.4.7. Choose a Unique Name for a model element (Classes and Interfaces)

Suggestion that the class or interface specified has the same name as another (in the namespace), which is bad design and will prevent valid code generation.

15.4.8. Choose a Name (Attributes)

The attribute concerned has been given no name (it will appear in the model as `(Unnamed Attribute)`). Suggestion that good design requires that all attributes are named.

15.4.9. Choose a Name (Operations)

The operation concerned has been given no name (it will appear in the model as `(Unnamed Operation)`). Suggestion that good design requires that all operations are named.

15.4.10. Choose a Name (States)

The state concerned has been given no name (it will appear in the model as `(Unnamed State)`). Suggestion that good design requires that all states are named.

15.4.11. Choose a Unique Name for a (State related) model element

Suggestion that the state specified has the same name as another (in the current statechart diagram), which is bad design and will prevent valid code generation.

15.4.12. Revise Name to Avoid Confusion

Two names in the same namespace have very similar names (differing only by one character). Suggestion this could potentially lead to confusion.

Caution

This critic can be particularly annoying, since at times it is useful and good design to have a series of model elements `var1`, `var2` etc.

It is important to remember that critics offer guidance, and are not always correct. ArgoUML lets you dismiss the resulting to-do items through the to-do pane (see Chapter 14, *The To-Do Pane*).

15.4.13. Choose a Legal Name

All model element names in ArgoUML must use only letters, digits and underscore characters. This critic suggests an entity has not met this requirement.

15.4.14. Change a model element to a Non-Reserved Word

Suggestion that this model element's name is the same as a reserved word in UML (or within one character of one), which is not permitted.

15.4.15. Choose a Better Operation Name

Suggestion that an operation has not followed the naming convention that operation names begin with lower case letters.

Caution

Following the Java and C++ convention most designers give their constructors the same name as the class, which begins with an upper case character. In ArgoUML, this will trigger this critic, unless the constructor is stereotyped «`create`».

It is important to remember that critics offer guidance, and are not always correct. ArgoUML lets you dismiss the resulting to-do items through the to-do pane (see Chapter 14, *The To-Do Pane*).

15.4.16. Choose a Better Attribute Name

Suggestion that an attribute has not followed the naming convention that attribute names begin with lower case letters.

15.4.17. Capitalize Class Name

Suggestion that a class has not followed the naming convention that classes begin with upper case letters.

Note

Although not triggering this critic, the same convention should apply to interfaces.

15.4.18. Revise Package Name

Suggestion that a package has not followed the naming convention of using lower case letters with periods used to indicated sub-packages.

15.5. Storage

Critics concerning attributes of classes.

The current version of ArgoUML has the following critics in this category.

15.5.1. Revise Attribute Names to Avoid Conflict

This critic is discussed under an earlier design issues category (see Section 15.4.2, “Revise Attribute Names to Avoid Conflict”).

15.5.2. Add Instance Variables to a Class

Suggestion that no instance variables have been specified for the given class. Such classes may be created to specify static attributes and methods, but by convention should then be given the stereotype «`utility`».

15.5.3. Add a Constructor to a Class

You have not yet defined a constructor for class *class*. Constructors initialize new instances such that their attributes have valid values. This class probably needs a constructor because not all of its attributes have initial values.

Defining good constructors is key to establishing class invariants, and class invariants are a powerful aid in writing solid code.

To fix this, add a constructor manually by clicking on *class* in the explorer and adding an operation using the context sensitive pop-up menu in the property tab, or select *class* where it appears on a class diagram and use the `Add Operation` tool.

In the UML 1.4 standard, a constructor is an operation with the stereotype «create». Although not strictly standard, ArgoUML will also accept «Create» as a stereotype for constructors.

By convention in Java and C++ a constructor has the same name as the class, is not static, and returns no value. ArgoUML will also accept any operation that follows these conventions as a constructor even if it is not stereotyped «create».

Caution

Operators are created in ArgoUML with a default return parameter (named `return`). You will need to remove this parameter to meet the Java/C++ convention.

15.5.4. Reduce Attributes on a Class

Suggestion that the class has too many attributes for a good design, and is at risk of becoming a design bottleneck.

The Wizard of this critic allows setting of the treshold, i.e. the maximum number of attributes allowed before this critic fires.

Caution

This number is not stored persistently, and there is no way to reduce it after it has been set higher, except by creating more attributes until the critic fires again. Restarting ArgoUML resets this number to its default: 7.

15.6. Planned Extensions

Critics concerning interfaces and subclasses.

Note

It is not clear why this category has the name “Planned Extensions”.

The current version of ArgoUML has three critics in this category.

15.6.1. Operations in Interfaces must be public

Suggestion that there is no point in having non-public operations in Interfaces, since they must be visible to be realized by a class.

15.6.2. Interfaces may only have operations

Suggestion that an interfaces has attributes defined. The UML standard defines interfaces to have operations.

Caution

ArgoUML does not allow you to add attributes to interfaces, so this should never occur in the ArgoUML model. It might trigger if a project has been loaded with XMI created by another tool.

15.6.3. Remove Reference to Specific Subclass

Suggestion that in a good design, a class should not reference its subclasses directly through attributes, operations or associations.

15.7. State Machines

Critics concerning state machines.

ArgoUML has the following critics in this category.

15.7.1. Reduce Transitions on <state>

Suggestion given state is involved in so many transitions it may be a maintenance bottleneck.

The Wizard of this critic allows setting of the treshold, i.e. the maximum number of transitions allowed before this critic fires.

Caution

This number is not stored persistently, and there is no way to reduce it after it has been set higher, except by creating more transition until the critic fires again. Restarting ArgoUML resets this number to its default: 10.

15.7.2. Reduce States in machine <machine>

Suggestion that the given state machine has so many states as to be confusing and should be simplified (perhaps by breaking into several machines, or using a hierarchy).

The Wizard of this critic allows setting of the treshold, i.e. the maximum number of states allowed before this critic fires.

Caution

This number is not stored persistently, and there is no way to reduce it after it has been set higher, except by creating more states until the critic fires again. Restarting ArgoUML resets this number to its default: 20.

15.7.3. Add Transitions to <state>

Suggestion that the given state requires both incoming and outgoing transitions.

15.7.4. Add Incoming Transitions to <model element>

Suggestion that the given state requires incoming transitions.

15.7.5. Add Outgoing Transitions from <model element>

Suggestion that the given state requires outgoing transitions.

15.7.6. Remove Extra Initial States

Suggestion that there is more than one initial state in the state machine or composite state, which is not permitted in UML.

15.7.7. Place an Initial State

Suggestion that there is no initial state in the state machine or composite state.

15.7.8. Add Trigger or Guard to Transition

Suggestion that a transition is missing either a trigger or guard, one at least of which is required for it to be taken.

15.7.9. Change Join Transitions

Suggestion that the join pseudostate has an invalid number of transitions. Normally there should be one outgoing and two or more incoming.

15.7.10. Change Fork Transitions

Suggestion that the fork pseudostate has an invalid number of transitions. Normally there should be one incoming and two or more outgoing.

15.7.11. Add Choice/Junction Transitions

Suggestion that the branch (choice or junction) pseudostate has an invalid number of transitions. Normally there should be at least one incoming transition and at least one outgoing transition.

15.7.12. Add Guard to Transition

Suggestion that the transition requires a guard.

Caution

It is not clear that this is a valid critic. It is perfectly acceptable to have a transition without a guard—the transition is always taken when the trigger is invoked.

15.7.13. Clean Up Diagram

This critic is discussed under an earlier design issues category (see Section 15.3.3, “Clean Up Diagram”).

15.7.14. Make Edge More Visible

Suggestion that an edge model element such as an association or abstraction is so short it may be missed. Move the connected model elements apart to make the edge more visible.

15.7.15. Composite Association End with Multiplicity > 1

An instance may not belong by composition to more than one composite instance. You must change the multiplicity at the composite end of the association to either 0..1 or 1..1 (1) for your model to make sense.

Remember that composition is the stronger aggregation kind and aggregation is the weaker. The problem can be compared to a model where a finger can be an integral part of several hands at the same time.

This is the second well-formedness rule on AssociationEnd in UML 1.4.

15.8. Design Patterns

Critics concerning design pattern usage in ArgoUML.

These relate to the use of patterns as described by the so called “Gang of Four”. ArgoUML also uses this category for critics associated with deployment and sequence diagrams. The current version of ArgoUML has the following critics in this category.

15.8.1. Consider using Singleton Pattern for <class>

The *class* has no non-static attributes nor any associations that are navigable away from instances of this class. This means that every instance of this class will be identical to every other instance, since there will be nothing about the instances that can differentiate them.

Under these circumstances you should consider making explicit that you have exactly one instance of this class, by using the singleton Pattern. Using the singleton pattern can save time and memory space. Within ArgoUML this can be done by using the «singleton» stereotype on this class.

If it is not your intent to have a single instance, you should define instance variables (i.e. non-static attributes) and/or outgoing associations that will represent differences bewteen instances.

Having specified *class* as a singleton, you need to define the class so there can only be a single instance. This will complete the information representation part of your design. To achieve this you need to do the following.

1. You must define a static attribute (a class variable) holding the instance. This must therefore have *class* as its type.

2. You must have only private constructors so that new instances cannot be made by other code. The creation of the single instance could be through a suitable helper operation, which invokes this private constructor just once.

3. You must have at least one constructor to override the default constructor, so that the default constructor is not used to create multiple instances.

For the definition of a constructor under the UML 1.4 standard, and extensions to that definition accepted by ArgoUML see Section 15.5.3, “Add a Constructor to a Class” .

15.8.2. Singleton Stereotype Violated in <class>

This class is marked with the «singleton» stereotype, but it does not satisfy the constraints imposed on singletons (ArgoUML will also accept «Singleton» stereotype as defining a singleton). A singleton class can have at most one instance. This means that the class must meet the design criteria for a singleton (see Section 15.8.1, “Consider using Singleton Pattern for <class>”).

Whenever you mark a class with a stereotype, the class should satisfy all constraints of the stereotype. This is an important part of making a self-consistent and understangle design. Using the singleton pattern can save time and memory space.

If you no longer want this class to be a singleton, remove the «singleton» stereotype by clicking on the class and selecting the blank selection on the stereotype drop-down within the properties tab.

To apply the singleton pattern you should follow the directions in Section 15.8.1, “Consider using Singleton Pattern for <class>” .

15.8.3. Nodes normally have no enclosers

A suggestion that nodes should not be drawn inside other model elements on the deployment diagram, since they represent an autonomous physical object.

15.8.4. NodeInstances normally have no enclosers

A suggestion that node instances should not be drawn inside other model elements on the deployment diagram, since they represent an autonomous physical object.

15.8.5. Components normally are inside nodes

A suggestion that components represent the logical entities within physical nodes, and so should be drawn within a node, where nodes are shown on the deployment diagram.

15.8.6. ComponentInstances normally are inside nodes

A suggestion that component instances represent the logical entities within physical nodes, and so should be drawn within a node instance, where node instances are shown on the deployment diagram.

15.8.7. Classes normally are inside components

A suggestion that classes, as model elements making up components, should be drawn within components on the deployment diagram.

15.8.8. Interfaces normally are inside components

A suggestion that interfaces, as model elements making up components, should be drawn within components on the deployment diagram.

15.8.9. Objects normally are inside components

A suggestion that objects, as instances of model elements making up components, should be drawn within components or component instances on the deployment diagram.

15.8.10. LinkEnds have not the same locations

A suggestion that a link (e.g. association) connecting objects on a deployment diagram has one end in a

component and the other in a component instance (since objects can be in either). This makes no sense.

15.8.11. Set classifier (Deployment Diagram)

Suggestion that there is an instance (object) without an associated classifier (class, datatype) on a deployment diagram.

15.8.12. Missing return-actions

Suggestion that a sequence diagram has a send or call action without a corresponding return action.

15.8.13. Missing call(send)-action

Suggestion that a sequence diagram has a return action, but no preceding call or send action.

15.8.14. No Stimuli on these links

Suggestion that a sequence diagram has a link connecting objects without an associated stimulus (without which the link is meaningless).

Warning

Triggering this critic indicates a serious problem, since ArgoUML provides no mechanism for creating a link without a stimulus. It probably indicates that the diagram was created by loading a corrupt project, with an XMI file describing a link without a stimulus, possibly created by a tool other than ArgoUML.

15.8.15. Set Classifier (Sequence Diagram)

Suggestion that there is an object without an associated classifier (class, datatype) on a sequence diagram.

15.8.16. Wrong position of these stimuli

Suggestion that the initiation of send/call-return message exchanges in a sequence diagram does not properly initiate from left to right.

15.9. Relationships

Critics concerning associations in ArgoUML.

The current version of ArgoUML has the following critics in this category.

15.9.1. Circular Association

Suggestion that an association class has a role that refers back directly to itself, which is not permitted.

Warning

This critic is meaningless in the V0.14 version of ArgoUML which does not support association classes.

15.9.2. Make <association> Navigable

Suggestion that the association referred to is not navigable in either direction. This is permitted in the UML standard, but has no obvious meaning in any practical design.

15.9.3. Remove Navigation from Interface via <association>

Associations involving an interface can be not be navigable in the direction from the interface. This is because interfaces contain only operation declarations and cannot hold pointers to other objects.

This part of the design should be changed before you can generate code from this design. If you do generate code before fixing this problem, the code will not match the design.

To fix this, select the association and use the `Properties` tab to select in turn each association end that is *not* connected to the interface. Uncheck `Navigable` for each of these ends.

The association should then appear with a stick arrowhead pointed towards the interface

When an association between a class and interface is created in ArgoUML, it is by default navigable only from the class to the interface. However, ArgoUML does not prevent to change the navigability afterwards into a wrong situation. Which will cause this critic to be triggered.

15.9.4. Add Associations to <model element>

Suggestion that the specified model element (actor, use case or class) has no associations connecting it to other model elements. This is required for the model element to be useful in a design.

15.9.5. Remove Reference to Specific Subclass

This critic is discussed under an earlier design issues category (see Section 15.6.3, “Remove Reference to Specific Subclass”).

15.9.6. Reduce Associations on <model element>

Suggestion that the given model element (actor, use case, class or interface) has so many associations it may be a maintenance bottleneck.

The Wizard of this critic allows setting of the treshold, i.e. the maximum number of associations allowed before this critic fires.

Caution

This number is not stored persistently, and there is no way to reduce it after it has been set higher, except by creating more associations until the critic fires again. Restarting ArgoUML resets this number to its default: 7.

15.9.7. Make Edge More Visible

This critic is discussed under an earlier design issues category (see Section 15.7.14, “Make Edge More Visible”).

15.10. Instantiation

Critics concerning instantiation of classifiers in ArgoUML.

The current version of ArgoUML has no critics in this category.

15.11. Modularity

Critics concerning modular development in ArgoUML.

The current version of ArgoUML has the following critics in this category.

15.11.1. Classifier not in Namespace of its Association

One of the well-formedness rules in UML 1.4 for associations, is that all the classifiers attached to the ends of the association should belong to the same namespace as the association.

If this were not the case, there would be no naming, by which each end could refer to all the others.

This critic is triggered when an association does not meet this criterion. The solution is to delete the association, and recreate it on a diagram, whose namespace includes those of all the attached classifiers.

Caution

In the current implementation of ArgoUML this critic does not handle hierarchical namespaces. As a consequence it will trigger for associations where the immediate namespaces of the attached classifiers is different, even though they are part of the same namespace hierarchy.

15.11.2. Add Elements to Package <package>

Suggestion that the specified package has no content. Good design suggests packages are created to put things in.

Note

This will always trigger when you first create a package, since you cannot create one that is not empty!

15.12. Expected Usage

Critics concerning generally accepted good practice in ArgoUML.

The current version of ArgoUML has one critic in this category.

15.12.1. Clean Up Diagram

This critic is discussed under an earlier design issues category (see Section 15.3.3, “Clean Up Diagram”).

15.13. Methods

Critics concerning operations in ArgoUML.

The current version of ArgoUML has the following critics in this category.

15.13.1. Change Names or Signatures in <model element>

This critic is discussed under an earlier design issues category (see Section 15.4.3, “Change Names or Signatures in a model element”).

15.13.2. Class Must be Abstract

Suggestion that a class that inherits or defines abstract operations must be marked abstract.

15.13.3. Add Operations to <class>

Suggestion that the specified class has no operations defined. This is required for the class to be useful in a design.

15.13.4. Reduce Operations on <model element>

Suggestion that the model element (class or interface) has too many operations for a good design, and is at risk of becoming a design bottleneck.

The Wizard of this critic allows setting of the treshold, i.e. the maximum number of operations allowed before this critic fires.

Caution

This number is not stored persistently, and there is no way to reduce it after it has been set higher, except by creating more operations until the critic fires again. Restarting ArgoUML resets this number to its default: 20.

15.14. Code Generation

Critics concerning code generation in ArgoUML.

The current version of ArgoUML has one critic in this category.

15.14.1. Change Multiple Inheritance to interfaces

Suggestion that a class has multiple generalizations, which is permitted by UML, but cannot be generated into Java code, because Java does not support multiple inheritance.

15.15. Stereotypes

Critics concerning stereotypes in ArgoUML.

The current version of ArgoUML has no critics in this category.

15.16. Inheritance

Critics concerning generalization and specialization in ArgoUML.

The current version of ArgoUML has the following critics in this category.

15.16.1. Revise Attribute Names to Avoid Conflict

This critic is discussed under an earlier design issues category (see Section 15.4.2, “Revise Attribute Names to Avoid Conflict”).

15.16.2. Remove <class>'s Circular Inheritance

Suggestion that a class inherits from itself, through a chain of generalizations, which is not permitted.

Caution

This critic is marked inactive by default in the current release of ArgoUML (the only one so marked). It will not trigger unless made active.

15.16.3. Class Must be Abstract

This critic is discussed under an earlier design issues category (see Section 15.13.2, “Class Must be Abstract”).

15.16.4. Remove final keyword or remove subclasses

Suggestion that a class that is final has specializations, which is not permitted in UML.

15.16.5. Illegal Generalization

Suggestion that there is a generalization between model elements of different UML metaclasses, which is not permitted.

Caution

It is not clear that such a generalization can be created within ArgoUML. It probably indicates that the diagram was created by loading a corrupt project, with an XMI file describing such a generalization, possibly created by a tool other than ArgoUML.

15.16.6. Remove Unneeded Realizes from <class>

Suggestion that the specified class has a realization relationship both directly and indirectly to the same interface (by realization from two interfaces, one of which is a generalization of the other for example). Good design deprecates such duplication.

15.16.7. Define Concrete (Sub)Class

Suggestion that a class is abstract with no concrete subclasses, and so can never be realized.

15.16.8. Define Class to Implement <interface>

Suggestion that the interface referred to has no influence on the running system, since it is never implemented by a class.

15.16.9. Change Multiple Inheritance to interfaces

This critic is discussed under an earlier design issues category (see Section 15.14.1, “Change Multiple Inheritance to interfaces”).

15.16.10. Make Edge More Visible

This critic is discussed under an earlier design issues category (see Section 15.7.14, “Make Edge More Visible”).

15.17. Containment

Critics concerning containment in ArgoUML, that is where one model element forms a component part of another.

The current version of ArgoUML has the following critics in this category.

15.17.1. Remove Circular Composition

Suggestion that there is a series of composition relationships (associations with black diamonds) that form a cycle, which is not permitted.

15.17.2. Duplicate Parameter Name

Suggestion that a parameter list to an operation or event has two or more parameters with the same name, which is not permitted.

15.17.3. Two Aggregate Ends (Roles) in Binary Association

Only one end (role) of a binary association can be aggregate or composite. This a well-formedness rule of the UML 1.4 standard.

Aggregation and composition are used to indicate whole-part relationships, and by definition, the “part” end cannot be aggregate.

To fix this, identify the “part” end of the association, and use the critic wizard (the `Next>` button, or manually set its aggregation to `none` using the button 2 pop-up menu or the property sheet.

Composition (more correctly called composite aggregation) is used where there is a whole-part relationship that is one-to-one or one-to-many, and the lifetime of the part is inextricably tied to the lifetime of the whole. Instances of the whole will have responsibility for creating and destroying instances of the associated part. This also means that a class can only be a part in one composite aggregation.

An example of a composite aggregation might be a database of cars and their wheels. This is a one-to-four relationship, and the database entry for a wheel is associated with its car. When the car ceases to exist in the database, so do its wheels.

Aggregation (more correctly called shared aggregation) is used where there is a whole-part relationship,

that does not meet the criteria for a composite aggregation. An example might be a database of university courses and the students that attend them. There is a whole-part relationship between courses and students. However there is no lifetime relationship between students and course (a student continues to exist even after a course is finished) and the relationship is many-to-many.

15.17.4. Aggregate End (Role) in 3-way (or More) Association

Three-way (or more) associations can not have aggregate ends (roles). This a well-formedness rule of the UML 1.4 standard.

Aggregation and composition are used to indicate whole-part relationships, and by definition can only apply to binary associations between model elements.

To fix this, manually select the association, and set the aggregation of each of its ends (roles) to none using the button 2 pop-up menu or the property sheet.

15.17.5. Wrap DataType

This critic is discussed under an earlier design issues category (see Section 15.3.1, “Wrap DataType”).

Part 3. Model Reference

Chapter 16. Top Level Model Element Reference

16.1. Introduction

This chapter describes each model element that can be created within ArgoUML. The chapter covers top-level "general" model elements. The following chapters (see Chapter 17, *Use Case Diagram Model Element Reference* through Chapter 23, *Deployment Diagram Model Element Reference*) cover each of the ArgoUML diagrams.

There is a close relationship between this material and the properties tab of the details pane (see Section 13.3, "Properties Tab"). That section covers properties in general, in this chapter they are linked to specific model elements.

16.2. The Model

The model is the top level model element within ArgoUML. In the UML meta-model it is a sub-class of package. In many respects within ArgoUML it behaves similarly to a package (see Section 18.2, "Package").

Note

ArgoUML is restricted to one model within the tool.

Standard data types, classes and packages are loaded (the default, see Chapter 24, *Built In DataTypes, Classes, Interfaces and Stereotypes*) as sub-packages of the model. These sub-packages are not initially present in the model but are added to the model when used.

16.2.1. Model Details Tabs

The details tabs that are active for the model are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 16.2.2, "Model Property Toolbar" and Section 16.2.3, "Property Fields For The Model" below.

`Documentation`
: Standard tab. See Section 13.4, "Documentation Tab".

`Stereotype`
: Standard tab. This contains a a list of the stereotypes applied to this model, and a list of available stereotypes that may be applied to the model.

`Tagged Values`
: Standard tab. In the UML meta-model, `Model` has the following standard tagged values defined.

  - `derived` (from the superclass, `ModelElement`).

Values `true`, meaning the class is redundant — it can be formally derived from other elements, or `false` meaning it cannot.

Derived models have their value in analysis to introduce useful names or concepts, and in design to avoid re-computation.

16.2.2. Model Property Toolbar


`Go up`

Navigate up through the composition structure of the model.

Since the model is the top package nothing can happen, and this button is allways downlighted.


`New Package`

This creates a new Package (see Section 18.2, “Package”) within the model (which appears on no diagram), navigating immediately to the properties tab for that package.

Tip

While it can make sense to create Packages of the model this way, it is usually a lot clearer to create them within diagrams where you want them.


`New DataType`

This creates a new DataType (see Section 16.3, “Datatype”) within the model (which appears on no diagram), navigating immediately to the properties tab for that DataType.


`New Enumeration`

This creates a new Enumeration (see Section 16.4, “Enumeration”) within the model (which appears on no diagram), navigating immediately to the properties tab for that Enumeration.


`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) within the model, navigating immediately to the properties tab for that stereotype.


`Delete`

This tool is always downlighted, since it is meaningless to delete the model!

16.2.3. Property Fields For The Model

`Name`

Text box. The name of the model. The name of a model, like all packages, is by convention all lower case.

Note

The default name supplied to a new model by ArgoUML, `untitledModel`, is thus erroneous and guarantees that ArgoUML always starts up with at least one problem being reported by the design critics.

`Stereotype`

Drop down selector. Model is provided by default with the UML standard stereotypes for model (`systemModel` and `metamodel`) and package (`facade`, `framework`, `stub`).

Stereotyping models is a useful thing, although it is of limited value in ArgoUML where you have only a single model.

`Navigate Stereotype`

 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 16.6, “Stereotype”).

`Namespace`

Text box. Records the namespace for the model. This is the package hierarchy. However since the model is at the top of the hierarchy in ArgoUML, this box is always empty.

`Visibility`

Radio box, with entries `public`, `private`, `protected`, and `package`.

Records the visibility for the model. Since ArgoUML only permits one model, this has no meaningful use.

`Modifiers`

Check box, with entries `Abstract`, `Leaf` and `Root`.

- `abstract` is used to declare that this model cannot be instantiated, but must always be specialized.

  The meaning of `abstract` applied to a model is not that clear. It might mean that the model contains interfaces or abstract classes without realizations. Since ArgoUML only permits one model, this is not a meaningful box to check.

- `Leaf` indicates that this model can have no further subpackages, while `root` indicates it is the top level model.

  Within ArgoUML `root` only meaningfully applies to the Model, since all packages sit within the model. In the absence of the `topLevel` stereotype, this could be used to emphasize that the Model is at the top level.

`Generalizations`

Text area. Lists any model that *generalizes* this model.

Note

Since there is only one model in ArgoUML there is no sensible specialization or generalization that could be created.

`Specializations`
Text box. Lists any specialized model (i.e. for which this model is a generalization.

Note

Since there is only one model in ArgoUML there is no sensible specialization or generalization that could be created.

`Owned Elements`

Text area. A listing of the top level packages, classes, interfaces, datatypes, actors, use cases, associations, generalizations, and stereotypes within the model.

Button 1 double click on any of the model elements yields navigating to that model element.

16.3. Datatype

Datatypes can be thought of as simple classes. They have no attributes, and any operations on them must have no side-effects. A useful analogy is primitive datatypes in a language like Java. The integer “3” stands on its own—it has no inner structure. There are operations (for example addition) on the integers, but when I perform `3 + 4` the result is a new number, “3” and “4” are unchanged by the exercise.

Within UML 1.4, `DataType` is a sub-class of the `Classifier` metaclass. It embraces the predefined primitive types (`byte`, `char`, `double`, `float`, `int`, `long` and `short`), the predefined enumeration, `boolean` and user defined *enumeration types*.

Note

Also `void` is implemented as a datatype within ArgoUML

Within ArgoUML new datatypes may be created using the `New datatype` button on the property tabs of the model and packages (in which case the new datatype is restricted in scope to the package), as well as the properties tab for datatype. Datatypes can also be created with the tool in the diagram toolbar of a class diagram.

The UML 1.4 standard allows user defined datatypes to be placed on class diagrams to define their inheritence structure. This is also possible in ArgoUML. It is represented on the diagram by a box with two compartments, of which the top one is marked with «`datatype`», and contains the name. The lower one contains operations.

16.3.1. Datatype Details Tabs

The details tabs that are active for datatypes are as follows.

`ToDoItem`
Standard tab.

`Properties`

See Section 16.3.2, “Datatype Property Toolbar” and Section 16.3.3, “Property Fields For Datatype” below.

`Documentation`

Standard tab. See Section 13.4, “Documentation Tab”.

`Source`
Standard tab. Unused. One would expect a class declaration for the new datatype to support code generation.

`Tagged Values`

Standard tab. In the UML metamodel, `Datatype` has the following standard tagged values defined.

- `persistence` (from the superclass, `Classifier`). Values `transitory`, indicating state is destroyed when an instance is destroyed or `persistent`, marking state is preserved when an instance is destroyed.

Tip

Since user defined datatypes are enumerations, they have no state to preserve, and the value of this tagged value is irrelevant.

- `semantics` (from the superclass, `Classifier`). The value is a specification of the semantics of the datatype.
- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the class is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Tip

While formally available, a derived datatype does not have an obvious value, and so datatypes should always be marked with `derived=false`.

16.3.2. Datatype Property Toolbar


`Go up`

Navigate up through the package structure.


`New datatype`

This creates a new datatype (see Section 18.6, “Class”) within the same package as the current datatype.

Tip

While it can make sense to create datatypes this way, it can be clearer to create them within the package or model where you want them.


`New Enumeration`

This creates a new Enumeration (see Section 16.4, “Enumeration”) in the same package as the datatype, navigating immediately to the properties tab for that Enumeration.


`New Operation`

This creates a new operation within the datatype, navigating immediately to the properties tab for that operation.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) within the same package as the datatype, navigating immediately to the properties tab for that stereotype.

 `Delete`

This deletes the datatype from the model.

16.3.3. Property Fields For Datatype

`Name`

Text box. The name of the datatype. The primitive datatypes all have lower case names, but there is no formal convention.

Note

The default name supplied for a newly created datatype is the empty string “”. Datatypes with empty string names will appear with the name `(Unnamed Datatype)` in the explorer.

`Namespace`

Drop down selector with navigate button. Allows changing the namespace for the datatype. This is the package hierarchy.

`Modifiers`

Check box, with entries `Abstract`, `Leaf` and `Root`.

- `Abstract` is used to declare that this datatype cannot be instantiated, but must always be specialized.

  ### Note

  ArgoUML provides no mechanism for specializing datatypes, so this check box is of little use.

- `Leaf` indicates that this datatype can have no further sub-types, while `Root` indicates it is a top level datatype.

  ### Tip

  You can define the specialization of datatypes in a class diagram by drawing generalizations between them.

`Visibility`

Radio box, with entries `public`, `private`, `protected`, and `package`.

Records the visibility for the Datatype.

`Client Dependencies`
Text area. Lists any elements that depend on this datatype.

Caution

It is not clear that dependencies between datatypes makes much sense.

`Supplier Dependencies`
Text area. Lists any elements that this datatype depends on.

Caution

It is not clear that dependencies between datatypes makes much sense.

`Generalizations`
Text area. Lists any datatype that *generalizes* this datatype.

`Specializations`
Text box. Lists any specialized datatype (i.e. for which this datatype is a generalization.

`Operations`
Text area. Lists all the operations defined on this datatype. Button 1 double click navigates to the selected operation. button 2 click brings up a pop up menu with two entries.

- `Move Up`. Only available where there are two or more operations, and the operation selected is not at the top. It is moved up one.
- `Move Down`. Only available where there are two or more operations listed, and the operation selected is not at the bottom. It is moved down one.

See Section 18.8, “Operation” for details of operations.

Caution

ArgoUML treats all operations as equivalent. Any operations created here will use the same mechanism as operations for classes. Remember that operations on datatypes must have no side effects (they are read-only). This means the `query` modifier *must* be checked for all operations.

16.4. Enumeration

An enumeration is a primitive datatype that can have a fixed short list of values. It has no attributes, and any operations on them must have no side-effects. A useful analogy is the primitive datatype boolean in a language like Java. The boolean stands on its own—it has no inner structure. There are operations (for example logical xor) on the booleans, but when I perform `true xor true` the result is a new boolean, and the original 2 booleans “true” are unchanged by the exercise.

Within UML 1.4, `Enumeration` is a sub-class of the `DataType` metaclass.

The big difference with other DataTypes, is that an Enumeration has `EnumerationLiteral`s. E.g. the Enumeration “boolean” is defined as having 2 EnumerationLiterals, “true” and “false”.

Within ArgoUML new enumerations may be created using the `New Enumeration` button on the property tabs of the model and packages (in which case the new enumeration is restricted in scope to the package), as well as the properties tab for datatype and enumeration. Enumerations can also be created with the tool in the diagram toolbar of a class diagram.

The UML 1.4 standard allows user defined enumerations to be placed on class diagrams to define their inheritence structure. This is also possible in ArgoUML. It is represented on the diagram by a box with three compartments, of which the top one is marked with «`enumeration`», and contains the name. The middle compartment shows the enumeration literals. The lower one contains operations.

16.4.1. Enumeration Details Tabs

The details tabs that are active for enumerations are as follows.

`ToDoItem`
Standard tab.

`Properties`

See Section 16.4.2, “Enumeration Property Toolbar” and Section 16.4.3, “Property Fields For Enumeration” below.

`Documentation`
Standard tab. See Section 13.4, “Documentation Tab”.

`Presentation`
Standard tab.

`Source`
Standard tab.

`Stereotype`
Standard tab. The UML metamodel has the following stereotypes defined by default for a Classifier, which also apply to an Enumeration:

- `metaclass` (from the superclass, `Classifier`).
- `powertype` (from the superclass, `Classifier`).
- `process` (from the superclass, `Classifier`).
- `thread` (from the superclass, `Classifier`).
- `utility` (from the superclass, `Classifier`).

`Tagged Values`

Standard tab. In the UML metamodel, `Enumeration` has no standard tagged values defined.

16.4.2. Enumeration Property Toolbar

 `Go up`

Navigate up through the composition structure.

`New datatype`

This creates a new datatype (see Section 18.6, “Class”) within the same package as the current enumeration.

`New enumeration`

This creates a new enumeration within the same namespace as the current enumeration, navigating immediately to the properties tab for new enumeration.

`New enumeration literal`

This creates a new enumeration literal within the enumeration, navigating immediately to the properties tab for that literal.

`New Operation`

This creates a new operation within the enumeration, navigating immediately to the properties tab for that operation.

`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) within the same package as the enumeration, navigating immediately to the properties tab for that stereotype.

`Delete from Model`

This deletes the datatype from the model.

16.4.3. Property Fields For Enumeration

`Name`

Text box. The name of the enumeration. The primitive enumerations all have lower case names, but there is no formal convention.

Note

The default name supplied for a newly created datatype is the empty string “”. Enumerations with empty string names will appear with the name `(Unnamed Enumeration)` in the explorer.

`Namespace`

Drop down selector with navigation button. Allows changing the namespace for the enumeration. This is the composition hierarchy.

`Modifiers`

Check box, with entries `Abstract`, `Leaf` and `Root`.

- `Abstract` is used to declare that this enumeration cannot be instantiated, but must always be specialized.
- `Leaf` indicates that this enumeration can have no further sub-types, while `Root` indicates it is a top level enumeration.

`Visibility`

Radio box, with entries `public`, `private`, `protected`, and `package`.

Records the visibility for the Enumeration.

`Client Dependencies`
Text area. Lists any elements that depend on this enumeration. Button 1 double click navigates to the selected modelelement. Button 2 click brings up a pop up menu with following entry.

- `Add...`. This brings up a dialog box that allows to create dependencies from other modelelements.

`Supplier Dependencies`
Text area. Lists any elements that this enumeration depends on. Button 1 double click navigates to the selected modelelement. Button 2 click brings up a pop up menu with the following entry.

- `Add...`. This brings up a dialog box that allows to create dependencies to other modelelements.

`Generalizations`
Text area. Lists any enumeration that *generalizes* this enumeration.

`Specializations`
Text box. Lists any specialized enumerations (i.e. for which this enumeration is a generalization.

`Operations`
Text area. Lists all the operations defined on this enumeration. Button 1 double click navigates to the selected operation. Button 2 click brings up a pop up menu with two entries.

- `Move Up`. Only available where there are two or more operations, and the operation selected is not at the top. It is moved up one.

- `Move Down`. Only available where there are two or more operations listed, and the operation selected is not at the bottom. It is moved down one.

See Section 18.8, “Operation” for details of operations.

Caution

ArgoUML treats all operations as equivalent. Any operations created here will use the same mechanism as operations for classes. Remember that operations on enumerations must have no side effects (they are read-only). This means the `query` modifier *must* be checked for all operations.

`Literals`

Text area. Lists all the enumeration literals defined for this enumeration. Button 1 double click navigates to the selected literal, button 2 click brings up a pop up menu with two entries.

- `Move Up`. Only available where there are two or more literals, and the literal selected is not at the top. It is moved up one.

- `Move Down`. Only available where there are two or more literals listed, and the literal selected is not at the bottom. It is moved down one.

16.5. Enumeration Literal

An enumeration literal is one of the predefined values of an Enumeration.

16.6. Stereotype

Stereotypes are the main extension mechanism of UML, providing a way to derive specializations of the standard metaclasses. `Stereotype` is a sub-class of `GeneralizableElement` in the UML metamodel. Stereotypes are supplemented by *constraints* and *tagged values*.

New stereotypes are added from the property tab of almost any model element. Properties of existing stereotypes can be reached by selecting the property tab for any model element with that stereotype and using the navigate button () within the property tab.

16.6.1. Stereotype Details Tabs

The details tabs that are active for stereotypes are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 16.6.2, “Stereotype Property Toolbar” and Section 16.6.3, “Property Fields For Stereotype” below.

`Documentation`
: Standard tab. See Section 13.4, “Documentation Tab”.

`Stereotype`
: Standard tab.

Warning

Here you can set stereotypes of stereotypes, not a very usefull thing to do.

`Tagged Values`
: Standard tab. In the UML metamodel, `Stereotype` has the following standard tagged values defined.

  - `derived` (from the superclass, `ModelElement`). Values `true`, meaning the class is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

This indicates any element with this stereotype has the `derived` tag set accordingly.

Caution

Tagged values for a stereotype are rather different to those for elements in the UML core architecture, in that they apply to all model elements to which the stereotype is applied, *not* just the stereotype itself.

16.6.2. Stereotype Property Toolbar

 `Go up`

Navigate up through the package structure of the model.

 `Add stereotype`

This creates a new stereotype (see Section 16.6, “Stereotype”) within the model (which appears on no diagram), navigating immediately to the properties tab for that stereotype.

 `New Tag Definition`

This creates a new tag definition (see Section 16.7, “Tag Definition”) within the model (which appears on no diagram), navigating immediately to the properties tab for that tagdefinition.

 `Delete`

This deletes the stereotype from the model.

16.6.3. Property Fields For Stereotype

`Name`

Text box. The name of the stereotype. There is no convention for naming stereotypes, beyond starting them with a lower case letter. Even the standard UML stereotypes vary between all lower case (e.g. `metamodel`), bumpy caps (e.g. `systemModel`) and space separated (e.g. `object model`).

Note

ArgoUML does not enforce any naming convention for stereotypes

`Base Class`

Drop down selector. Any stereotype must be derived from one of the metaclasses in the UML metamodel or the model element classes that derive from them. The stereotype will then be available to model elements that derive from that same metaclass or that model element.

`Namespace`

Drop down selector with navigation button. Records the namespace for the stereotype. This is the package hierarchy.

`Modifiers`

Check box, with entries `Abstract`, `Leaf` and `Root`.

- `Abstract` is used to declare that model elements that use this stereotype cannot be instantiated, but must always be specialized.
- `Leaf` indicates that model elements that use this stereotype can have no further sub-types, while `Root` indicates it is a top level model element.

Caution

Remember that these modifiers apply to the model elements using the stereotype, not just the stereotype.

Warning

ArgoUML neither imposes, nor checks that model elements using a stereotype adopt the stereotype's modifiers.

`Visibility`

Radio box, with entries `public`, `private`, `protected`, and `package`.

Records the visibility for the stereotype.

`Generalizations`
Text area. Lists any stereotype that *generalizes* this stereotype.

`Specializations`
Text area. Lists any specialized stereotype (i.e. for which this stereotype is a generalization.

`Tag Definitions`
Text area. Lists any tag definitions that are defined for this stereotype.

`Extended Elements`
Text area. Lists all modelelements that are stereotyped by this stereotype.

16.7. Tag Definition

(To Be Written)

16.8. Diagram

The UML standard specifies eight principal diagrams, all of which are supported by ArgoUML.

- *Use case diagram*. Used to capture and analyse the requirements for any OOA&D project. See Chapter 17, *Use Case Diagram Model Element Reference* for details of the ArgoUML use case diagram and the model elements it supports.
- *Class diagram*. This diagram captures the static structure of the system being designed, showing the classes, interfaces and datatypes and how they are related. Variants of this diagram are used to show package structures within a system (the *package diagram*) and the relationships between particular instances (the *object diagram*).

  The ArgoUML class diagram provides support for class and package diagrams. See Chapter 18, *Class Diagram Model Element Reference* for details of the model elements it supports. The object diagram is suported on the Deployment diagram.
- *Behavior diagrams*. There are four such diagrams (or strictly speaking, five, since the use case diagram is a type of behavior diagram), which show the dynamic behavior of the system at all levels.
  - *Statechart diagram*. Used to show the dynamic behavior of a single object (class instance). This

diagram is of particular use in systems using complex communication protocols, such as in telecommunications. See Chapter 20, *Statechart Diagram Model Element Reference* for details of the ArgoUML statechart diagram and the model elements it supports.

- *Activity diagram*. Used to show the dynamic behavior of groups of objects (class instance). This diagram is an alternative to the statechart diagram, and is better suited to systems with a great deal of user interaction. See Chapter 22, *Activity Diagram Model Element Reference* for details of the ArgoUML activity diagram and the model elements it supports.

- *Interaction diagrams*. There are two diagrams in this category, used to show the dynamic interaction between objects (class instances) in the system.

  - *Sequence diagram*. Shows the interactions (typically messages or procedure calls) between instances of classes (objects) and actors against a timeline. Particularly useful where the timing relationships between interactions are important. See Chapter 19, *Sequence Diagram Model Element Reference* for details of the ArgoUML sequence diagram and the model elements it supports.

  - *Collaboration diagram*. Shows the interactions (typically messages or procedure calls) between instances of classes (objects) and actors against the structural relationships between those instances. Particularly suitable where it is useful to relate interactions to the static structure of the system. See Chapter 21, *Collaboration Diagram Model Element Reference* for details of the ArgoUML collaboration diagram and the model elements it supports.

- *Implementation diagrams*. UML defines two implementation diagrams to show the relationship between the software components that make up a system (the *component diagram*) and the relationship between the software and the hardware on which it is deployed at run-time (the *deployment diagram*.

  The ArgoUML deployment diagram provides support for both component and deployment diagrams, and additionally for object diagrams. See Chapter 23, *Deployment Diagram Model Element Reference* for details of the diagram and the model elements it supports.

Diagrams are created using the `Create` drop down menu (see Section 10.6, “The Create Menu”), or with the tools on the toolbar (see Section 9.4, “Create operations”), or with the pop-up menus in the explorer.

Note

ArgoUML uses its deployment diagram to create the UML 1.4 component, deployment and object diagrams.

Caution

Statechart and activity diagrams are associated with a particular class or operation (or the latter also with a package), and can only be created when this modelelement has been selected.

Warning

In ArgoUML version 0.24, the UML 1.4 object diagram as a variant of the class diagram is not directly supported. However, it is possible to create simple object diagrams within the ArgoUML deployment diagram.

16.8.1. Diagram Details Tabs

The details tabs that are active for diagrams are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 16.8.3, “Property Fields For Diagram” below.

16.8.2. Diagram Property Toolbar

 `Go up`

Navigate up through the package structure of the model.

 `Delete`

This deletes the diagram from the model. As a consequence, in case of a statechart diagram or an activity diagram, all contained elements are deleted, too.

16.8.3. Property Fields For Diagram

`Name`

The name of the diagram. There are no conventions for naming diagrams. By default, ArgoUML uses the (space separated) diagram name and a sequence number, thus `Use Case Diagram 1`.

Tip

This name is used to generate a filename when activating the “Save Graphics...” menu-item.

`Home Model`

The Home Model of the diagram is not something defined in the UML specification. The Home Model is the modelelement represented by the diagram. Hence its type depends on the type of diagram: e.g. it is the namespace represented by a class diagram, or the statemachine in case of a Statechart diagram.

Chapter 17. Use Case Diagram Model Element Reference

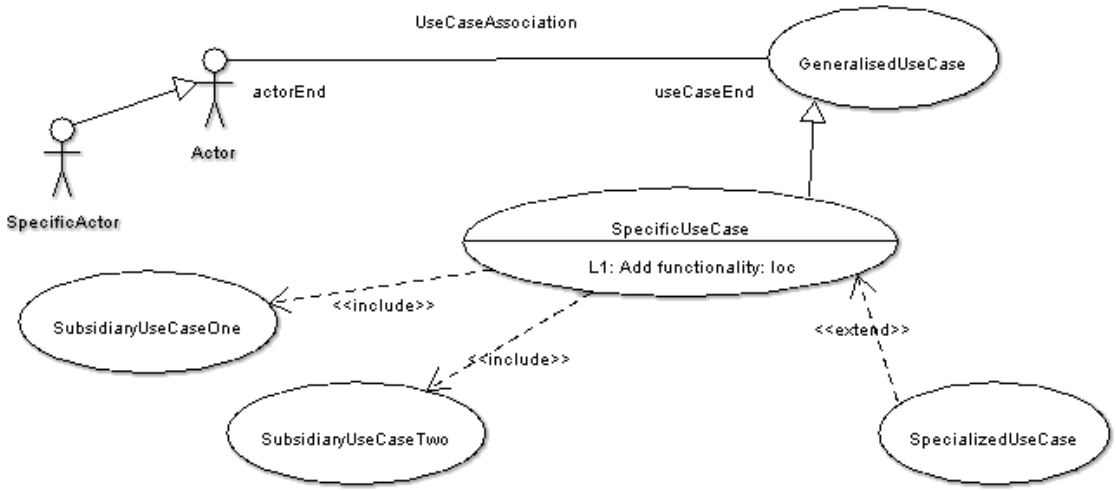
17.1. Introduction

This chapter describes each model element that can be created within a use case diagram. Note that some sub-model elements of model elements on the diagram may not actually themselves appear on the diagram.

There is a close relationship between this material and the properties tab of the details pane (see Section 13.3, “Properties Tab”). That section covers properties in general, in this chapter they are linked to specific model elements.

Figure 17.1, “Typical model elements on a use case diagram.” shows a use case diagram with all typical model elements displayed.

Figure 17.1. Typical model elements on a use case diagram.



17.1.1. ArgoUML Limitations Concerning Use Case Diagrams

Use case diagrams are now well supported within ArgoUML. There still are some minor limitations though, especially with extension points.

Note

Earlier versions of ArgoUML (0.9 and earlier) implemented extend and include relationships by using a stereotyped dependency relationship. Although such diagrams will show correctly on the diagram, they will not link correctly to the use cases, and should be replaced by proper extend and include relationships using the current system.

17.2. Actor

An actor represents any external entity (human or machine) that interacts with the system, providing input, receiving output, or both.

Within the UML metamodel, actor is a sub-class of `classifier`.

The actor is represented by a “stick man” figure on the diagram (see Figure 17.1, “Typical model elements on a use case diagram.”).

17.2.1. Actor Details Tabs

The details tabs that are active for actors are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 17.2.2, “Actor Property Toolbar” and Section 17.2.3, “Property Fields For Actor” below.

`Documentation`
: Standard tab. See Section 13.4, “Documentation Tab”.

`Presentation`
: Standard tab. The fill color is used for the stick man's head.

`Source`
: Standard tab. Usually, no code is provided for an actor, since it is external to the system.

`Stereotype`
: Standard tab.

`Tagged Values`
: Standard tab. In the UML metamodel, `Actor` has the following standard tagged values defined.

- `persistence` (from the superclass, `Classifier`). Values `transitory`, indicating state is destroyed when an instance is destroyed or `persistent`, marking state is preserved when an instance is destroyed.

  Tip

  Actors sit outside the system, and so their internal behavior is of little concern, and this tagged value is best ignored.

- `semantics` (from the superclass, `Classifier`). The value is a specification of the semantics of the actor.

- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the actor is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

  Note

  Derived actors have limited value, since they sit outside the system being designed. They may have their value in analysis to introduce useful names or concepts.

`Checklist`
Standard tab for a Classifier.

17.2.2. Actor Property Toolbar

 `Go up`

Navigate up through the package structure of the model.

 `Add Actor`

This creates a new actor within the model, (but not within the diagram), navigating immediately to the properties tab for that actor.

Tip

This method of creating a new actor may be confusing. It is much better to create an actor on the diagram.

 `New Reception`

This creates a new reception within the model,(but not within the diagram), navigating immediately to the properties tab for that rception.

Tip

A reception is a declaration that the actor handles a signal, but the actual handling is specified by a state machine.

 `Delete`

This deletes the selected actor from the model.

Warning

This is a deletion from the model *not* just the diagram. To delete an actor from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

17.2.3. Property Fields For Actor

`Name`

Text box. The name of the actor. The diagram shows this name below the stick man figure. Since an actor is a classifier, it would be conventional to Capitalize the first letter (and initial letters of any component words), e.g. `RemoteSensor`.

Note

ArgoUML does not enforce any naming convention for actors

`Namespace`

Text box with navigation button. Records the namespace for the actor. This is the package hierarchy.

`Modifiers`

Check box, with entries `Abstract`, `Leaf` and `Root`.

- `Abstract` is used to declare that this actor cannot be instantiated, but must always be specialized.

Caution

While actors can be specialized and generalized, it is not clear that an abstract actor has any meaning. Perhaps it might be used to indicate an actor that does not itself interact with a use case, but whose children do.

- `leaf` indicates that this actor can have no further children, while `Root` indicates it is a top level actor with no parent.

`Generalizations`

Text area. Lists any actor that *generalizes* this actor.

Button 1 double click navigates to the generalization and opens its property tab.

`Specializations`

Text box. Lists any specialized actor (i.e. for which this actor is a generalization. The specialized actors can communicate with the same use case instances as this actor.

Button 1 double click navigates to the generalization and opens its property tab.

`Association Ends`

Text area. Lists any association ends of associations connected to this actor.

Button 1 double click navigates to the selected entry.

17.3. Use Case

A use case represents a complete meaningful “chunk” of activity by the system in relation to its external users (actors), human or machine. It represents the primary route through which requirements are captured for the system under construction

Within the UML metamodel, use case is a sub-class of `classifier`.

The use case icon is an oval (see Figure 17.1, “Typical model elements on a use case diagram.”). It may be split in two, with the lower compartment showing extension points

Caution

By default ArgoUML does not show the extension point compartment. It may be revealed by the context sensitive `Show` menu (using button 2 click), or from the Presentation tab.

17.3.1. Use Case Details Tabs

The details tabs that are active for use cases are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 17.3.2, “Use Case Property Toolbar” and Section 17.3.3, “Property Fields For Use Case” below.

`Documentation`
Standard tab. See Section 13.4, “Documentation Tab”.

`Presentation`
Standard tab. The Fill color is used for the use case oval.

The `Display: Extension Points` check box is used to control whether an extension point compartment is displayed.

`Source`
Standard tab. It would not be usual to provide any code for a use case, since it is primarily a vehicle for capturing requirements about the system under construction, not creating the solution.

`Stereotype`
Standard tab.

`Tagged Values`
Standard tab. In the UML metamodel, `UseCase` has the following standard tagged values defined.

- `persistence` (from the superclass, `Classifier`). Values `transitory`, indicating state is destroyed when an instance is destroyed or `persistent`, marking state is preserved when an instance is destroyed.

  ### Tip

  In general the instantiation of use cases is not a major aspect of any design method (they are mostly concerned with requirements capture. For most OOA&D methodologies, this tag can safely be ignored.

- `semantics` (from the superclass, `Classifier`). The value is a specification of the semantics of the use case.

- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the use case is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

  ### Note

  Derived use cases still have their value in analysis to introduce useful names or

concepts.

`Checklist`
Standard tab for a Classifier.

17.3.2. Use Case Property Toolbar


`Go up`

Navigate up through the package structure of the model.


`New use case`

This creates a new use case within the model, (but not within the diagram), and shows immediately the properties tab for that use case.

Tip

This method of creating a new use case can be confusing. It is much better to create a new use case on the diagram of your choice.


`New extension point`

This creates a new use extension point within the namespace of the current use case, with the current use case as its associated use case, navigating immediately to the properties tab for that extension point.


`New Attribute`

This creates a new attribute within the current use case, navigating immediately to the properties tab for that attribute.


`New Operation`

This creates a new operation within the current use case, navigating immediately to the properties tab for that operation.


`New Reception`

This creates a new reception within the current use case, navigating immediately to the properties tab for that reception.


`New Stereotype`

This creates a new stereotype within the current use case, navigating immediately to the properties tab for that stereotype.


`Delete`

This deletes the selected use case from the model.

Warning

This is a deletion from the model *not* just the diagram. To delete a use case from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

17.3.3. Property Fields For Use Case

`Name`

Text box. The name of the use case. Since a use case is a classifier, it would be conventional to Capitalize the first letter (and initial letters of any component words), e.g. `RemoteSensor`. The name is shown inside the oval representation of the use case on the diagram.

Note

ArgoUML does not enforce any naming convention for use cases

`Namespace`

Text box with navigation button. Records the namespace for the use case. This is the package hierarchy.

`Modifiers`

Check box, with entries `Abstract` `Leaf` and `Root`.

- `Abstract` is used to declare that this actor cannot be instantiated, but must always be specialized. .
- `Leaf` indicates that this use case can have no further children, while `Root` indicates it is a top level use case with no parent.

`Client Dependencies`

Text area. Lists the “depending” ends of the relationship, i.e. the end that makes use of the other end.

Button 1 double click navigates to the dependency and opens its property tab.

Button 2 click shows a pop-up menu with one entry `Add...` that opens a dialog box where you can add and remove depending modelelements.

`Supplier Dependencies`

Text area. Lists the “supplying” ends of the relationship, i.e. the end supplying what is needed by the other end.

Button 1 double click navigates to the dependency and opens its property tab.

Button 2 click shows a pop-up menu with one entry `Add...` that opens a dialog box where you can add and remove dependent modelelements.

`Generalizations`

Text area. Lists use cases which are generalizations of this one. Will be set whenever a generaliza-

tion is created on the from this Use Case. Button 1 Double Click on a generalization will navigate to that generalization.

`Specializations`

Text box. Lists any specialized use case (i.e. for which this use case is a generalization.

Button 1 double click navigates to the generalization and opens its property tab.

`Extends`

Text box. Lists any class that is extended by this use case.

Where an extends relationship has been created, button 1 double click will navigate to that relationship.

`Includes`

Text box. Lists any use case that this use case includes.

Where an include relationship has been created, button 1 Double Click will navigate to that relationship.

`Attributes`

Text area. Lists all the attributes (see Section 18.7, “Attribute”) defined for this use case. Button 1 double click navigates to the selected attribute. Button 2 gives a pop up menu with two entries, which allow reordering the attributes.

- `Move Up`. Only available where there are two or more attributes listed, and the attribute selected is not at the top. It moves the attribute up one position.
- `Move Down`. Only available where there are two or more attributes listed, and the attribute selected is not at the bottom. It moves the attribute down one position.

`Association Ends`

Text box. Lists any association ends (see Section 18.12, “Association”) of associations connected to this use case.

Button 1 double click navigates to the selected entry.

`Operations`

Text area. Lists all the operations (see Section 18.8, “Operation”) defined on this use case. Button 1 click navigates to the selected operation. Button 2 gives a pop up menu with two entries, which allow reordering the operations.

- `Move Up`. Only available where there are two or more operations listed, and the operation selected is not at the top. It moves the operation up one position.
- `Move Down`. Only available where there are two or more operations listed, and the operation selected is not at the bottom. It moves the operation down one position.

`Extension Points`

Text box. If this use case is, or can be extended, this field lists the extension points for the use case.

Note

Extension points are listed by their location point rather than their name.

Where an extension point has been created (see below), button 1 Double Click will navigate to that relationship. Button 2 gives a pop up menu with the following entries.

- `New`. Add a new extension point and navigate to it, making this use case the owning use case of the extension point.
- `Move Up`. Only available where there are two or more extension points listed, and the extension point selected is not at the top. It moves the extension point up one position.
- `Move Down`. Only available where there are two or more extension points listed, and the extension point selected is not at the bottom. It moves the extension point down one position.

17.4. Extension Point

An extension point describes a point in a use case where an extending use case may provide additional behavior.

Examples for a travel agent sales system might be the use case for paying for a ticket, which has an extension point in the specification of the payment. Extending use cases may then extend at this point to pay by cash, credit card etc.

Within the UML metamodel, `Extension Point` is a sub-class of `ModelElement`. A use case may display an extension point compartment (see Section 17.3, “Use Case” for details), in which extension points are shown with the following syntax.

name : *location*.

17.4.1. Extension Point Details Tabs

The details tabs that are active for extension points are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 17.4.2, “Extension Point Property Toolbar” and Section 17.4.3, “Property Fields For Extension Point” below.

`Documentation`
Standard tab. See Section 13.4, “Documentation Tab”.

`Stereotype`
Standard tab.

Extensionpoints do not have any stereotypes defined by default.

`Tagged Values`
Standard tab. In the UML metamodel, `ExtensionPoint` has the following standard tagged values defined.

- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the extension point is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

It is not clear how derived extension points could have any value in analysis.

17.4.2. Extension Point Property Toolbar

`Go up`

Navigate up to the use case which owns this extension point.

`New Extension Point`

This creates a new Extension Point below the selected extension point, navigating immediately to the properties tab of the newly created extension point.

`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected extension point, navigating immediately to the properties tab for that stereotype.

`Delete`

This deletes the selected extension point from the model.

17.4.3. Property Fields For Extension Point

`Name`

Text box. The name of the extension point.

Tip

It is quite common to leave extension points unnamed in use case analysis, since they are always listed (within use cases and extend relationships) by their location.

Note

ArgoUML does not enforce any naming convention for extension points.

`Location`

Text box. A description of the location of this extension point within the owning use case.

Tip

Extension points are always listed (within use cases and extend relationships) by their location. Typically this will be the number/name of the paragraph in the specification.

`Base Use Case`

Text box. Shows the base use case within which this extension point is defined. Button 1 Double Click will navigate to the use case.

`Extend`

Text box. Lists all use cases which extend the base use case through this extension point.

Where an extending use case exists, button 1 double click will navigate to that relationship.

17.5. Association

An association on a use case diagram represents a relationship between an actor and a use case showing that actor's involvement in the use case. The invocation of the use case will involve some (significant) change perceived by the actor.

Associations are described fully under class diagrams (see Section 18.12, “Association”).

17.6. Association End

Association ends are described under class diagrams (see Section 18.13, “Association End”).

17.7. Dependency

Dependencies are described under class diagrams (see Section 18.14, “Dependency”).

Caution

Dependency has little use in use case diagrams. It is provided, because earlier versions of ArgoUML used it (incorrectly) to implement include and extends relationships.

17.8. Generalization

Generalization is a relationship between two use cases or two actors. Where A is a generalization of B, it means A describes more general behavior and B a more specific version of that behavior.

Examples for a travel agent sales system might be the use case for making a booking as a generalization of the use case for making a flight booking and a salesman actor being a generalization of a supervisor actor (since supervisors can also act as salesmen, but not vice versa).

Generalization is analogous to class inheritance within OO programming.

Note

It is easy to confuse *extends* relationships between use cases with generalization. However extends is about augmenting a use case's behavior at a specific point. Generalization is about specializing the behavior throughout the use case.

Within the UML metamodel, `Generalization` is a sub-class of `Relationship`.

Generalization is represented as an arrow with white filled head from the specialized use case or actor to the generalized use case or actor (see Figure 17.1, “Typical model elements on a use case diagram.”).

17.8.1. Generalization Details Tabs

The details tabs that are active for associations are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 17.8.2, “Generalization Property Toolbar” and Section 17.8.3, “Property Fields For Generalization” below.

`Documentation`
Standard tab. See Section 13.4, “Documentation Tab”.

`Presentation`
Standard tab.

Note

The values for the bounds of the generalization are downlighted, since they have no meaning, given that the generalization is tied to a particular couple of actors or use cases.

`Stereotype`
Standard tab.

`Tagged Values`
Standard tab. In the UML metamodel, `Generalization` has the following standard tagged values defined.

- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the generalization is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived generalizations still have their value in analysis to introduce useful names or concepts, and in design to avoid re-computation.

17.8.2. Generalization Property Toolbar

 `Go up`

Navigate up through the package structure of the model. For a generalization this will be the package containing the generalization.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected generalization, navigating immediately to the properties tab for that generalization.


`Delete`

This deletes the selected generalization from the model.

Warning

This is a deletion from the model *not* just the diagram. To delete a generalization from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

17.8.3. Property Fields For Generalization

`Name`

Text box. The name of the generalization.

Tip

It is quite common to leave generalizations unnamed in use case analysis.

Note

ArgoUML does not enforce any naming convention for associations.

Note

There is no representation of the name of a generalization on the diagram.

`Discriminator`

Text box. The name of a discriminator for the specialization. UML 1.4 allows grouping of specializations into a number of sets, on the basis of this value.

Tip

The empty string “” is a valid entry (and the default) for this field. The discriminator is only of practical use in cases of multiple inheritance. A (class diagram) example is shown in Figure 17.2, “Example use of a discriminator with generalization”. Here each type of user should inherit from two sorts of user. One distinguishing between local or remote user (which can be identified by one discriminator) and one indicating their function as a user (identified by a different discriminator).

There is little point in using this within a use case diagram.

`Namespace`

Text box with navigation button. Records the namespace for the generalization. This is the package hierarchy.

`Parent`

Text box. Shows the use case or actor that is the *parent* in this relationship, i.e. the more general end of the relationship. Button 1 Double Click on this entry will navigate to that use case or actor.

`Child`

Text box. Shows the use case or actor that is the *child* in this relationship, i.e. the more specific end of the relationship. Button 1 Double Click on this entry will navigate to that use case or actor.

`Powertype`

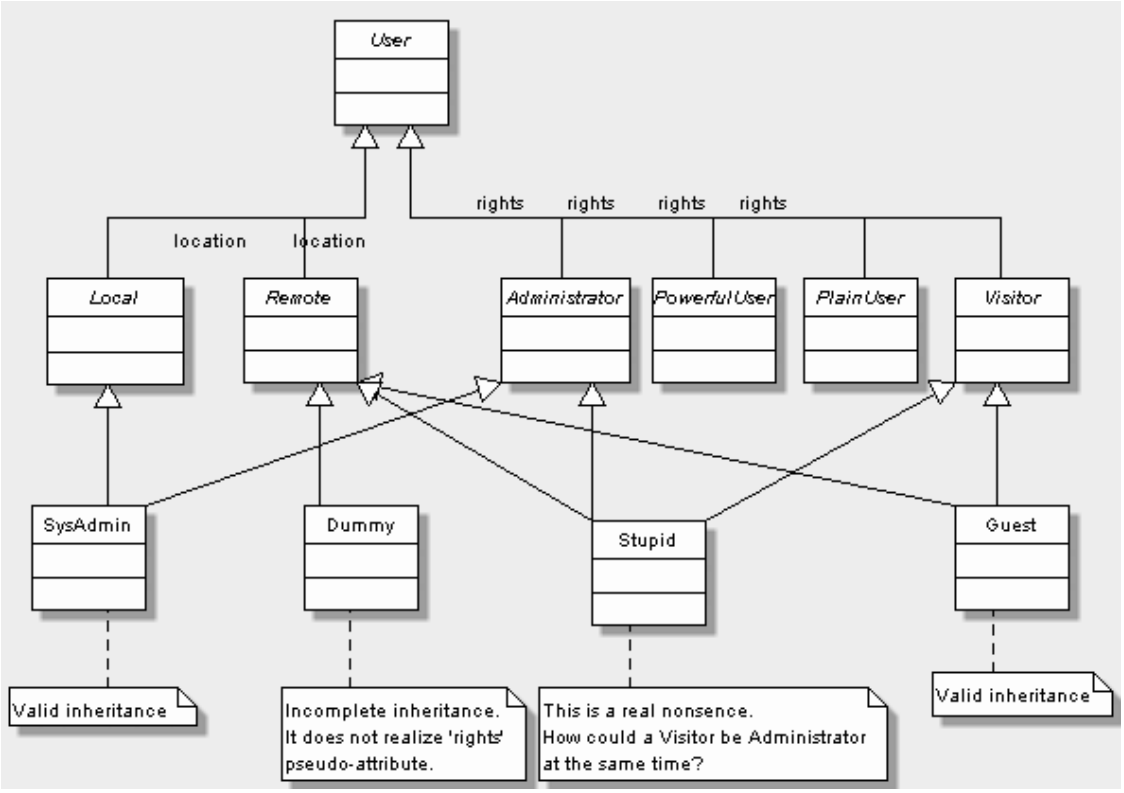
Drop down selector providing access to all standard UML types provided by ArgoUML and all new classes created within the current model.

This is the type of the child entity of the generalization.

Tip

This can be ignored for use case analysis. The only sensible value to put in would be the child use case type (as a classifier, this appears in the drop down list.

Figure 17.2. Example use of a discriminator with generalization



17.9. Extend

Extend is a relationship between two use cases. Where A extends B, it means A describes some addi-

tional behavior that is executed conditionally (under exceptional circumstances) at some point during the normal behavior of B.

In some respects extend is like generalization. However the key difference is that the extended use case defines *extension points* (see Section 17.4, “Extension Point”), which are the only places where its behavior may be extended. The extending use case must define at which of these extension points it adds behavior.

This makes the use of extend more tightly controlled than general extension, and it is thus preferred wherever possible.

Examples for a travel agent sales system might be the use case for paying for a ticket, which has an extension point in the specification of the payment. Extending use cases may then extend at this point to pay by cash, credit card etc.

Within the UML metamodel, `Extend` is a sub-class of `Relationship`.

An extend relationship is represented as a dotted link with an open arrow head and a label «`extend`». If a condition is defined, it is shown under the «`extend`» label (see Figure 17.1, “Typical model elements on a use case diagram.”).

17.9.1. Extend Details Tabs

The details tabs that are active for extend relationships are as follows.

Note

There is no source tab, since there is no source code that could be generated for an extend relationship.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 17.9.2, “Extend Property Toolbar” and Section 17.9.3, “Property Fields For Extend” below.

`Documentation`
: Standard tab. See Section 13.4, “Documentation Tab”.

`Presentation`
: Standard tab

Note

The values for the bounds are downlighted, since the extend is tied to a particular pair of use cases.

`Stereotype`
: Standard tab.

`Tagged Values`
: Standard tab. In the UML metamodel, `Extend` has the following standard tagged values defined.

  - `derived` (from the superclass, `ModelElement`). Values `true`, meaning the extend relation-

ship is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived extend relationships could have their value in analysis to introduce useful names or concepts.

17.9.2. Extend Property Toolbar

 `Go up`

Navigate up through the package structure of the model. For a extend this will be the package containing the extend.

 `New extension point`

This creates a new use case extension point within the namespace of the current extend relationship, with the current extend relationship as its first extending relationship.

Tip

While it is perfectly valid to create extension points from an extend relationship, the created extension point will have no associated use case (it can subsequently be set up).

It would be more usual to instead create the extension point within a use case and subsequently link to it from an extend relationship (see Section 17.9.3, “Property Fields For Extend” below).

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected extent relationship, navigating immediately to the properties tab for that stereotype.

 `Delete`

This deletes the selected extend relationship from the model.

Warning

This is a deletion from the model *not* just the diagram. To delete a extend from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

17.9.3. Property Fields For Extend

`Name`

Text box. The name of the extend relationship.

Tip

It is quite common to leave extends unnamed in use case analysis.

Note

ArgoUML does not enforce any naming convention for extend relationships.

`Namespace`

Text box. Records the namespace for the extend relationship. This is the package hierarchy.

Button 1 Double Click on the entry will navigate to the package defining this namespace (or the model for the top level namespace).

`Base Use Case`

Text box. Shows the use case that is being extended by this extend relationship. Button 1 double click on this entry will navigate to the base use case.

`Extension`

Text box. Show the use case that is doing the extending through this extend relationship. Button 1 double click on this entry will navigate to the extension use case.

`Extension Points`

Text box. Lists the extension points of the base use case where the extension will be applied if the condition holds.

Note

If the condition is fulfilled, the sequence obeyed by the use-case instance is extended to include the sequence of the extending use case. The different parts of the extending use case are inserted at the locations defined by the sequence of extension points in the relationship -- one part at each referenced extension point. Note that the condition is only evaluated once: at the first referenced extension point, and if it is fulfilled all of the extending use case is inserted in the original sequence.

Hence, the sequence of the extension points is irrelevant, except for the position of the first one; since that one determines where the condition is evaluated.

Where an extension point has been created, button 1 double click will navigate to that relationship. Button 2 gives a pop up menu with the following entries.

- `Add`. The “Ad/Remove ExtensionPoints” window opens. In this window it is possible to build a list of extension points.
- `New`. Add a new extension point in the list and navigate to it. The current extend relationship is added as the first in list of extending relationships of the new extension point.
- `Move Up`. Only available where there are two or more extension points listed, and the extension point selected is not at the top. It moves the extension point up one position.

- `Move Down`. Only available where there are two or more extension points listed, and the extension point selected is not at the bottom. It moves the extension point down one position.

`Condition`
Text area. Multi-line textual description of any condition attached to the extend relationship.

The text entered here is shown on the diagram.

17.10. Include

Include is a relationship between two use cases. Where A includes B, it means B described behavior that is to be included in the description of the behavior of A at some point (defined internally by A).

Examples for a travel agent sales system might be the use case for booking travel, which includes use cases for booking flights and taking payment.

Within the UML metamodel, `Include` is a sub-class of `Relationship`.

An include relationship is represented as a dotted link with an open arrow head and a label «`include`» (see Figure 17.1, “Typical model elements on a use case diagram.”).

17.10.1. Include Details Tabs

The details tabs that are active for include relationships are as follows.

Note

There is no source tab, since there is no source code that could be generated for an include relationship.

`ToDoItem`
Standard tab.

`Properties`
See Section 17.10.2, “Include Property Toolbar” and Section 17.10.3, “Property Fields For Include” below.

`Documentation`
Standard tab. See Section 13.4, “Documentation Tab”.

`Presentation`
Standard tab

Note

The values for the bounds of the include relationships are downlighted, since the include relationship is represented by a line between a particular pair of use cases.

`Tagged Values`
Standard tab. In the UML metamodel, `Include` has the following standard tagged values defined.

- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the include relationship is redundant—it can be formally derived from other elements, or `false` meaning it

cannot.

Note

Derived include relationships could have their value in analysis to introduce useful names or concepts.

17.10.2. Include Property Toolbar

 `Go up`

Navigate up through the package structure of the model. For a include this will be the package containing the include.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected include relationship, navigating immediately to the properties tab for that stereotype.

 `Delete`

This deletes the selected include relationship from the model.

Warning

This is a deletion from the model *not* just the diagram. To delete a include from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

17.10.3. Property Fields For Include

`Name`

Text box. The name of the include relationship.

Tip

It is quite common to leave include relationships unnamed in use case analysis.

Note

ArgoUML does not enforce any naming convention for include relationships.

`Namespace`

Text box. Records the namespace for the include. This is the package hierarchy.

Button 1 click on the entry will navigate to the package defining this namespace (or the model for the top level namespace).

`Base Use Case`

Drop down selector. Records the use case that is doing the including in this include relationship. Button 1 click on this entry will give a drop down menu of all available use cases which may be selected by button 1 click.

`Included Use Case`

Drop down selector. Records the use case that is being included by this include relationship. Button 1 click on this entry will give a drop down menu of all available use cases (and an empty entry) which may be selected by button 1 click.

Chapter 18. Class Diagram Model Element Reference

18.1. Introduction

This chapter describes each model element that can be created within a class diagram. Note that some sub-model elements of model elements on the diagram may not actually themselves appear on the diagram.

Class diagrams are used for only one of the UML static structure diagrams, the class diagram itself. Object diagrams are represented on the ArgoUML deployment diagram.

In addition, ArgoUML uses the class diagram to show model structure through the use of packages.

There is a close relationship between this material and the Properties Tab of the Details Pane (see Section 13.3, “Properties Tab”). That section covers Properties in general, in this chapter they are linked to specific model elements.

Figure 18.1, “Possible model elements on a class diagram.” shows a class diagram with all possible model elements displayed.

Figure 18.1. Possible model elements on a class diagram.

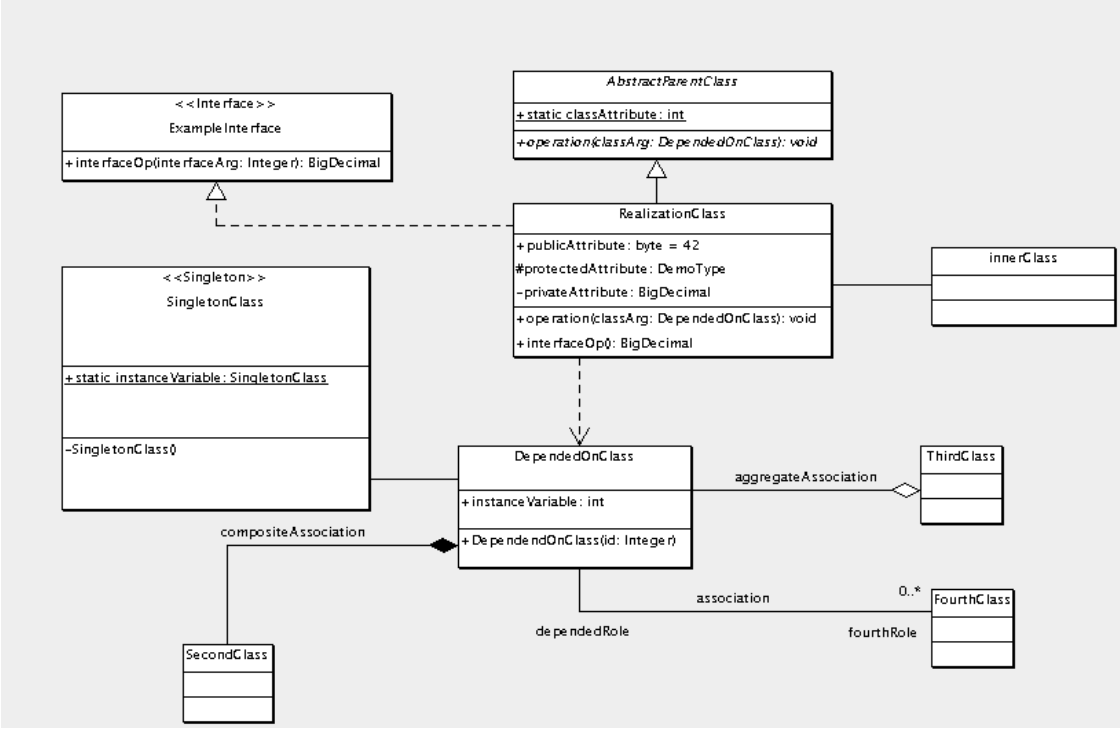


Figure 18.2, “Possible model elements on a package diagram.” shows a package diagram with all possible model elements displayed.

Figure 18.2. Possible model elements on a package diagram.

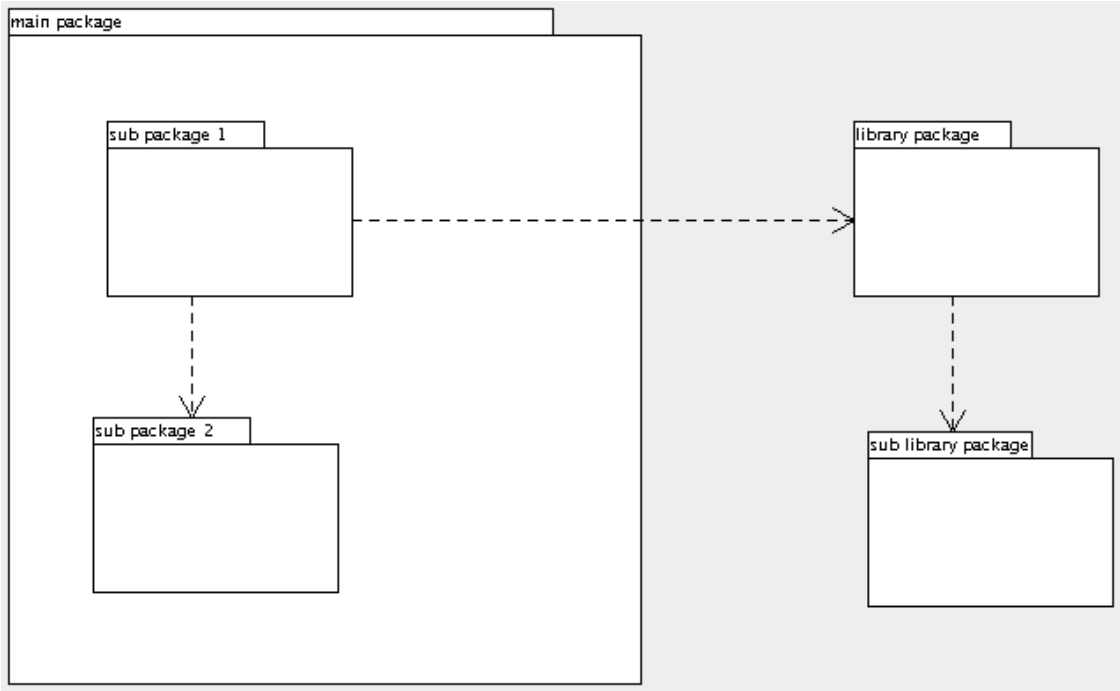


Figure 18.3, “Possible model elements on a datatype diagram.” shows a datatype diagram with a datatype and an enumeration displayed.

Figure 18.3. Possible model elements on a datatype diagram.

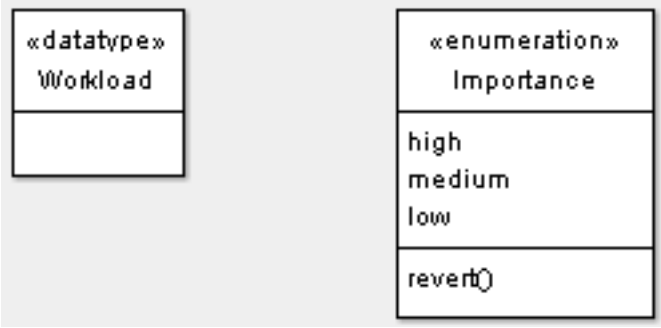
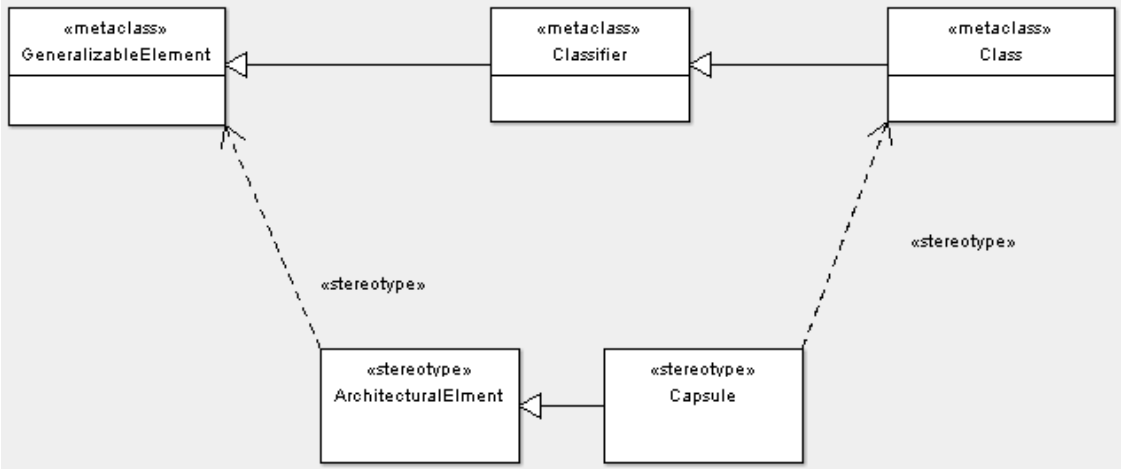


Figure 18.4, “Possible model elements on a stereotype definition diagram.” shows a stereotype definition diagram with all possible model elements displayed.

Figure 18.4. Possible model elements on a stereotype definition diagram.



18.1.1. Limitations Concerning Class Diagrams in ArgoUML

Various limitations exist in V0.24 of ArgoUML for stereotype definition diagrams. E.g. the current implementation does not allow stereotype compartments to be shown on stereotype definition diagrams.

Another variant of the class diagram within the UML standard is the *object diagram*. There is currently no support for objects or links within ArgoUML class diagrams. Instead the ArgoUML deployment diagram does have both objects and links, and can be used to draw object diagrams.

18.2. Package

The package is the main organizational model element within ArgoUML. In the UML metamodel it is a sub-class of both `Namespace` and `GeneralizableElement`.

Note

ArgoUML also implements the UML `Model` model element as a sub-class of package, but *not* the `Subsystem` model element.

ArgoUML also implements some less common aspects of UML model management. In particular the relationship UML 1.4 defines as `Generalization` and the sub-class dependency `Permission` for use between packages.

18.2.1. Package Details Tabs

The details tabs that are active for packages are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 18.2.2, “Package Property Toolbar” and Section 18.2.3, “Property Fields For Package” below.

`Documentation`

Standard tab. See Section 13.4, “Documentation Tab”.

`Presentation`
Standard tab. The editable `Bounds` field defines the bounding box for the package on the diagram.

`Stereotype`
Standard tab.

`Tagged Values`
Standard tab. In the UML metamodel, `Package` has the following standard tagged values defined.

- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the package is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived packages still have their value in analysis to introduce useful names or concepts, and in design to avoid re-computation.

18.2.2. Package Property Toolbar


`Go up`

Navigate up through the package structure.


`New Package`

This creates a new package within the package (which appears on no diagram), navigating immediately to the properties tab for that package.


`New Datatype`

This creates a new Datatype (see Section 16.3, “Datatype”) for the selected package, navigating immediately to the properties tab for that datatype.


`New Enumeration`

This creates a new Enumeration (see Section 16.4, “Enumeration”) for the selected package, navigating immediately to the properties tab for that enumeration.


`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected package, navigating immediately to the properties tab for that stereotype.


`New Tag Definition`

This creates a new tag definition (see Section 16.7, “Tag Definition”) within the package (which appears on no diagram), navigating immediately to the properties tab for that tagdefinition.


`Delete Package`

Deletes the package from the model.

Warning

This is a deletion from the model *not* just the diagram. To delete a package from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

18.2.3. Property Fields For Package

`Name`

Text box. The name of the package. The name of a package, like all packages, is by convention all lower case, not containing any punctuation marks.

Note

By default a new package has no name defined. The package will appear with the name `(Unnamed Package)` in the explorer.

`Namespace`

Drop down selector. Records the namespace for the package. This is the package hierarchy.

`Visibility`

Radio box, with four entries `public`, `private`, `protected`, and `package`. Indicates whether the package is visible outside the package.

`Modifiers`

Check box, with entries `abstract`, `leaf` and `root`.

- `Abstract` is used to declare that this package cannot be instantiated, but must always be specialized.

Tip

The meaning of `abstract` applied to a package if not that clear. It might mean that the package contains interfaces or abstract classes without realizations. This is probably better handled through stereotyping of the package (for example «`facade`»).

- `Leaf` indicates that this package can have no further subpackages.
- `Root` indicates that it is the top level package.

Tip

Within ArgoUML `Root` only meaningfully applies to the Model, since all packages sit within the model. This could be used to emphasize that the Model is at the top level.

`Generalizations`

Text area. Lists any package that *generalizes* this package.

Button 1 double click navigates to the generalization and opens its property tab.

`Specializations`

Text box. Lists any specialized package (i.e. for which this package is a generalization.

button 1 double click navigates to the generalization and opens its property tab.

`Owned Elements`

Text area. A listing of all the packages, classes, interfaces, datatypes, actors, use cases, associations, generalizations, stereotypes, etc. within the package.

Button 1 double click on any item listed here navigates to that model element.

`Imported Elements`

Text Area. A listing of all imported elements, i.e. elements that are owned by a different package, but are explicitely made visible in this package.

Button 1 double click on any item listed here navigates to that model element. Button 2 gives a pop up menu with the following entries.

- `Add`. The "Add/Remove Imported Elements" window opens. In this window it is possible to build a list of imported elements.
- `Remove`. Removes the import.

18.3. Datatype

Datatypes are not specific to packages or class diagrams, and are discussed within the chapter on top level model elements (see Section 16.3, "Datatype").

18.4. Enumeration

Enumeration are not specific to packages or class diagrams, and are discussed within the chapter on top level model elements (see Section 16.4, "Enumeration").

18.5. Stereotype

Stereotypes are not specific to packages or class diagrams, and are discussed within the chapter on top level model elements (see Section 16.6, "Stereotype").

18.6. Class

The class is the dominant model element on a class diagram. In the UML metamodel it is a sub-class of `Classifier` and `GeneralizableElement`.

A class is represented on a class diagram as a rectangle with three compartments. The top compartment displays the class name (and stereotypes), the second compartment any attributes and the third any operations. These last two compartments may optionally be hidden.

18.6.1. Class Details Tabs

The details tabs that are active for classes are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 18.6.2, “Class Property Toolbar” and Section 18.6.3, “Property Fields For Class” below.

`Documentation`
Standard tab. See Section 13.4, “Documentation Tab”.

`Presentation`
Standard tab. The tick boxes, `Attributes` and `Operations` allow the attributes and operations compartments to be shown (the default) or hidden. This is a setting valid for only the current diagram that shows the class. The editable `Bounds` field defines the bounding box for the package on the diagram.

`Source`
Standard tab. This contains a template for the class declaration and declarations of associated classes.

`Constraints`
Standard tab. There are no standard constraints defined for `Class` within the UML metamodel.

`Stereotypes`
Standard tab.

`Tagged Values`

Standard tab. In the UML metamodel, `Class` has the following standard tagged values defined.

- `persistence` (from the superclass, `Classifier`). Values `transitory`, indicating state is destroyed when an instance is destroyed or `persistent`, marking state is preserved when an instance is destroyed.
- `semantics` (from the superclass, `Classifier`). The value is a specification of the semantics of the class.
- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the class is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived classes still have their value in analysis to introduce useful names or concepts, and in design to avoid re-computation.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

`Checklist`

Standard tab for a Classifier.

18.6.2. Class Property Toolbar


`Go up`

Navigate up through the package structure.


`New attribute`

This creates a new attribute (see Section 18.7, “Attribute”) within the class, navigating immediately to the properties tab for that attribute.


`New operation`

This creates a new operation (see Section 18.8, “Operation”) within the class, navigating immediately to the properties tab for that operation.


`New reception`

This creates a new reception, navigating immediately to the properties tab for that reception.


`New inner class`

This creates a new inner class (which appears on no diagram) within the class. This belongs to the class and is restricted to the namespace of the class. It exactly models the Java concept of inner class. As an inner class it needs no attributes or operations, since it shares those of its owner.

Note

Inner class is not a separate concept in UML. This is a convenient shorthand for creating a class that is restricted to the namespace of its owning class.


`New class`

This creates a new class (which appears on no diagram) within the same namespace as the current class.


`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected class, navigating immediately to the properties tab for that stereotype.


`Delete`

This deletes the class from the model

Warning

This is a deletion from the model *not* just the diagram. To delete a class from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

18.6.3. Property Fields For Class

`Name`

Text box. The name of the class. The name of a class has a leading capital letter, with words separated by “bumpy caps”.

Note

The ArgoUML critics will complain about class names that do not have an initial capital.

`Namespace`

Drop down selector with navigation button. Records and allows setting of the namespace for the class. This is the package hierarchy.

Button 1 click on the entry will move the class to the selected namespace.

`Modifiers`

Check box, with entries `Abstract`, `Leaf`, `Root`, and `Active`.

- `Abstract` is used to declare that this class cannot be instantiated, but must always be subclassed. The name of an abstract class is displayed in italics on the diagram.

Caution

If a class has any abstract operations, then it should be declared abstract. ArgoUML will not enforce this.

- `Leaf` indicates that this class cannot be further subclassed, while `Root` indicates it can have no superclass. It is possible for a class to be both Abstract and Leaf, since its static operations may still be referenced.

- `Active` indicates that this class exhibits dynamic behavior (and is thus associated with a state or activity diagram).

`Visibility`

Radio box, with four entries `public`, `private`, `protected`, and `package`. Indicates whether the class is visible outside the namespace.

`Client Dependencies`

Text area. Lists the “depending” ends of the relationship, i.e. the end that makes use of the other end.

Button 1 double click navigates to the dependency and opens its property tab.

Button 2 click shows a pop-up menu with one entry `Add...` that opens a dialog box where you can add and remove depending modelelements.

`Supplier Dependencies`

Text area. Lists the “supplying” ends of the relationship, i.e. the end supplying what is needed by the other end.

Button 1 double click navigates to the dependency and opens its property tab.

Button 2 click shows a pop-up menu with one entry `Add...` that opens a dialog box where you can add and remove dependent modelelements.

`Generalizations`
Text area. Lists any class that *generalizes* this class.

Button 1 double click navigates to the generalization and opens its property tab.

`Specializations`
Text box. Lists any specialized class (i.e. for which this class is a generalization).

Button 1 double click navigates to the generalization and opens its property tab.

`Attributes`
Text area. Lists all the attributes (see Section 18.7, “Attribute”) defined for this class. Button 1 double click navigates to the selected attribute. Button 2 gives a pop up menu with two entries, which allow reordering the attributes.

- `Move Up`. Only available where there are two or more attributes listed, and the attribute selected is not at the top. It moves the attribute up one position.
- `Move Down`. Only available where there are two or more attributes listed, and the attribute selected is not at the bottom. It moves the attribute down one position.

`Association Ends`
Text box. Lists any association ends (see Section 18.12, “Association”) of associations connected to this class.

Button 1 double click navigates to the selected entry.

`Operations`
Text area. Lists all the operations (see Section 18.8, “Operation”) defined on this class. Button 1 click navigates to the selected operation. Button 2 gives a pop up menu with two entries, which allow reordering the operations.

- `Move Up`. Only available where there are two or more operations listed, and the operation selected is not at the top. It moves the operation up one position.
- `Move Down`. Only available where there are two or more operations listed, and the operation selected is not at the bottom. It moves the operation down one position.

`Owned Elements`
Text area. A listing of model elements contained within the classes' namespace. This is where any inner class (see Section 18.6.2, “Class Property Toolbar”) will appear

Button 1 double click on any of the model elements navigates to that model element.

Tip

Most namespace hierarchies should be managed through the package mechanism. Namespace hierarchies through classes are best restricted to inner classes. Conceivable datatypes, signals and interfaces could also appear here, but actors and use cases would seem of no value.

18.7. Attribute

Attribute is a named slot within a class (or other `Classifier`) describing a range of values that may be held by instances of the class. In the UML metamodel it is a sub-class of `StructuralFeature` which is itself a sub-class of `Feature`.

An attribute is represented in the diagram on a single line within the attribute compartment of the class. Its syntax is as follows:

visibility attributeName : *type* [= *initialValue*]

visibility is +, #, - or ~ corresponding to `public`, `protected`, `private`, or `package` visibility respectively.

attributeName is the actual name of the attribute being declared.

type is the type (UML datatype, class or interface) declared for the attribute.

initialValue is any initial value to be given to the attribute when an instance of the class is created. This may be overridden by any constructor operation.

In addition any attribute declared static will have its whole entry underlined on the diagram.

18.7.1. Attribute Details Tabs

The details tabs that are active for attributes are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 18.7.2, “Attribute Property Toolbar” and Section 18.7.3, “Property Fields For Attribute” below.

`Documentation`
Standard tab. See Section 13.4, “Documentation Tab”.

`Constraints`
Standard tab. There are no standard constraints defined for `Attribute` within the UML metamodel.

`Stereotype`
Standard tab.

`Tagged Values`

Standard tab. In the UML metamodel, `Attribute` has the following standard tagged values defined.

- `transient`.
- `volatile`. This is an ArgoUML extension to the UML 1.4 standard to indicate that this attribute is realized in some volatile form (for example it will be a memory mapped control register).

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

`Checklist`

Standard tab for a Attribute.

18.7.2. Attribute Property Toolbar


Go up

Navigate up through the package structure.


Go to Previous

Navigate to the previous attribute of the class that owns them. This button is downlighted if the current attribute is the first one.


Go to Next

Navigate to the next attribute of the class that owns them. This button is downlighted if the current attribute is the last one.


New attribute

This creates a new attribute within the owning class of the current attribute, navigating immediately to the properties tab for that attribute.

Tip

This is a very convenient way to add a number of attributes, one after the other, to a class.


New Datatype

This creates a new Datatype (see Section 16.3, “Datatype”) for the selected attribute, navigating immediately to the properties tab for that datatype.


New Enumeration

This creates a new Enumeration (see Section 16.4, “Enumeration”) for the package that owns the class, navigating immediately to the properties tab for that enumeration.


New Stereotype

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected attribute, navigating immediately to the properties tab for that stereotype.


Delete

This deletes the attribute from the model

Warning

This is a deletion from the model *not* just the diagram. If desired the whole attribute compartment can be hidden on the diagram using the *style* tab (see Section 18.7.2, “Attribute Property Toolbar”) or the button 2 pop up menu for the class on the diagram.

18.7.3. Property Fields For Attribute

`Name`

Text box. The name of the attribute. The name of a attribute has a leading lower case letter, with words separated by “bumpy caps”.

Note

The ArgoUML critics will complain about attribute names that do not have an initial lower case letter.

`Owner`

Text box. Records the class which contains this attribute.

Button 1 double click on the entry will navigate to the class.

`Multiplicity`

Editable drop down selector with checkmark. The default value (1) is that there is one instance of this attribute for each instance of the class, i.e. it is a scalar. The drop down provides a number of commonly used specifications for non-scalar attributes.

When the checkmark is unchecked, then the multiplicity remains undefind in the model (and the drop down selector is downlighted).

Note

ArgoUML presents a number of predefined ranges for multiplicity for easy access. The user may also enter any user defined range that follows the UML syntax, such as “1..3,7,10”.

The value `1..1` is equivalent to the default (exactly one scalar instance). The selection `0..1` indicates an optional scalar attribute.

`Visibility`

Radio box, with entries `public`, `private`, `protected` and `package`.

- `public`. The attribute is available to any model element that can see the owning class.
- `private`. The attribute is available only to the owning class (and any inner classes).
- `protected`. The attribute is available only to the owning class, or model elements that are subclasses of the owning class.
- `package`. The attribute is available only to model elements contained in the same package.

`Changeability`

Radio box, with entries `addOnly`, `changeable`, and `frozen`.

- `addOnly`. Meaningful only if the multiplicity is not fixed to a single value. Additional values may be added to the set of values, but once created a value may not be removed or altered.
- `changeable`. There are no restrictions of modification.

- `frozen`. Also named “immutable”. The value of the attribute may not change during the lifetime of the owner class. The value must be set at object creation, and may never change after that. This implies that there is usually an argument for this value in a constructor and that there is no operation that updates this value.

`Modifiers`

Check box for `static`. If unchecked (the defaults) then the attribute has “instance scope”. If checked, then the attribute is static, i.e. it has “class scope”. Static attributes are indicated on the diagram by underlining.

`Type`

Drop down selector with navigation button. The type of this attribute. This can be any UML `Classifier`, although in practice only `Class`, `DataType`, or `Interface` make any sense.

Pressing the navigation button will navigate to the property panel for the currently selected type. (see Section 18.6, “Class”, Section 18.3, “Datatype” and Section 18.16, “Interface”).

Note

A type must be declared (it can be `void`). By default ArgoUML supplies `int` as the type.

`Initial Value`

Text box with 2 compartments. This allows you to set an initial value for the attribute if desired (this is optional). The drop down menu provides access to the common values `0`, `1`, `2`, and `null`.

The left hand side of this field contains the body of the expression that forms the initial value. The right hand side defines the language in which the expression is written.

Hovering the mouse pointer over these fields, reveals a tooltip `Body` or `Language`, to help remember which is which.

Caution

Any constructor operation may ignore this initial value.

18.8. Operation

An operation is a service that can be requested from an object to effect behavior. In the UML metamodel it is a sub-class of `BehavioralFeature` which is itself a sub-class of `Feature`.

In the diagram, an operation is represented on a single line within the operation compartment of the class. Its syntax is as follows:

visibility name (parameter list) : return-type-expression {property-string}

You can edit this line directly in the diagram, by double-clicking on it. All elements are optional and, if left unspecified, the old values will be preserved.

A *stereotype* can be given between any two elements in the line in the format: `<<stereotype>>`.

The following properties are recognized to have special meaning: abstract, concurrency, concurrent,

guarded, leaf, query, root and sequential.

The *visibility* is +, #, - or ~ corresponding to `public`, `protected`, `private` visibility, or `package` visibility respectively.

`static` and `final` optionally appear if the operation has those modifiers. Any operation declared static will have its whole entry underlined on the diagram.

There may be zero or more entries in the *parameter list* separated by commas. Every entry is a pair of the form:

name : *type*

The *return-type-expression* is the type (UML datatype, class or interface) of the result returned.

Finally the whole entry is shown in italics if the operation is declared abstract.

18.8.1. Operation Details Tabs

The details tabs that are active for operations are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 18.8.2, “Operation Property Toolbar” and Section 18.8.3, “Property Fields For Operation” below.

`Documentation`
Standard tab. See Section 13.4, “Documentation Tab”.

`Presentation`
Standard tab. The `Bounds:` field does allow editing, but the changes have no effect.

`Source`
Standard tab. This contains a declaration for the operation.

`Constraints`
Standard tab. There are no standard constraints defined for `Operation` within the UML metamodel.

`Tagged Values`

Standard tab. In the UML metamodel, `Operation` has the following standard tagged values defined.

- `semantics`. The value is a specification of the semantics of the operation.
- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the operation is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived operations still have their value in analysis to introduce useful names or concepts, and in design to avoid re-computation.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

`Checklist`
Standard tab for an Operation.

18.8.2. Operation Property Toolbar

`Go up`

Navigate up through the package structure.

`New operation`

This creates a new operation within the owning class of the current operation, navigating immediately to the properties tab for that operation.

Tip

This is a very convenient way to add a number of operations, one after the other, to a class.

`New parameter`

This creates a new parameter for the operation, navigating immediately to the properties tab for that parameter.

`New raised signal`

This creates a new raised signal for the operation, navigating immediately to the properties tab for that raised signal.

`New Datatype`

This creates a new Datatype (see Section 16.3, “Datatype”) in the namespace of the owner of the operation, navigating immediately to the properties tab for that datatype.

`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected operation, navigating immediately to the properties tab for that stereotype.

`Delete`

This deletes the operation from the model

Warning

This is a deletion from the model *not* just the diagram. If desired the whole operation compartment can be hidden on the diagram using the *presentation* tab (see Sec-

tion 18.8.2, “Operation Property Toolbar”) or the button 2 pop up menu for the class on the diagram.

18.8.3. Property Fields For Operation

`Name`

Text box. The name of the operation. The name of an operation has a leading lower case letter, with words separated by “bumpy caps”.

Note

The ArgoUML critics will complain about operation names that do not have an initial lower case letter.

Tip

If you wish to follow the Java convention of constructors having the same name as the class, you will violate this rule. Silence the critic by setting the stereotype `create` for the constructor operation.

`Stereotype`

Drop down selector. There are two UML standard stereotypes for `Operation` (from the parent metaclass, `BehavioralFeature`), `create` and `destroy`.

Tip

You should use `create` as the stereotype for constructors, and `destroy` for destructors (which are called “finalize” methods under Java).

`Navigate Stereotype`

 icon. If a stereotype has been selected, clicking button 1 will navigate to the stereotype property panel (see Section 18.5, “Stereotype”).

`Owner`

Text box. Records the class which contains this operation.

Button 1 double click on the entry will navigate to the class.

`Visibility`

Radio box, with entries `public`, `private`, `protected` and `package`.

- `public`. The operation is available to any model element that can see the owning class.
- `private`. The operation is available only to the owning class (and any inner classes).
- `protected`. The operation is available only to the owning class, or model elements that are subclasses of the owning class.

- `package`. The operation is available only model elements contained in the same package.

`Modifiers`

Check box, with entries `abstract`, `leaf`, `root`, `query`, and `static`.

- `abstract`. This operation has no implementation with this class. The implementation must be provided by a subclass.

Important

Any class with an abstract operation must itself be declared abstract.

- `leaf`. The implementation of this operation must not be overridden by any subclass.
- `root`. The declaration of this operation must not override a declaration of the operation from a superclass.
- `query`. This indicates that the operation must have no side effects (i.e. it must not change the state of the system). It can only return a value.

Caution

Operations for user defined datatypes must always check this modifier.

- `static`. There is only one instance of this operation associated with the class (as opposed to one for each instance of the class). This is the `OwnerScope` attribute of a `Feature` metaclass within UML. Any operation declared static is shown underlined on the class diagram.

`Concurrency`

Radio box, with entries `guarded`, `sequential`, and `concurrent`.

- `guarded`. Multiple calls from concurrent threads may occur simultaneously to one instance (on any guarded operation), but only one is allowed to commence. The others are blocked until the performance of the first operation is complete.

Caution

It is up to the system designer to ensure that deadlock cannot occur. It is the responsibility of the operation to implement the blocking behavior (as opposed to the system).

- `sequential`. Only one call to an instance (of the class with the operation) may be outstanding at any one time. There is no protection, and no guarantee of behavior if the system violates this rule.
- `concurrent`. Multiple calls to one instance may execute at the same time. The operation is responsible for ensuring correct behavior. This must be managed even if there are other sequential or synchronized (guarded) operations executing at the time.

`Parameter`

Text area, with entries for all the parameters of the operation (see Section 18.9, “Parameter”). A new operation is always created with one new parameter, `return` to define the return type of the

operation.

Button 1 double click on any of the parameters navigates to that parameter. Button 2 click brings up a pop up menu with two entries.

- `Move Up`. Only available where there are two or more parameters, and the parameter selected is not at the top. It is moved up one position.
- `Move Down`. Only available where there are two or more parameters listed, and the parameter selected is not at the bottom. It is moved down one position.

`Raised Signals`

Text area, with entries for all the signals (see Section 18.10, “Signal”) that can be raised by the operation.

Caution

ArgoUML at present (V0.18) has limited support for signals. In particular they are not linked to signal events that could drive state machines.

Button 1 double click on any of the signals navigates to that parameter.

18.9. Parameter

A parameter is a variable that can be passed. In the UML metamodel it is a sub-class of `ModelElement`.

A parameter is represented within the operation declaration in the operation compartment of a class as follows.

name : *type*

name is the name of the parameter.

type is the type (UML datatype, class or interface) of the parameter.

The exception is any parameter representing a return value, whose type only is shown at the end of the operation declaration.

18.9.1. Parameter Details Tabs

The details tabs that are active for parameters are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 18.9.2, “Parameter Property Toolbar” and Section 18.9.3, “Property Fields For Parameter” below.

`Documentation`
Standard tab. See Section 13.4, “Documentation Tab”.

`Source`
Standard tab. This contains a declaration for the parameter.

`Tagged Values`

Standard tab. In the UML metamodel, `Parameter` has the following standard tagged values defined.

- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the parameter is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Caution

A derived parameter is a meaningless concept.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

18.9.2. Parameter Property Toolbar


`Go up`

Navigate up through the package structure.


`New parameter`

This creates a new parameter for the for the same operation as the current parameter, navigating immediately to the properties tab for that parameter.

Tip

This is a convenient way to add a series of parameters for the same operation.


`New Datatype`

This creates a new Datatype (see Section 16.3, “Datatype”) in the namespace of the owner of the operation of the parameter, navigating immediately to the properties tab for that datatype.


`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected parameter, navigating immediately to the properties tab for that stereotype.


`Delete`

This deletes the parameter from the model

Warning

This is a deletion from the model *not* just the diagram. If desired the whole operation compartment can be hidden on the diagram using the *presentation* tab or the button 2 pop up menu for the class on the diagram.

18.9.3. Property Fields For Parameter

`Name`

Text box. The name of the parameter. By convention, the name of a parameter has a leading lower case letter, with words separated by “bumpy caps”.

Note

The ArgoUML critics do not complain about parameter names that do not have an initial lower case letter.

`Stereotype`

Drop down selector. There are no UML standard stereotypes for `Parameter`.

`Navigate Stereotype`

 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 16.6, “Stereotype”).

`Owner`

Text box. Records the operation which contains this parameter.

Button 1 double click on the entry will navigate to the operation.

`Type`

Drop down selector. The type of this parameter. This can be any UML `Classifier`, although in practice only `Class`, `DataType`, or `Interface` make any sense.

Note

A type must be declared (it can be `void`, but this only makes sense for a return parameter). By default ArgoUML supplies `int` as the type the first time a parameter is created, and thereafter the type of the most recently created parameter.

`Default Value`

Text box with drop down. This allows you to set an initial value for the parameter if desired (this is optional). The drop down menu provides access to the common values `0`, `1`, `2`, and `null`.

Caution

This only makes sense for `out` or `return` parameters.

`Kind`

Radio box, with entries `out`, `in/out`, `return`, and `in`.

- `out`. The parameter is used only to pass values back from the operation.
- `in/out`. The parameter is used both to pass values in and to pass results back out of the operation.

Note

This is the default for any new parameter.

- `return`. The parameter is a return result from the call.

Note

There is nothing to stop you declaring more than one return parameter (some programming languages support this concept).

Tip

The name of the return parameter does not appear on the diagram, but it is convenient to give it an appropriate name (such as the default `return` to identify it in the list of parameters on the operation property tab.

- `in`. The parameter is used only to pass values in to the operation.

18.10. Signal

A signal is a specification of an asynchronous stimulus communicated between instances. In the UML metamodel it is a sub-class of `Classifier`.

Within ArgoUML signals are not fully handled. Their value is when they are received as *signal events* driving the asynchronous behavior of state machines and when associated with *send actions* in state machines and messages for collaboration diagrams.

Tip

In general there is limited value at present in defining signals within ArgoUML. It may prove more useful to define signals as classes, with a (user defined) stereotype of «`signal`» as suggested in the UML 1.4 standard. This allows any dependency relationships between signals to be shown.

18.10.1. Signal Details Tabs

The details tabs that are active for signals are as follows.

`ToDoItem`
: Standard tab.

`Properties`

See Section 18.10.2, “Signal Property Toolbar” and Section 18.10.3, “Property Fields For Signal” below.

`Documentation`
Standard tab. See Section 13.4, “Documentation Tab”.

`Source`
Standard tab. There is nothing generated for a signal.

`Tagged Values`

Standard tab. In the UML metamodel, `Signal` has the following standard tagged values defined.

- `persistence` (from the superclass, `Classifier`). Values `transitory`, indicating state is destroyed when an instance is destroyed or `persistent`, marking state is preserved when an instance is destroyed.
- `semantics` (from the superclass, `Classifier`). The value is a specification of the semantics of the signal.
- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the signal is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived signals still have their value in analysis to introduce useful names or concepts, and in design to avoid re-computation.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

18.10.2. Signal Property Toolbar

`Go up`

Navigate up through the package structure.


`New signal`

This creates a new signal, navigating immediately to the properties tab for that signal.

Caution

The signal is not associated with the same operation as the original signal, so this will have to be done afterwards.

`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected signal, navigating immediately to the properties tab for that stereotype.


`Delete`

This deletes the signal from the model

Warning

This is a deletion from the model.

18.10.3. Property Fields For Signal

`Name`

Text box. The name of the signal. From their similarity to classes, by convention, the name of a signal has a leading upper case letter, with words separated by “bumpy caps”.

Note

The ArgoUML critics do not complain about signal names that do not have an initial upper case letter.

`Stereotype`

Drop down selector. Signal is provided by default with the UML standard stereotypes for its parent in the UML meta-model, `Classifier` (`metaclass`, `powerType`, `process`, `thread`, and `utility`).

`Navigate Stereotype`

 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 16.6, “Stereotype”).

`Namespace`

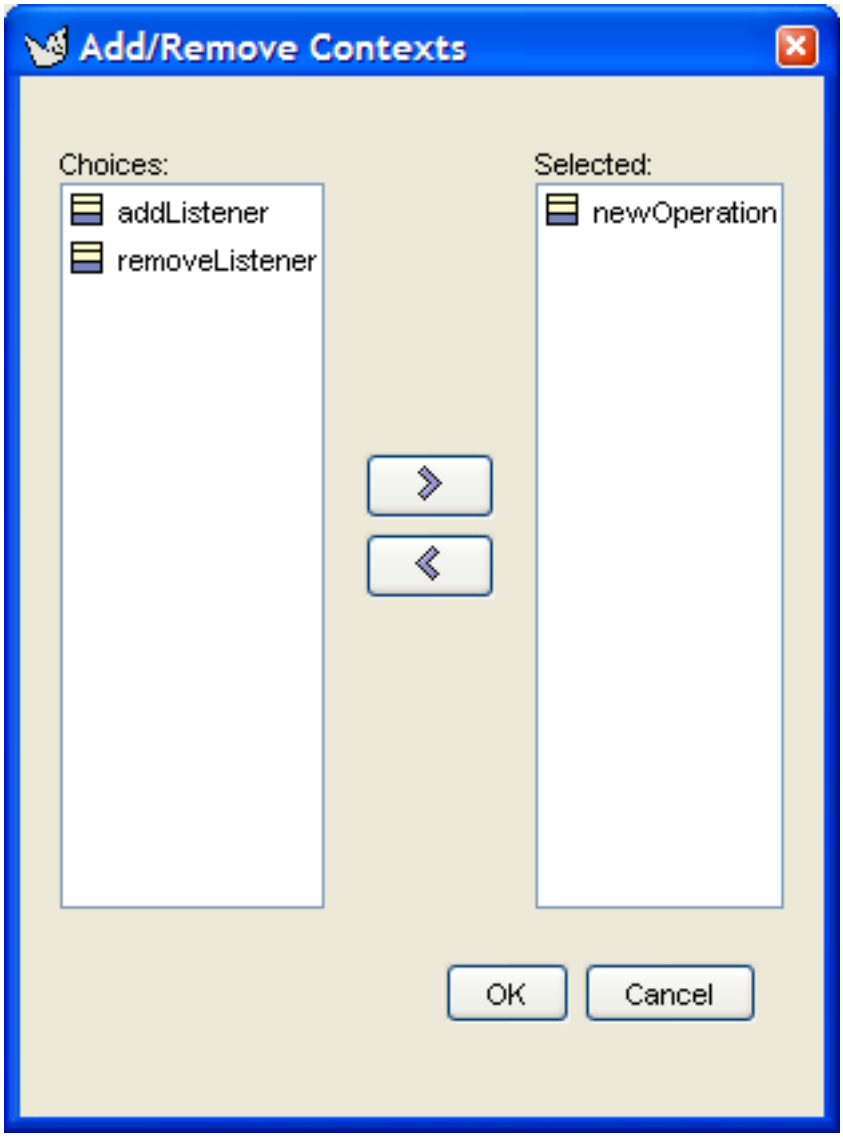
Drop down selector. Records and allows changing the namespace for the signal. This is the package hierarchy of the signal.

`Contexts`

Text area. Lists all the contexts defined for this signal. Button 1 double click navigates to the selected context, button 2 click brings up a pop up menu with one entry.

- `Add`. Add a new context. This opens the *Add/Remove Contexts* dialog box (see figure below), which allows choosing between all possible operations, and adding them to the selected list.

Figure 18.5. The “add/remove context” dialog box



18.11. Reception (to be written)

A reception is ...

18.12. Association

An association on a class diagram represents a relationship between classes, or between a class and an interface. On a usecase diagram, an association binds an actor to a usecase.

Within the UML metamodel, `Association` is a sub-class of both `Relationship` and `GeneralizableElement`.

The association is represented as a solid line connecting actor and usecase or class or interface (see Figure 18.1, “Possible model elements on a class diagram.”). The name of the association and any stereo-

type appear above the line.

ArgoUML is not restricted to binary associations. See Section 18.12.1, “Three-way and Greater Associations and Association Classes” for more on this.

Associations are permitted between interfaces and classes, but UML 1.3 specifies they must only be navigable toward the interface—in other words the interface cannot see the class. ArgoUML will draw such associations with the appropriate navigation.

Associations are often not named, when their meaning is obvious from the context.

Note

ArgoUML provides no specific way of showing the direction of the association as described in the UML 1.4 standard. The naming should attempt to make this clear.

The association contains at least two ends, which may be navigated to via the association property sheet. See Section 18.13, “Association End” for more information.

18.12.1. Three-way and Greater Associations and Association Classes

UML 1.3 provides for N-ary associations and associations that are governed by a third *associative class*. Both are supported by ArgoUML.

N-ary associations are created by drawing with the association tool from an existing association to a third class. The current implementation of ArgoUML does not allow the inverse: drawing from a 3rd class towards an existing association is not possible.

Association Classes are drawn exactly like a normal association, i.e. between two classes, but with a different dedicated tool from the diagram toolbar.

18.12.2. Association Details Tabs

The details tabs that are active for associations are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 18.12.3, “Association Property Toolbar” and Section 18.12.4, “Property Fields For Association” below.

`Documentation`
: Standard tab. See Section 13.4, “Documentation Tab”.

`Presentation`
: Standard tab.

Note

The values for the bounds of the Association have no meaning, since they are determined by the location of the connected items. Changing them has no effect on the diagram.

`Source`
Standard tab. You would not expect to generate any code for an association, and any code entered here is ignored (it will have disappeared when you come back to the association.

`Tagged Values`

Standard tab. In the UML metamodel, `Association` has the following standard tagged values defined.

- `persistence`. Values `transitory`, indicating state is destroyed when an instance is destroyed or `persistent`, marking state is preserved when an instance is destroyed.
- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the association is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived associations still have their value in analysis to introduce useful names or concepts, and in design to avoid re-computation.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

18.12.3. Association Property Toolbar


`Go up`

Navigate up through the package structure of the model. For an association this will be the package containing the association.

`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected association, navigating immediately to the properties tab for that stereotype.


`Delete`

This deletes the selected association from the model.

Warning

This is a deletion from the model *not* just the diagram. To delete an association from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

18.12.4. Property Fields For Association

`Name`

Text box. The name of the association. By convention association names start with a lower case letter, with “bumpy caps” used to indicate words within the name, thus: `salesHandling`.

Note

ArgoUML does not enforce any naming convention for associations.

Tip

Although the design critics will advise otherwise, it is perfectly normal not to name associations on a class diagram, since the relationship is often obvious from the classes (or class and interface) name.

`Stereotype`

Drop down selector. Association is provided by default with the UML standard stereotype for Association (`implicit`) .

Stereotyping can be useful when creating associations in the problem domain (requirements capture) and solution domain (analysis), as well as for processes based on patterns.

The stereotype is shown between « and » below the name of the association on the diagram.

`Navigate Stereotype`

 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 16.6, “Stereotype”).

`Namespace`

Drop down selector. Records and allows changing the namespace for the association. This is the package hierarchy.

`Connections`

Text area. Lists the ends of this association. An association can have two or more ends. For more on association ends see Section 18.13, “Association End”.

The names of the association ends are listed, unless the association end has no name (the case when it is first created), in which case `(Unnamed AssociationEnd)` is shown.

Note

The only representation of association ends on a diagram is that their name appears at the relevant end of the corresponding association.

Button 1 double click on an association end will navigate to that end.

`Association Roles`

Text area. (To be written)

`Links`

Text area. (To be written)

18.13. Association End

Two or more association ends are associated with each association (see Section 17.5, “Association”).

Within the UML metamodel, `AssociationEnd` is a sub-class of `ModelElement`.

The association end has no direct access on any diagram for binary associations. The ends of an N-ary association may be selected by clicking on the line in the diagram. The stereotype, name and multiplicity are shown at the relevant end of the parent association (see Figure 17.1, “Typical model elements on a use case diagram.”). Where shared or composite aggregation is selected for one association end, the opposite end is shown as a solid diamond (composite aggregation) or hollow diamond (shared aggregation).

Tip

Although you can change attributes of association ends when creating a use case model, this is often not necessary. Many of the properties of an association end relate to its use in class diagrams, and are of limited relevance to use cases. The most useful attributes to consider altering are the name (used as the role name) and the multiplicity.

Note

ArgoUML does not currently support showing qualifiers on the diagram, as described in the UML 1.3 standard.

18.13.1. Association End Details Tabs

The details tabs that are active for associations are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 18.13.2, “Association End Property Toolbar” and Section 18.13.3, “Property Fields For Association End” below.

`Documentation`
: Standard tab. See Section 13.4, “Documentation Tab”.

`Presentation`
: Standard tab.

`Source`
: Standard tab. This tab contains a declaration for the association end as an instance of the model element to which it is connected.

`Tagged Values`

: Standard tab. In the UML metamodel, `AssociationEnd` has the following standard tagged values defined.

  - `derived` (from the superclass, `ModelElement`). Values `true`, meaning the association end is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Tip

Derived association ends still have their value in analysis to introduce useful names or concepts, and in design to avoid re-computation. However the tag only makes sense for an association end if it is also applied to the parent association.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

18.13.2. Association End Property Toolbar

`Go up`

Navigate up to the association to which this end belongs.

`Go Opposite`

This navigates to the other end of the association.

`New Qualifier`

This creates a new Qualifier for the selected association-end, navigating immediately to the properties tab for that qualifier.

Warning

Qualifiers are only partly supported in ArgoUML V0.18. Hence, activating this button creates a qualifier in the model, which is not shown on the diagram. Also, the properties panel for a qualifier equals that of a regular attribute.

`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected association-end, navigating immediately to the properties tab for that stereotype.

`Delete`

This deletes the selected association-end from the model.

Note

This button is downlighted for binary associations, since an association needs at least *two* ends. Only for N-ary associations, this button is accessable, and deletes just one end from the association.

18.13.3. Property Fields For Association End

`Name`

Text box. The name of the association end, which provides a *role name* for this end of the association. This role name can be used for navigation, and in an implementation context, provides a name by which the source end of an association can reference the target end.

Note

ArgoUML does not enforce any naming convention for association ends.

`Stereotype`

Drop down selector. Association end is provided by default with the UML standard stereotypes for AssociationEnd (`association`, `global`, `local`, `parameter`, `self`).

`Navigate Stereotype`

 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 16.6, “Stereotype”).

`Association`

Text box. Records the parent association for this association end. Button 1 double click on this entry will navigate to that association.

`Type`

Drop down selector providing access to all standard UML types provided by ArgoUML and all new classes created within the current model.

This is the type of the entity attached to this end of the association.

Tip

By default ArgoUML will select the class of the model element to which the linkend is connected. However, an association can be moved to another class by selecting another entry here.

`Multiplicity`

Drop down menu with edit box. The value can be chosen from the drop down box, or a new one can be edited in the text box. Records the multiplicity of this association end (with respect to the other end), i.e. how many instances of this end may be associated with an instance of the other end. The multiplicity is shown on the diagram at that end of the association.

`Modifiers`

There are 3 modifiers: `navigable`, `ordered` and `static`. All 3 are checkboxes.

- `navigable`. Indicates that this end can be navigated to from the other end.

Note

The UML 1.4 standard provides a number of options for how navigation is dis-

played on an association end. ArgoUML uses option 3, which means that arrow heads are shown at the end of an association, when navigation is enabled at only one end, to indicate the direction in which navigation is possible. This means that the default, with both ends navigable has no arrows.

- `ordered` When placed on one end, specifies whether the set of links from the other instance to this instance is ordered. The ordering must be determined and maintained by Operations that add links. It represents additional information not inherent in the objects or links themselves. Possibilities for the checkbox are: Unchecked - The links form a set with no inherent ordering. Checked - A set of ordered links can be scanned in order.

- `Static` (To be written)

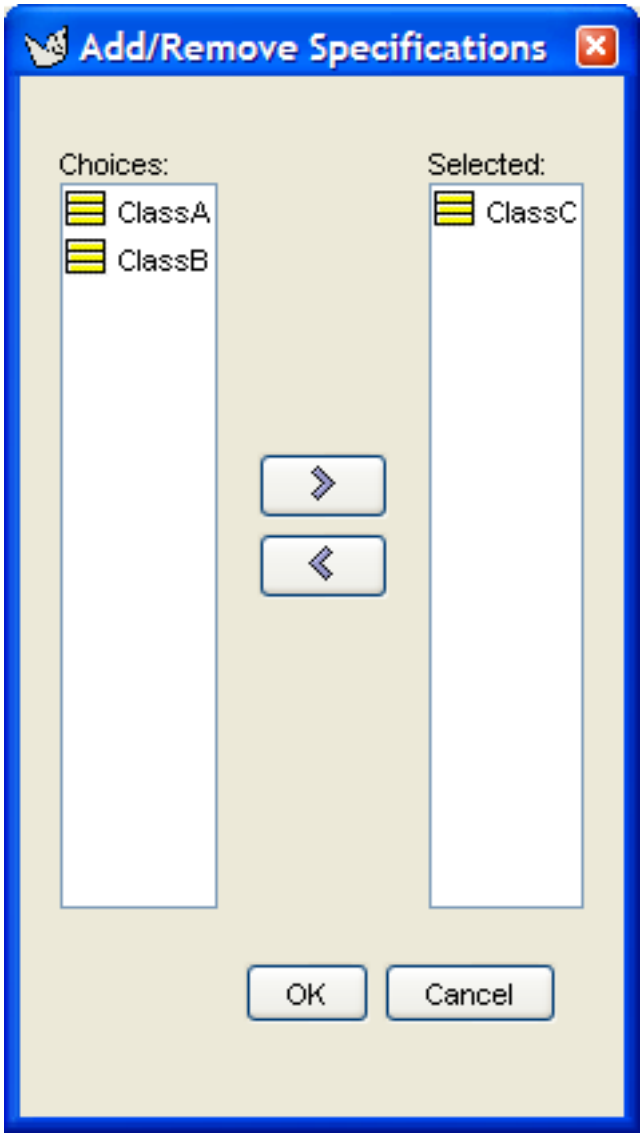
`Specification`

List. Designates zero or more Classifiers that specify the Operations that may be applied to an Instance accessed by the AssociationEnd across the Association. These determine the minimum interface that must be realized by the actual Classifier attached to the end to support the intent of the Association. May be an Interface or another Classifier. The type of classifier is indicated by an icon.

Button 1 double click navigates to the selected classifier, button 2 click brings a pop up menu with one entry.

- `Add`. Add a new specification classifier. This opens the *Add/Remove Specifications* dialog box (see figure below), which allows choosing between all possible classifiers, and adding or removing them to the selected list.

Figure 18.6. The “Add/Remove Specifications” dialog box



`Qualifiers`

Text box. Records the qualifiers for this association end. Button 1 double click on this entry will navigate to that qualifier. Button 2 click will show a popup menu containing two items: `Move Up` and `Move Down`, which allow reordering the qualifiers.

`Aggregation`

Radio box, with three entries `composite`, `none` and `aggregate`. Indicates whether the relationship with the far end represents some type of loose whole-part relationship (`aggregation`) or tight whole-part relationship (`composite`).

Shared aggregation is shown by a hollow diamond at the “whole” end of the association. Composite aggregation is shown by a solid diamond.

Note

You may not have aggregation at both ends of an association. ArgoUML does not en-

force this constraint.

The “whole” end of a composite aggregation should have a multiplicity of one. ArgoUML does not enforce this constraint.

`Changeability`

Radio box, with three entries `add only`, `changeable` and `frozen`. Indicates whether instances of this end of the association-end may be: i) created but not deleted after the target instance is created; ii) created and deleted by the source after the target instance is created; or iii) not created or deleted by the source after the target instance is created.

`Visibility`

Radio box, with four entries `public`, `private`, `protected`, and `package`. Indicates whether navigation to this end may be by: i) any classifier; ii) only by the source classifier; or iii) only the source classifier and its children.

18.14. Dependency

Dependency is a relationship between two model elements showing that one depends on the other.

Within the UML metamodel, `Dependency` is a sub-class of `Relationship`.

Dependency is represented as a dashed line with an open arrow head from the depending model element to that which it is dependent upon.

18.14.1. Dependency Details Tabs

The details tabs that are active for dependencies are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 18.14.2, “Dependency Property Toolbar” and Section 18.14.3, “Property Fields For Dependency” below.

`Documentation`
Standard tab. See Section 13.4, “Documentation Tab”.

`Presentation`
Standard tab

Note

The values for the bounds of the dependency are downlighted, given the dependency is tied to a particular modelelement.

`Tagged Values`
Standard tab. In the UML metamodel, `Dependency` has no tagged values of its own, but through superclasses has the following standard tagged values defined.

- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the dependency relationship is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived dependencies still have their value in analysis to introduce useful names or concepts.

18.14.2. Dependency Property Toolbar

`Go up`

Navigate up through the package structure of the model. For a dependency this will be the package containing the dependency.

`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected dependency, navigating immediately to the properties tab for that stereotype.

`Delete`

This deletes the selected dependency from the model.

Warning

This is a deletion from the model *not* just the diagram. To delete a dependency from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

18.14.3. Property Fields For Dependency

`Name`

Text box. The name of the dependency.

Tip

It is quite common to leave dependencies unnamed.

Note

ArgoUML does not enforce any naming convention for associations.

Note

There is no representation of the name of a dependency on the diagram.

`Stereotype`

Drop down selector. Dependency has no standard stereotypes of its own under UML 1.3. and so ArgoUML does not provide any. The stereotype is shown between « and » above or across the generalization.

`Navigate Stereotype`

 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 16.6, “Stereotype”).

`Namespace`

Text box. Records the namespace for the dependency. This is the package hierarchy.

`Suppliers`

Text area. Lists the end of the relationship that is supplying what is needed by the other end.

Button 1 double click on a supplier will navigate to that element.

`Clients`

Text area. Lists the “depending” ends of the relationship, i.e. the end that makes use of the other end.

Button 1 double click on a client will navigate to that element.

18.15. Generalization

Generalization is described under use case diagrams (see Section 17.8, “Generalization”).

Note

Within the context of classes, generalization and specialization are the UML terms describing class inheritance.

18.16. Interface

An interface is a set of operations characterizing the behavior of an element. It can be usefully thought of as an abstract class with no attributes and no non-abstract operations. In the UML metamodel it is a sub-class of `Classifier` and through that `GeneralizableElement`.

An interface is represented on a class diagram as a rectangle with two horizontal compartments. The top compartment displays the interface name (and above it «`interface`») and the second any operations. Just like a class, the operations compartment can be hidden.

18.16.1. Interface Details Tabs

The details tabs that are active for interfaces are as follows.

`ToDoItem`

Standard tab.

`Properties`
See Section 18.16.2, “Interface Property Toolbar” and Section 18.16.3, “Property Fields For Interface” below.

`Documentation`
Standard tab. See Section 13.4, “Documentation Tab”.

`Presentation`
Standard tab. The tick box `Display Operations` allows the operation compartment to be shown (the default) or hidden. This is a setting valid for only the current diagram. The `Bounds:` field defines the bounding box for the package on the diagram.

`Source`
Standard tab. This contains a template for the interface declaration and declarations of associated interfaces.

`Tagged Values`
Standard tab. In the UML metamodel, `Interface` has the following standard tagged values defined.

- `persistence` (from the superclass, `Classifier`). Values `transitory`, indicating state is destroyed when an instance is destroyed or `persistent`, marking state is preserved when an instance is destroyed.

Warning

Since interfaces are by definition abstract, they can have no instance, and so this tagged value must refer to the properties of the realizing class.

- `semantics` (from the superclass, `Classifier`). The value is a specification of the semantics of the interface.
- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the interface is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived interfaces still have their value in analysis to introduce useful names or concepts, and in design to avoid re-computation.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

`Checklist`
Standard tab for an Interface.

18.16.2. Interface Property Toolbar

`Go up`

Navigate up through the package structure.

 `New operation`

This creates a new operation (see Section 18.8, “Operation”) within the interface, navigating immediately to the properties tab for that operation.

 `New reception`

This creates a new reception, navigating immediately to the properties tab for that reception.

 `New interface`

This creates a new interface in the same namespace as the selected interface, navigating immediately to the properties tab for the new interface.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected interface, navigating immediately to the properties tab for that stereotype.

 `Delete`

This deletes the interface from the model

Warning

This is a deletion from the model *not* just the diagram. To delete an interface from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

18.16.3. Property Fields For Interface

`Name`
Text box. The name of the interface. The name of an interface has a leading capital letter, with words separated by “bumpy caps”.

Note

Unlike classes, the ArgoUML critics will not complain about interface names that do not have an initial capital.

`Stereotype`
Drop down selector. Interface is provided by default with the UML standard stereotypes for the parent meta-class, `Classifier` (`metaclass`, `powertype`, `process`, `thread` and `utility`).

`Navigate Stereotype`
 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 16.6, “Stereotype”).

`Namespace`
Drop down selector. Records and allows changing the namespace for the interface. This is the package hierarchy.

`Modifiers`
Check box, with entries `Abstract`, `Leaf` and `Root`.

- `Abstract` is used to declare that this interface cannot be instantiated, but must always be specialized. The name of an abstract interface is displayed in italics on the diagram.

Caution

This is meaningless, since by definition an interface is an abstract entity. The UML 1.3 standard offers no clarification.

- `Leaf` indicates that this interface cannot be further specialized, while `Root` indicates it can have no generalizations.

`Visibility`
Radio box, with three entries `public`, `protected`, `private` and `package`. Indicates whether navigation to this end may be by: i) any classifier; ii) only the source classifier and its children; or iii) only by the source classifier.

`Generalizations`
Text area. Lists any interface that *generalizes* this interface.

Button 1 double click navigates to the generalization and opens its property tab.

`Specializations`
Text box. Lists any specialized interface (i.e. for which this interface is a generalization.

Button 1 double click navigates to the generalization and opens its property tab.

`AssociationEnds`
Text box. Lists any AssociationEnds (see Section 18.13, “Association End”) connected to this interface.

Note

Associations between classes and interfaces *must* be navigable *only* from the class to the interface. ArgoUML will create associations between classes and interfaces with the correct navigability, but does not prevent the user from altering this.

Button 1 double click navigates to the selected entry.

`Operations`
Text area. Lists all the operations (see Section 18.8, “Operation”) defined on this interface. Button 1 double click navigates to the selected operation. Button 2 click will show a popup menu with two items: `Move Up` and `Move Down`, which allow reordering the operations.

Caution

All operations on an interface *must* be public. The ArgoUML critics will complain if this is not the case.

18.17. Abstraction

An abstraction is a dependency relationship joining two model elements within the model at different levels of abstraction. Within ArgoUML it is principally used through its specific stereotype `realize` to define realization dependencies, which link model elements that *specify* behavior to the corresponding model elements that *implement* the behavior.

In the UML metamodel `Abstraction` is a sub-class of `Dependency` and through that `Relationship`.

An abstraction with stereotype `realize` is represented on a class diagram as a dotted line with a solid white head at the specifying end.

Caution

All other stereotypes of abstraction should be represented using an open arrow head, but this is not supported by ArgoUML.

18.17.1. Abstraction Details Tabs

The details tabs that are active for abstractions are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 18.17.2, “Abstraction Property Toolbar” and Section 18.17.3, “Property Fields For Abstraction” below.

`Documentation`
: Standard tab. See Section 13.4, “Documentation Tab”.

`Presentation`
: Standard tab.

Note

The values for the bounds of the abstraction are downlighted, since the association is tied to particular model elements.

`Source`
: Standard tab. This contains the single downlighted text `N/A`.

`Tagged Values`
: Standard tab. In the UML metamodel, `Abstraction` has the following standard tagged values defined.

- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the abstraction is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived abstractions still have their value in analysis to introduce useful names or concepts, and in design to avoid re-computation.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

18.17.2. Abstraction Property Toolbar

 `Go up`

Navigate up through the package structure.

 `Delete`

This deletes the abstraction from the model

Warning

This is a deletion from the model *not* just the diagram. To delete an abstraction from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

18.17.3. Property Fields For Abstraction

`Name`
Text box. The name of the abstraction. There are no constraints on the name of an abstraction, which is not shown on any diagram.

`Stereotype`
Drop down selector. Abstraction is provided by default with the UML standard stereotypes `derive`, `realize`, `refine` and `trace`.

Caution

ArgoUML automatically selects the stereotype realize when an abstraction is created. The user is free to change the stereotype to use the abstraction to indicate for example a `trace` relationship. However ArgoUML will not alter the representation on the diagram accordingly.

`Navigate Stereotype`
 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 16.6, “Stereotype”).

`Namespace`
Drop down selector. Records and allows changing the namespace for the abstraction. This is the package hierarchy.

`Suppliers`
Text area. Lists the model element that is the supplier end of this abstraction (for a realization this is

the end providing the implementation).

Note

Although this is a text area there is no mechanism for adding more than one supplier.

Button 1 double click navigates to the selected entry.

`Clients`
Text area. Lists the model element that is the client end of this abstraction (for a realization this is the end providing the specification).

Note

Although this is a text area there is no mechanism for adding more than one client.

Button 1 double click navigates to the selected entry.

Chapter 19. Sequence Diagram Model Element Reference

19.1. Introduction

This chapter describes each model element that can be created within a sequence diagram. Note that some sub-model elements of model elements on the diagram may not actually themselves appear on the diagram.

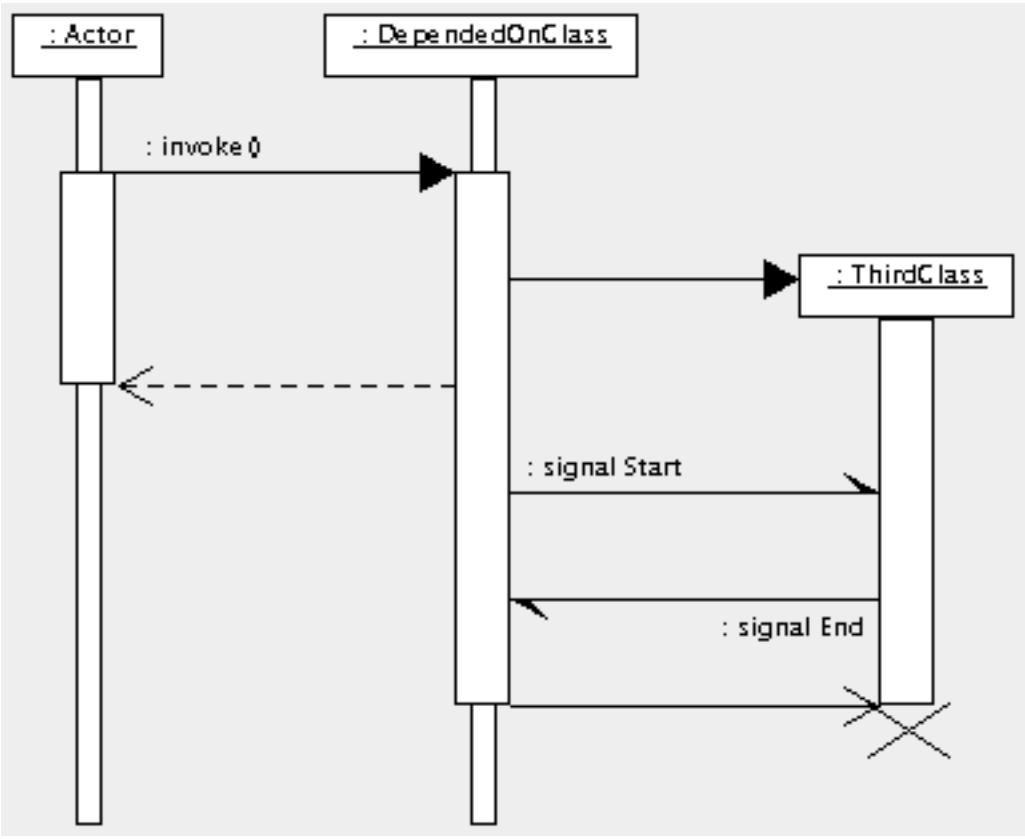
There is a close relationship between this material and the `Properties` tab of the details pane (see Section 13.3, “Properties Tab”). That section covers properties in general, in this chapter they are linked to specific model elements.

Caution

Sequence diagrams are not fully developed yet in ArgoUML. Many aspects are not fully implemented, or may not behave as expected.

Figure 19.1, “Possible model elements on a sequence diagram.” shows a sequence diagram with all possible model elements displayed.

Figure 19.1. Possible model elements on a sequence diagram.



19.1.1. Limitations Concerning Sequence Diagrams in ArgoUML

The sequence diagram is still rather under-developed in ArgoUML.

The biggest difficulties are with the actions behind the stimuli. These are purely textual in implementation, and there is no way to link them back to their associated operations or signals.

19.2. Object

An object is an instance of a class. In the UML metamodel `Object` is a sub-class of `Instance`. Within a sequence diagram objects may be used to represent a specific instance of a class. Unlike collaboration diagrams (see Chapter 21, *Collaboration Diagram Model Element Reference*), sequence diagrams cannot show generic behavior between classifier roles.

An object is represented on a sequence diagram in ArgoUML as a plain box labeled with the object name (if any) and class name, separated by a colon (:). As links with stimuli to and from other objects are added, a time line grows down from the object. This is thin where the object does not have control and thick where it does.

Caution

The current release of ArgoUML shows interactions between objects, although the UML standard for sequence diagrams is for interaction between instances of any classifier).

However the actual implementation in ArgoUML permits any classifier to be used with the object, and so the diagram can successfully represent instances of actors for example as well as classes.

19.2.1. Object Details Tabs

The details tabs that are active for objects are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 19.2.2, “Object Property Toolbar” and Section 19.2.3, “Property Fields For Object” below.

`Documentation`
: Standard tab.

`Presentation`
: Standard tab. The values for the bounds of the object notionally define the bounding box of the object and its time line. However if you change them it will have no effect, and the original values will be reset when you next revisit the tab.

`Source`
: Standard tab, but with no contents.

Caution

An object should not generate any code, so having this tab active is probably a mis-

take.

`Tagged Values`
Standard tab. In the UML metamodel, `Object` has the following standard tagged values defined.

- `persistence` (from the superclass, `Instance`. Showing the permanence of the state information associated with the object. Values `transitory` (state is destroyed when the object is destroyed) and `persistent` (state is preserved when the object is destroyed).
- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the object is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived objects still have their value in analysis and design to introduce useful names or concepts, and in design to avoid re-computation.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

`Checklist`
Standard tab for a Classifier.

19.2.2. Object Property Toolbar


`Go up`

Navigate up through the package structure.


`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected object, navigating immediately to the properties tab for that stereotype.


`Delete`

This deletes the object from the model

Warning

This is a deletion from the model *not* just the diagram. To delete an object from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

19.2.3. Property Fields For Object

`Name`
Text box. The name of the object. By convention object names start with a lower case letter and use bumpy caps to divide words within the name.

Note

ArgoUML does not enforce this naming convention.

`Stereotype`
Drop down selector. Object has no stereotypes by default in the UML standard.

`Navigate Stereotype`
 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 18.5, “Stereotype”).

`Namespace`
Text box. Records the namespace for the object. This is the package hierarchy.

`Stimuli Sent`
Text area. Lists the stimuli sent to this object.

`Stimuli Received`
Text area. Lists the stimuli received by this object.

`Classifier`
Drop down selector. The name of the classifier of which this is an object.

Caution

In the current release of ArgoUML the drop down selector will include *all* classifiers (i.e. interfaces, actors, use cases and datatypes as well), which is what is wanted on the diagram, although it should properly be called an instance, rather than an object. In practice only instances of classes and actors make much sense.

Note

In the current release of ArgoUML the same graphical presentation is used, even if the object is actually representing an instance of an actor (when a stick-man would be more usual).

19.3. Stimulus

A stimulus is a communication between two instances and is generated by an action. On a sequence diagram a stimulus is associated with a link—an instance of an association linking two object instances. In the UML metamodel `Stimulus` is a sub-class of `ModelElement`.

The link (see Section 19.9, “Link”) associated with a stimulus is represented on a sequence diagram in ArgoUML as an arrow between the time lines of the object instances (or the object head in the case of stimulus create, described below) labeled with the name of the action (if any), and the action, separated by a colon (:). The type of line and arrowhead depends on the type of action that generated the stimulus:

- `Stimulus Call`. Generated by a call action, itself the result of an operation of a class. Shown as a solid line with a solid arrowhead to the time line of the object instance receiving the stimulus.
- `Stimulus Create`. Generated by a create action for the class for which an instance is to be created Shown as a solid line with a solid arrowhead to the object head of the object instance being created.
- `Stimulus Destroy`. Generated by a destroy action of the originating object. Shown as a solid line with an open arrowhead terminating in a diagonal cross at the end of the time line of the receiving (destroyed) object instance.
- `Stimulus Send`. Generated by a send action, the result of a signal raised by an operation of the sending object instance and handled by the receiving object instance. Shown as a solid line with half an open arrowhead.
- `Stimulus Return`. Generated by an object instance that has received an earlier call stimulus and is returning a result to the calling object instance. Shown as a dotted line with an open arrowhead.

Note

ArgoUML does not allow you to create stimuli directly, but instead provides tools to create stimuli of each of the five types above.

Caution

In the current release of ArgoUML there is no way to show a terminate action where an object instance destroys itself. One way is to draw a destroy action that loops back to the object itself, give it an action with no name and use the style tab to set an invisible line, but this still leaves the arrow head showing, which is unsightly. It is also semantically incorrect anyway to use a destroy action to represent a terminate action.

19.3.1. Stimulus Details Tabs

The details tabs that are active for stimuli are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 19.3.2, “Stimulus Property Toolbar” and Section 19.3.3, “Property Fields For Stimulus” below.

`Documentation`
: Standard tab.

`Style`
: Standard tab. The values for the bounds of the stimulus notionally define the bounding box of the stimulus and its time line. However if you change them it will have no effect, and the original values will be reset when you next revisit the tab.

  Altering the `Fill` and `Shadow` entries has no effect. Rather bizarrely you can set the `Line` entry and it will draw a line around the signal, which is not a standard UML representation.

Tip

To change the color of the line, you should select the associated link (click on it a little way from the stimulus) and use its style tab (see Section 19.9, “Link”).

Caution

In the current release of ArgoUML changing the values of the `Bounds` field is possible, but will make only a temporary change to the position of the stimulus. Selecting any model element on the screen causes the stimulus to return to its original position and the original values to be restored.

`Source`
Standard tab, but with no contents.

Caution

A stimulus should not generate any code, so having this tab active is probably a mistake.

`Constraints`
Standard tab. ArgoUML only supports constraints on Classes and Features (Attributes, Operations, Receptions, and Methods), so this tab is grayed out.

`Tagged Values`
Standard tab. In the UML metamodel, `Stimulus` has the following standard tagged values defined.

- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the stimulus is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived stimuli still have their value in analysis and design to introduce useful names or concepts, and in design to avoid re-computation.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

19.3.2. Stimulus Property Toolbar


`Go up`

Navigate up through the package structure.


`Delete`

This deletes the stimulus from the model

Warning

This is a deletion from the model *not* just the diagram. To delete an stimulus from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

19.3.3. Property Fields For Stimulus

`Name`
Text box. There is no convention for naming stimuli, and it is quite normal to leave them unnamed. The action is sufficient identification.

Tip

It is sometimes useful to give simple names to stimuli, so they can be referred to in attached notes giving timing constraints.

`Action`
Text box. This is used to identify the action that generated the stimulus.

Caution

The current release of ArgoUML only implements actions as textual descriptions.

As a practical convention it is suggested that call actions are shown as the name of the operation generating the action with any arguments in parentheses and that send actions are shown as the name of the signal generating the action with any arguments in parentheses. Return actions should be shown as the expression for the value they return, or empty otherwise. Create and destroy actions should be left empty, since they are implied by their representation.

`Stereotype`
Drop down selector. Stimulus has no stereotypes by default in the UML standard, but ArgoUML provides the stereotypes, `machine`, `organization` and `person`.

Caution

ArgoUML also provides the stereotype `realize` for stimuli. This appears to be an error, since this stereotype properly belongs to the `Abstraction` metaclass.

`Navigate Stereotype`
 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 18.5, “Stereotype”).

`Sender`
Text box. Identifies the instance which sent this stimulus.

Button 1 click navigates to the sender instance, button 2 gives a pop up menu with one entry.

- `Open`. Navigate to the selected sender instance.

`Receiver`
Text box. Identifies the instance which receives this stimulus.

Button 1 click navigates to the receiver instance, button 2 gives a pop up menu with one entry.

- `Open`. Navigate to the selected receiver instance.

Warning

In the current release of ArgoUML this field is broken. It always shows the entry `none` and the pop-up menu is grayed out.

`Namespace`
Text box. Records the namespace for the stimulus. This is the package hierarchy.

Button 1 click on the entry will navigate to the package defining this namespace (or the model for the top level namespace).

19.4. Stimulus Call

This tool creates a stimulus associated with a call action on the diagram, creating at the same time the associated link between sender and receiving instances.

All details tabs and properties are identical to to that of stimulus in general (see Section 19.3, “Stimulus”). Its graphical representation on the diagram is that of a stimulus associated with a call action, i.e. a solid line with a solid arrow head.

Note

Because the current release of ArgoUML does not fully implement actions, there is no enforcement of the relationship to a call action.

19.5. Stimulus Create

This tool creates a stimulus associated with a create action on the diagram, creating at the same time the associated link between sender and receiving instances.

All details tabs and properties are identical to to that of stimulus in general (see Section 19.3, “Stimulus”). Its graphical representation on the diagram is that of a stimulus associated with a create action, i.e. a solid line with a solid arrow head terminating at the head of the created instance.

Note

Because the current release of ArgoUML does not fully implement actions, there is no enforcement of the relationship to a create action.

19.6. Stimulus Destroy

This tool creates a stimulus associated with a destroy action on the diagram, creating at the same time the associated link between sender and receiving instances.

All details tabs and properties are identical to to that of stimulus in general (see Section 19.3, “Stimulus”). Its graphical representation on the diagram is that of a stimulus associated with a destroy action, i.e. a solid line with an open arrow head terminating at a cross at the bottom of the destroyed instance's time line.

Note

Because the current release of ArgoUML does not fully implement actions, there is no enforcement of the relationship to a destroy action.

19.7. Stimulus Send

This tool creates a stimulus associated with a send action on the diagram, creating at the same time the associated link between sender and receiving instances.

All details tabs and properties are identical to to that of stimulus in general (see Section 19.3, “Stimulus”). Its graphical representation on the diagram is that of a stimulus associated with a send action, i.e. a solid line with half an open arrow head.

Note

Because the current release of ArgoUML does not fully implement actions, there is no enforcement of the relationship to a send action.

19.8. Stimulus Return

This tool creates a stimulus associated with a return action on the diagram, creating at the same time the associated link between sender and receiving instances.

All details tabs and properties are identical to to that of stimulus in general (see Section 19.3, “Stimulus”). Its graphical representation on the diagram is that of a stimulus associated with a return action, i.e. a dotted line with an open arrow head.

Note

Because the current release of ArgoUML does not fully implement actions, there is no enforcement of the relationship to a return action.

19.9. Link

A link is an instance of an association. In the UML metamodel `Link` is a sub-class of `Instance`. Within a sequence diagram links are created indirectly when an associated stimulus is created.

An link is represented on a sequence diagram in ArgoUML as a line connecting the instances concerned. However on a sequence diagram the representation is modified to reflect the type of action associated with the stimulus carried on the link (see Section 19.3, “Stimulus”).

19.9.1. Link Details Tabs

The details tabs that are active for links are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 19.9.2, “Link Property Toolbar” and Section 19.9.3, “Property Fields For Link” below.

`Documentation`
Standard tab.

`Presentation`
Standard tab. The values for the bounds of the link are downlighted, since they are determined by the objects connected.

`Source`
Standard tab, but with no contents.

Caution

A link should not generate any code, so having this tab active is probably a mistake.

`Tagged Values`
Standard tab. In the UML metamodel, `Link` has the following standard tagged values defined.

- `persistence` (from the superclass, `Instance`. Showing the permanence of the state information associated with the link. Values `transitory` (state is destroyed when the link is destroyed) and `persistent` (state is preserved when the link is destroyed).
- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the link is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived links still have their value in analysis and design to introduce useful names or concepts, and in design to avoid re-computation.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

`Checklist`
Standard tab for a Classifier.

19.9.2. Link Property Toolbar


`Go up`

Navigate up through the package structure.


`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected link, navigating immediately to the properties tab for that stereotype.


`Delete`

This deletes the link from the model

Warning

This is a deletion from the model *not* just the diagram. To delete an link from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

19.9.3. Property Fields For Link

`Name`
Text box. The name of the link. By convention link names start with a lower case letter and use bumpy caps to divide words within the name.

Note

ArgoUML does not enforce this naming convention.

`Stereotype`
Drop down selector. Link has no stereotypes by default in the UML standard.

`Navigate Stereotype`
 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 18.5, “Stereotype”).

`Namespace`
Text box. Records the namespace for the link. This is the package hierarchy.

`Connections`
List box. Lists the connections of the link, i.e. the link-ends.

Button 1 double click on the entry will navigate to the link-end.

Chapter 20. Statechart Diagram Model Element Reference

20.1. Introduction

This chapter describes each model element that can be created within a statechart diagram. Note that some sub-model elements of model elements on the diagram may not actually themselves appear on the diagram.

There is a close relationship between this material and the Properties Tab of the Details Pane (see Section 13.3, “Properties Tab”). That section covers Properties in general, in this chapter they are linked to specific model elements.

Figure 20.1, “Statechart diagram model elements 1.” and Figure 20.2, “Statechart diagram model elements 2.” show statechart diagrams with most possible model elements displayed.

Figure 20.1. Statechart diagram model elements 1.

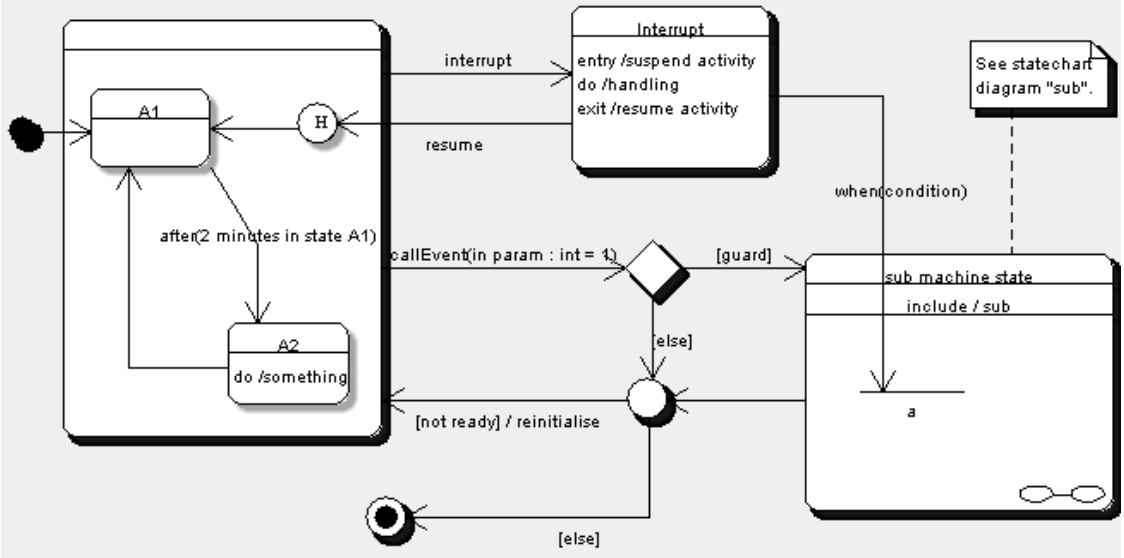
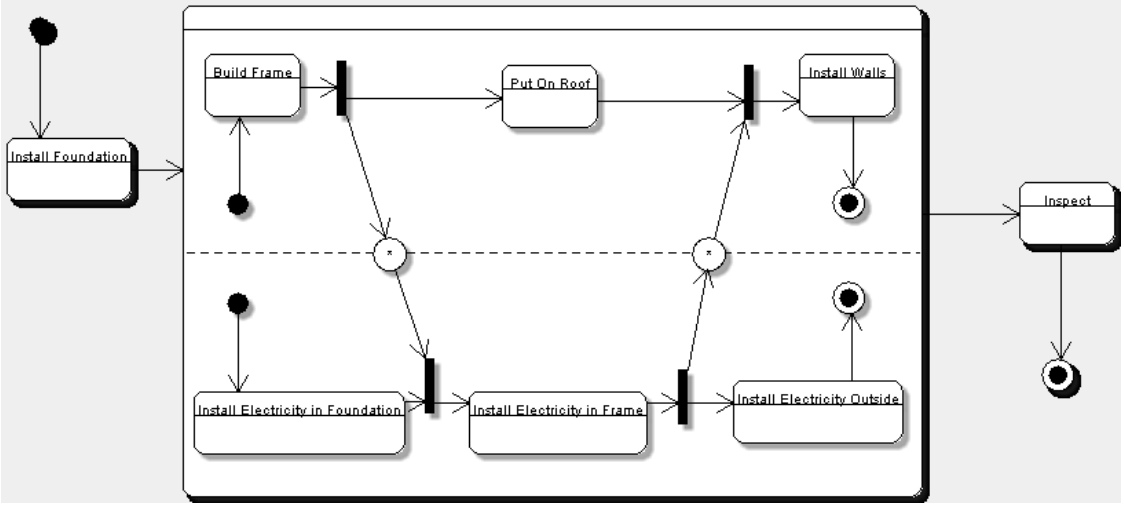


Figure 20.2. Statechart diagram model elements 2.



20.1.1. Limitations Concerning Statechart Diagrams in ArgoUML

The statechart diagrams support the 7 action types defined (CallAtion, CreateAction, DestroyAction, ReturnAction, SendAction, TerminateAction and UninterpretedAction), but there is no way to use the same action more than once. Also, in a few cases, it is not possible to set or select the related elements; e.g.there is no way to select a signal for a SendAction.

Code generation from statechart diagrams is not developed yet.

20.2. State

A state models a situation during which some (usually implicit) invariant condition holds for the parent class. This invariant may be a static situation such as an object waiting for some external event to occur, or some dynamic activity “in progress”.

A state is represented on a statechart diagram in ArgoUML as a rectangle with rounded corners, with a horizontal line separating the name at the top from the description of the behavior below. The description of the behavior includes the entry and exit actions and any internal transitions.

20.2.1. State Details Tabs

The details tabs that are active for states are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 20.2.2, “State Property Toolbar” and Section 20.2.3, “Property Fields For State” below.

`Documentation`
: Standard tab.

`Presentation`
: Standard tab. The values for the bounds of the state define the bounding box of the state.

`Stereotype`
Standard tab.

`Tagged Values`
Standard tab.

20.2.2. State Property Toolbar

 `Go up`

Navigate up through the package structure.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected state, navigating immediately to the properties tab for that stereotype.

 `Delete`

This deletes the state from the model

Note

This is a deletion from the model, *not* just the diagram. You can not just remove a state from the diagram, and keep it within the model, as is possible in other diagrams.

20.2.3. Property Fields For State

`Name`
Text box. The name of the state. By convention state names start with a lower case letter and use bumpy caps to divide words within the name.

Note

ArgoUML does not enforce this naming convention.

`Container`
Text box. Shows the container of the state. This is the state hierarchy.

Button 1 double click on the entry will navigate to the composite state that contains this state. All states are at least contained by the otherwise hidden top-level state (named “top”) that is the root of the state containment hierarchy.

`Entry-Action`
Text box. Shows the name of the action (if any) to be executed on entry to this state.

Note

This field shows the name of the action, while on the diagram the expression of the action is shown.

Button 1 double-click navigates to the selected entry, button 2 gives a pop up menu with two entries:

- `New`. Add a new Entry action of a certain kind. This menu has the following submenus to select the kind of action: Call Action, Create Action, Destroy Action, Return Action, Send Action, Terminate Action, Uninterpreted Action.
- `Delete From Model`. Delete the Entry-Action.

`Exit-Action`
Text box. Shows the action (if any) to be executed on exit from this state.

Button 1 click navigates to the selected action, button 2 gives a pop up menu with two entries.

- `New`. Add a new Exit action of a certain kind. This menu has the following submenus to select the kind of action: Call Action, Create Action, Destroy Action, Return Action, Send Action, Terminate Action, Uninterpreted Action.
- `Delete From Model`. Delete the Exit-Action.

`Do-Activity`
Text box. Shows the action (if any) to be executed while being in this state.

Button 1 click navigates to the selected action, button 2 gives a pop up menu with two entries.

- `New`. Add a new Do-Activity (action) of a certain kind. This menu has the following submenus to select the kind of action: Call Action, Create Action, Destroy Action, Return Action, Send Action, Terminate Action, Uninterpreted Action.
- `Delete From Model`. Delete the Do-Activity.

`Deferrable Events`
Text box. Shows a list of events that are candidates to be retained by the state machine if they trigger no transitions out of the state (not consumed).

Button 1 click navigates to the selected event, button 2 on an event gives a pop up menu with the following entries.

- `Select`. Allows to add already existing events to the list of deferred ones.
- `New`. Add a new event of a certain kind. This menu has the following submenus to select the kind of event: Call Event, Change Event, Signal Event, Time Event.
- `Delete From Model`. Delete the event.

`Incoming`
Text area. Lists all the transitions that enter this state.

Button 1 double click navigates to the selected entry.

`Outgoing`
Text area. Lists all the transitions that leave this state.

Button 1 double click navigates to the selected action.

`Internal Transitions`
Text area. Lists all the internal transitions of the state. Such transitions neither exit nor enter the state, so they do not cause a state change. Which means that the Entry and Exit actions are not invoked.

Note

This field shows the name of the transition, while on the diagram the name of the trigger is shown, separated with a `/` from the effect script.

Button 1 double-click navigates to the selected transition, button 2 gives a pop up menu with one entry.

- `New`. Add a new internal transition.

20.3. Action

An action specifies an executable statement and is an abstraction of a computational procedure that can change the state of the model. In the UML metamodel it is a child of `ModelElement`. Since in the metamodel an ActionSequence is itself an Action that is an aggregation of other actions (i.e. the "composite" pattern), an ActionSequence may be used anywhere an action may be.

There are a number of different types of action that are children of Action within the UML metamodel.

- `CreateAction`. Associated with a classifier, this action creates an instance of that classifier.
- `CallAction`. Associated with an operation, this action calls the given operation.
- `ReturnAction`. An action used to return a result to an earlier caller.
- `SendAction`. Associated with a signal, this action causes the signal to be raised.
- `TerminateAction`. Causes the invoking object to self-destruct.
- `UninterpretedAction`. An action used to specify language-specific actions that do not classify under the other types of actions.
- `DestroyAction`. Destroys the specified target object.

An action is represented on the diagram by the text of its expression.

Caution

The V0.20 release of ArgoUML only partially implements actions. As a practical convention it is suggested that call actions are shown as the name of the operation generating the action with any arguments in parentheses and that send actions are shown as the name of the signal generating the action with any arguments in parentheses. Return actions should be shown as the expression for the value they return, or empty otherwise. Create and destroy actions should shown as `create(<target>)` and `destroy(<target>)`. Terminate action should be shown as `terminate`.

20.3.1. Action Details Tabs

The details tabs that are active for actions are as follows.

`ToDoItem`

Standard tab.

`Properties`
See Section 20.3.2, “Action Property Toolbar” and Section 20.3.3, “Property Fields For Action” below.

`Documentation`
Standard tab.

`Stereotype`
Standard tab. In the UML metamodel, `Action` has no standard stereotypes defined.

`Tagged Values`
Standard tab. In the UML metamodel, `Action` has no standard tagged value defined.

20.3.2. Action Property Toolbar

 `Go up`

Navigate up through the hierarchical structure.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected action, navigating immediately to the properties tab for that stereotype.

 `Delete`

This deletes the Action from the model

20.3.3. Property Fields For Action

`Name`
Text box. The name of the action. By convention action names start with a lower case letter and use bumpy caps to divide words within the name.

Note

ArgoUML does not enforce this naming convention.

`Asynchronous`
Check box. Indicates if a dispatched Stimulus is asynchronous or not.

`Script`
Double text box with the expression that defines the action. This field consists of two parts, the first one contains the body (script) of the expression, and the second one contains the particular programming language used to write the expression.

`Recurrence`
Double Text box. An expression stating how many times the Action should be performed. The field consists of two parts: the first one for the expression, the second one for the language it is written in.

`Arguments`

Text box. This is an ordered list with the arguments of the action.

Button 1 double-click on any of the arguments navigates to that argument, button 2 click brings up a pop up menu with two entries.

- `New`. Create a new argument and navigate to it.
- `Remove`. Deletes the argument from the model.

`Instantiation (only for CreateAction)`
Text box. This shows the classifier that gets instantiated by the create-action.

Button 1 double-click on the classifier navigates to that argument, button 2 click brings up a pop up menu with one entry.

- `Add...`. This brings up a dialog box that allows selecting the one classifier that gets created.

20.4. Composite State

A composite state is a state that contains other states (known as sub-states), allowing hierarchical state machines to be constructed.

A composite state is represented on a statechart diagram in ArgoUML as a large rectangle with rounded corners, with a horizontal line separating the name at the top from the description of the behavior and the model of the sub-state machine below. The description of the behavior includes the entry, exit and do actions and any internal transitions.

Sub-states are placed within a composite machine by placing them entirely within the composite state. This can be done at creation time, i.e. when creating the state for the first time in the editing pane. Alternatively, an existing state can be dragged onto a composite state.

The description of a composite state is almost identical to that of a state (see Section 20.2, “State” and so is not duplicated here. The only differences is one additional tool, one missing field, and one additional field, which are described as follows.


`New Concurrent Region`
Adds a new concurrent region to the selected composite state.

`Deferrable Events`
This field is missing from V0.20 of ArgoUML.

`Subvertices`
Text area. Lists all the sub-states contained within this composite state.

Button 1 double-click navigates to the selected entry, button 2 gives a pop up menu with two entries.

- `New`. A submenu pops up, with a selection of 7 kinds of states, which can be added to the model. The 7 kinds of states supported are: Pseudo State, Synch State, Stub State, Composite State, Simple State, Final State, Submachine State.

Warning

Using this way of adding states to the model is not a good idea, since you will have to add the state to the diagram later. This can be done by selecting it in the explorer, and activating the pop-up menu, and selecting “Add to Diagram”. It is

advisable to use the toolbar of the diagram instead.

- `Delete From Model` Delete the selected state from the model.

20.5. Concurrent Region

A Concurrent Region is an “orthogonal conjunctive” component of a composite state, allowing concurrency to be constructed.

A concurrent region is represented on the diagram by a tile of a composite state, separated from other regions by a dashed line.

ArgoUML currently only supports a horizontal division of a concurrent composite state in regions.

The description of the details panels of a concurrent region is identical to that of a composite state (see Section 20.4, “Composite State” and so is not duplicated here.

20.6. Submachine State

A submachine state is a syntactical convenience that facilitates reuse and modularity. It is a shorthand that implies a macro-like expansion by another state machine and is semantically equivalent to a composite state. The state machine that is inserted is called the referenced state machine while the state machine that contains the submachine state is called the containing state machine. The same state machine may be referenced more than once in the context of a single containing state machine. In effect, a submachine state represents a *call* to a state machine *subroutine* with one or more entry and exit points. The entry and exit points are specified by stub states. SubmachineState is a child of State.

The submachine state is depicted as a normal state with the additional *include* declaration above (and separated by a line from) its internal transitions compartment. The expression following the `include` reserved word is the name of the invoked submachine.

ArgoUML currently only supports a horizontal division of a concurrent composite state in regions.

The description of the details panels of a concurrent region is almost identical to that of a composite state (see Section 20.4, “Composite State” and so is not duplicated here. The only difference is one additional field:

`Submachine`
: Drop-down selector. Allows selecting the submachine included within this composite state.

20.7. Stub State

A stub state only appears on a submachine state.

A submachine state represents the invocation of a state machine defined elsewhere. In the general case, an invoked state machine can be entered at any of its substates or through its default (initial) pseudostate. Similarly, it can be exited from any substate or as a result of the invoked state machine reaching its final state. The non-default entry and exits are specified through *stub states*. In the UML metamodel, StubState is a child of State.

Every Stub State has a label on the diagram, which corresponds to the pathname represented by the "Reference State" attribute of the stub state.

The description of the details panels of a stub state is almost identical to that of a pseudo state (see Section 20.11, "Pseudostate" and so is not duplicated here. The only difference is one additional field:

`Reference State`
: Drop-down selector. Allows entering the path name of the reference state.

20.8. Transition

A transition is a directed relation between a source state (any kind, e.g. composite state) and a destination state (any kind, e.g. composite state). Within the UML metamodel, `Transition` is a sub-class of `ModelElement`.

A transition is represented on a statechart diagram in ArgoUML as a line with arrow connecting the source to the destination state. Next to this line is a string containing the following three parts: The trigger event (e.g. a Call Event), which may have parameters between brackets `()`. Next follows (if any) the guard in square brackets (`[]`). Finally, if there is an effect (e.g. Call Action) defined, a slash (`/`) followed by the expression of the action.

20.8.1. Transition Details Tabs

The details tabs that are active for transitions are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 20.8.2, "Transition Property Toolbar" and Section 20.8.3, "Property Fields For Transition" below.

`Documentation`
: Standard tab.

`Presentation`
: Standard tab. The values for the bounds of the transition are downlighted, since the position of the transition is defined by its end points.

`Stereotype`
: Standard tab. In the UML metamodel, `Transition` has no stereotypes defined by default.

`Tagged Values`
: Standard tab. In the UML metamodel, `Transition` has no standard tagged values defined.

`Checklist`
: Standard tab for a transition.

20.8.2. Transition Property Toolbar

 `Go up`

Navigate up in the hierarchy to the parent state machine.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected transition, navigating immediately to the properties tab for that stereotype.

 `Delete`

This deletes the transition from the model.

Warning

This is a deletion from the model *not* just the diagram. To delete a transition from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

20.8.3. Property Fields For Transition

`Name`
Text box. The name of the transition. By convention transition names start with a lower case letter and use bumpy caps to divide words within the name.

Note

ArgoUML does not enforce this naming convention.

`StateMachine`
Text box. Shows the name of the parent StateMachine for the transition.

Button 1 double-click navigates to the StateMachine shown.

`State`
Text box. Shows the name of the parent State in case of an internal transition.

Button 1 double-click navigates to the State shown.

`Source`
Text box. Shows the source state for the transition.

Button 1 double-click navigates to the selected entry.

`Target`
Text box. Shows the target state for the transition.

Button 1 double-click navigates to the selected entry.

`Trigger`
Text box. Shows the trigger event (if any) which invokes this transition.

Note

UML does not require there to be a trigger, e.g. when a guard is defined. In this case, the transition is taken immediately if the guard is true.

Button 1 double-click navigates to the selected entry, button 2 gives a pop up menu with three entries.

- `Select - Add...`. This Add an existing trigger event. A sub-menu opens with 4 choices: Call Event, Change Event, Signal Event, Time Event.
- `New`. Add a new trigger event. A sub-menu opens with 4 choices: Call Event, Change Event, Signal Event, Time Event.
- `Delete From Model`. Delete the trigger event from the model. This feature is always down-lighted in the current version of ArgoUML.

`Guard`
Text box. Shows the name of a guard (if any). The expression of a guard must be true before this transition can be taken.

Button 1 double-click navigates to the selected entry, button 2 gives a pop up menu with one entry.

- `New`. Add a new guard.

`Effect`
Text box. Shows the action (if any) to be invoked as this transition is taken.

Button 1 double-click navigates to the selected action, button 2 gives a pop up menu with two entries.

- `New`. Add a new Effect (action) of a certain kind. This menu has the following submenus to select the kind of action: Call Action, Create Action, Destroy Action, Return Action, Send Action, Terminate Action, Uninterpreted Action.
- `Delete From Model`. Delete the selected action from the model.

20.9. Event

An event is an observable occurrence. In the UML metamodel it is a child of `ModelElement`.

There are a number of different types of events that are children of event within the UML metamodel.

- `CallEvent`. Associated with an operation of a class, this event is caused by a call to the given operation. The expected effect is that the steps of the operation will be executed.
- `SignalEvent`. Associated with a signal, this event is caused by the signal being raised.
- `TimeEvent`. An event cause by expiration of a timing deadline.
- `ChangeEvent`. An event caused by a particular expression (of attributes and associations) becoming true.

An event is represented by its name.

20.9.1. Event Details Tabs

The details tabs that are active for events are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 20.9.2, “Event Property Toolbar” and Section 20.9.3, “Property Fields For Event” below.

`Documentation`
Standard tab.

`Stereotype`
Standard tab. In the UML metamodel, an `Event` has the following standard stereotypes defined.

- `create` (for a `CallEvent` only). Create is a stereotyped call event denoting that the instance receiving that event has just been created. For state machines, it triggers the initial transition at the topmost level of the state machine (and is the only kind of trigger that may be applied to an initial transition).
- `destroy` (for a `CallEvent` only). Destroy is a stereotyped call event denoting that the instance receiving the event is being destroyed.

`Tagged Values`
Standard tab. In the UML metamodel, an `Event` has no standard tagged values defined.

20.9.2. Event Property Toolbar

 `Go up`

Navigate up through the composition structure.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected event, navigating immediately to the properties tab for that stereotype.

 `New parameter`

This creates a new parameter for the event operation as the current parameter, navigating immediately to the properties tab for that parameter (see Section 18.9, “Parameter”).

 `Delete`

This deletes the event from the model.

20.9.3. Property Fields For Event

`Name`
Text box. The name of the event. By convention event names start with a lower case letter and use bumpy caps to divide words within the name in the same way as operations.

Note

ArgoUML does not enforce this naming convention.

Tip

For call events it makes sense to use the name of the associated operation. For signal events it make sense to use the name of the signal, prefixed by `[sig]`. For time events use the time expression, prefixed by `[time]` and for change events the change expression, prefixed by `[change]`.

`Namespace`
Text field. Shows the namespace for the event. This is the composition hierarchy.

`Parameters`
Text area, with entries for all the actual parameter values of the event (see Section 18.9, “Parameter”).

Button 1 double-click on any of the parameters navigates to that parameter, button 2 click brings up a pop up menu with one entry.

- `New Parameter`. Create a new parameter and navigate to it.

`Transition`
This shows the transition caused by the event.

button 1 double-click on the transition navigates to that transition.

`Operations`
Drop-down selector. Only present for a Call Event. This allows specifying the operation that causes the event when called.

`Signal`
Text field. Only present for a Signal Event. This allows specifying the signal that causes the event when called.

Button 1 double-click navigates to the selected signal, button 2 gives a pop up menu with two entries.

- `Add...`. This opens a dialog box that allows selecting an already existing signal.
- `New Signal`. Creates a new Signal, and navigates to it.

`When`
Double text field. Only present for a Time Event. This allows expressing the time that the event is called.

The first of the two fields is for the body of the expression, and the second one for the language in which it is written.

Warning

In ArgoUML V0.20, the properties panel of a change event lacks a field to enter the change expression.

20.10. Guard

A guard is associated with a transition. At the time an event is dispatched, the guard is evaluated, and if false, its transition is disabled. In the UML metamodel, `Guard` is a child of `ModelElement`.

A guard is shown on the diagram by the text of its expression in square brackets (`[]`).

20.10.1. Guard Details Tabs

The details tabs that are active for guards are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 20.10.2, “Guard Property Toolbar” and Section 20.10.3, “Property Fields For Guard” below.

`Documentation`
: Standard tab.

`Stereotype`
: Standard tab, containing the stereotypes for the guard. In the UML metamodel, `Guard` has no standard stereotypes defined.

`Tagged Values`
: Standard tab. In the UML metamodel, `Guard` has no standard tagged values defined.

20.10.2. Guard Property Toolbar

 `Go up`

Navigate up through the package structure.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected guard, navigating immediately to the properties tab for that stereotype.

 `Delete from Model`

This deletes the guard from the model

Warning

This is a deletion from the model, *not* just the diagram.

20.10.3. Property Fields For Guard

`Name`
: Text box. The name of the guard. By convention guard names start with a lower case letter and use

bumpy caps to divide words within the name.

Note

ArgoUML does not enforce this naming convention.

`Transition`
Text box, showing the transition that owns this guard.

Button 1 double-click on the transition navigates to that transition.

`Expression`
Text box. The expression that defines the guard.

`Language`
Text box. This indicates that the expression is written in a particular interpretation language with which to evaluate the text.

20.11. Pseudostate

A pseudostate encompasses a number of different transient vertices on a state machine diagram. They are used, typically, to connect multiple transitions into more complex state transitions paths. For example, by combining a transition entering a fork pseudostate with a set of transitions exiting the fork pseudostate, we get a compound transition that leads to a set of concurrent target states. Pseudostates do not have the properties of a full state and serve only as a connection point for transactions (but with some semantic value). Within the UML metamodel, `Pseudostate` is a sub-class of `StateVertex`.

The representation of a pseudostate on a statechart diagram in ArgoUML depends on the particular kind of pseudostate: initial, deepHistory, shallowHistory, join, fork, junction and choice. ArgoUML lets you place any pseudostate directly by tools for the specific types of pseudostate. These are described in separate sections below (see Section 20.12, “Initial State”, Section 20.14, “Junction”, Section 20.15, “Choice”, Section 20.16, “Fork”, Section 20.17, “Join”, Section 20.18, “Shallow History” and Section 20.19, “Deep History”).

20.11.1. Pseudostate Details Tabs

The details tabs that are active for pseudostates are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 20.11.2, “Pseudostate Property Toolbar” and Section 20.11.3, “Property Fields For Pseudostate” below.

`Documentation`
Standard tab.

`Presentation`
Standard tab.

`Stereotype`
Standard tab, containing the stereotypes of the pseudostate. In the UML metamodel, `PseudoState` has the no standard stereotypes defined.

`Tagged Values`
Standard tab. In the UML metamodel, `Pseudostate` has no standard tagged values defined.

20.11.2. Pseudostate Property Toolbar

 `Go up`

Navigate up through the package structure.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected pseudostate, navigating immediately to the properties tab for that stereotype.

 `Delete from Model`

This deletes the pseudostate from the model

Warning

This is a deletion from the model *not* just the diagram.

20.11.3. Property Fields For Pseudostate

`Name`
Text box. The name of the pseudostate. By convention pseudostate names start with a lower case letter and use bumpy caps to divide words within the name.

Note

ArgoUML does not enforce this naming convention.

Tip

Pseudostate names are not shown on the diagram and it is not usually necessary to give them a name.

`Container`
Text box. Shows the container of the pseudostate. This is the state hierarchy.

Button 1 double click on the entry will navigate to the composite state that contains this state (or the top-level state that is the root of the state containment hierarchy).

`Incoming`
Text area. Lists any incoming transitions for the pseudostate.

Button 1 double-click navigates to the selected transition.

`Outgoing`
Text area. Lists any outgoing transitions for the pseudostate.

Button 1 double-click navigates to the selected transition.

20.12. Initial State

The initial state is a pseudostate (see Section 20.11, “Pseudostate”) representing a source for a single transition to the *default* state of a composite state. It is the state from which any initial transition is made.

As a consequence it is not permissible to have incoming transitions. ArgoUML will not let you create such transitions, and if you import a model that has such transitions, a critic will complain.

There can be at most one initial pseudostate in a composite state, which must have (at most) one outgoing transition.

An initial state is represented on the diagram as a solid disc.

20.13. Final State

If a transition reaches a final state, it implies completion of the activity associated with that composite state, or at the top level, of the complete state machine. In the UML metamodel `FinalState` is a child of `State`.

Note

A final state is a true state (with all its attributes), *not* a pseudostate.

Completion at the top level implies termination (i.e. destruction) of the owning object instance.

The representation of a final state on the diagram is a circle with a small disc at its center.

20.13.1. Final State Details Tabs

The details tabs that are active for final states are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 20.13.2, “Final State Property Toolbar” and Section 20.13.3, “Property Fields For Final State” below.

`Documentation`
: Standard tab.

`Presentation`
: Standard tab.

`Stereotype`
: Standard tab, containing the stereotypes of the final state. In the UML metamodel, a `Final State` has the no standard tagged values defined.

`Tagged Values`
: Standard tab. In the UML metamodel, `Final State` has no standard tagged values defined.

20.13.2. Final State Property Toolbar


`Go up`

Navigate up through the package structure.


`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected state, navigating immediately to the properties tab for that stereotype.


`Delete from Model`

This deletes the final state from the model

Warning

This is a deletion from the model *not* just the diagram.

20.13.3. Property Fields For Final State

`Name`
Text box. The name of the final state. By convention final state names start with a lower case letter and use bumpy caps to divide words within the name.

Note

ArgoUML does not enforce this naming convention.

Tip

Final state names are shown on the diagram but it is not usually necessary to give them a name.

`Container`
Text box. Shows the container of the final state. This is the state hierarchy.

Button 1 double click on the entry will navigate to the composite state that contains this state (or the top-level state that is the root of the state containment hierarchy).

`Entry-Action`
Text box. Shows the name of the action (if any) to be executed on entry to this final state.

Button 1 double-click navigates to the selected entry, button 2 gives a pop up menu with two entries:

- `New`. Add a new Entry action of a certain kind. This menu has the following 7 submenus to select the kind of action: Call Action, Create Action, Destroy Action, Return Action, Send Action, Terminate Action, Uninterpreted Action.
- `Delete From Model`. Delete the Entry-Action.

`Incoming`
Text area. Lists any incoming transitions for the final state.

Button 1 double-click navigates to the selected transition.

`Internal Transitions`
Text area. Lists all the internal transitions of the state. Such transitions neither exit nor enter the state, so they do not cause a state change. Which means that the Entry and Exit actions are not invoked.

Button 1 double-click navigates to the selected transition

20.14. Junction

Junction is a pseudostate (see Section 20.11, “Pseudostate”) which is used to split an incoming transition into multiple outgoing transition segments with different guard conditions. A Junction is also called a Merge or Static conditional branch. The chosen transition is that whose guard is true at the time of the transition.

A predefined guard denoted `else` may be defined for at most one outgoing transition. This transition is enabled if all the guards labeling the other transitions are false.

According the UML standard, its symbol is a small black circle. Alternatively, it may be represented by a diamond shape (in case of "Decision" for Activity diagrams). ArgoUML only represents a junction on the diagram as a solid (white by default) diamond, and does not support the black circle symbol for a junction.

20.15. Choice

Choice is a pseudostate (see Section 20.11, “Pseudostate”) which is used to split an incoming transition into multiple outgoing transition segments with different guard conditions. Hence, a Choice allows a dynamic choice of outgoing transitions. The chosen transition is that whose guard is true at the time of the transition (if more than one is true, one is selected at random).

A predefined guard denoted `else` may be defined for at most one outgoing transition. This transition is enabled if all the guards labeling the other transitions are false.

Note

This sort of pseudostate was formerly called a `Branch` by ArgoUML.

A choice is represented on the diagram as a small solid (white by default) circle (reminiscent of a small state icon).

20.16. Fork

Fork is a pseudostate (see Section 20.11, “Pseudostate”) which splits a transition into two or more concurrent transitions.

Caution

The outgoing transitions should not have guards. However ArgoUML will not enforce this.

A fork is represented on the diagram as a solid (black by default) horizontal bar.

Tip

This bar can be made vertical by selecting the fork, and dragging with button 1 one of its corners.

20.17. Join

Join is a pseudostate (see Section 20.11, “Pseudostate”) which joins two or more concurrent transitions into a single transition.

Caution

The incoming transitions should not have guards. However ArgoUML will not enforce this.

A join is represented on the diagram as a solid (black by default) horizontal bar.

Tip

This bar can be made vertical by selecting the join, and dragging with button 1 one of its corners.

20.18. Shallow History

Shallow History is a pseudostate (see Section 20.11, “Pseudostate”) that can remember the last state of its container that was active. The history pseudostate points to its default state with a transition arrow just like the initial pseudostate does. This transition points to the substate that will become active when there is no history. When the container composite state has been active before (i.e., when there is history), the substate that was active when the container state was exited, becomes active again.

When placed within a multi-level hierarchy of composite states, the shallow history only remembers the history for states that have the same container as the history pseudostate. It does not restore substates deeper in the hierarchy then the history pseudostate itself.

A shallow history is represented on the diagram as a circle containing the letter `H`.

20.19. Deep History

Deep History is a pseudostate (see Section 20.11, “Pseudostate”) that can remember the last state of its container that was active. The history pseudostate points to its default state with a transition arrow just like the initial pseudostate does. This transition points to the substate that will become active when there is no history. When the container composite state has been active before (i.e., when there is history), the substate that was active when the container state was exited, becomes active again.

When placed within a multi-level hierarchy of composite states, the deep history remembers the history for all states recursively which are contained in the history pseudostate container. It does restore any substates no matter how deep in the hierarchy.

A deep history is represented on the diagram as a circle containing the symbols `H*`.

20.20. Synch State

A synch state is for synchronizing concurrent regions of a state machine. It is used in conjunction with forks and joins to insure that one region leaves a particular state or states before another region can enter a particular state or states. The firing of outgoing transitions from a synch state can be limited by specifying a bound on the difference between the number of times outgoing and incoming transitions have fired. In the UML metamodel `Synch` is a child of `StateVertex`.

A synch state is shown as a small circle with the upper bound inside it. The bound is either a positive integer or a star ('*') for unlimited. Synch states are drawn on the boundary between two regions when possible.

20.20.1. Synch State Details Tabs

The details tabs that are active for Synch states are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 20.20.2, “Synch State Property Toolbar” and Section 20.20.3, “Property Fields For Synch State” below.

`Documentation`
: Standard tab.

`Presentation`
: Standard tab.

`Stereotype`
: Standard tab, containing the stereotypes of the Synch state. In the UML metamodel, `Synch State` has no standard stereotypes defined.

`Tagged Values`
: Standard tab. In the UML metamodel, `Synch State` has no standard tagged values defined.

20.20.2. Synch State Property Toolbar

 `Go up`

Navigate up through the package structure.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected synch state, navigating immediately to the properties tab for that stereotype.

 `Delete from Model`

This deletes the synch state from the model

Warning

This is a deletion from the model *not* just the diagram.

20.20.3. Property Fields For Synch State

`Name`
Text box. The name of the Synch state. By convention Synch state names start with a lower case letter and use bumpy caps to divide words within the name.

Note

ArgoUML does not enforce this naming convention.

Tip

Synch state names are not shown on the diagram and it is not usually necessary to give them a name.

`Container`
Text box. Shows the container of the Synch state. This is the state hierarchy.

Button 1 double click on the entry will navigate to the composite state that contains this state (or the top-level state that is the root of the state containment hierarchy).

`Bound`
Editable text box. Shows the Bound of the Synch state. Which is a positive integer or the value *unlimited* (represented by a "*") specifying the maximal count of the SynchState. The count is the difference between the number of times the incoming and outgoing transitions of the synch state are fired.

`Incoming`
Text area. Lists any incoming transitions for the final state.

Button 1 double-click navigates to the selected transition.

`Outgoing Transitions`
Text area. Lists any outgoing transitions for the final state.

Button 1 double-click navigates to the selected transition.

Chapter 21. Collaboration Diagram Model Element Reference

21.1. Introduction

This chapter describes each model element that can be created within a collaboration diagram. Note that some sub-model elements of model elements on the diagram may not actually themselves appear on the diagram.

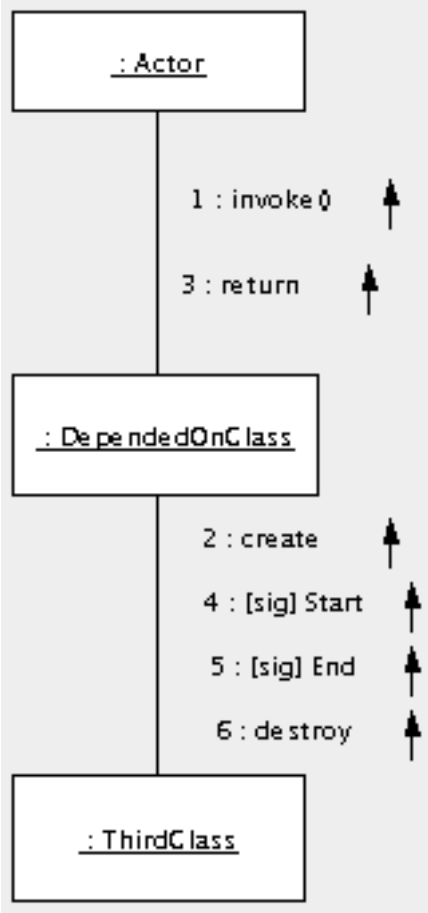
There is a close relationship between this material and the properties tab of the details pane (see Section 13.3, “Properties Tab”). That section covers Properties in general, in this chapter they are linked to specific model elements.

Caution

Collaboration diagrams are not fully developed yet in ArgoUML. Many aspects are not fully implemented, or may not behave as expected. In particular there are some serious problems with layout of the collaboration roles and messages.

Figure 21.1, “Possible model elements on a collaboration diagram.” shows a collaboration diagram with all possible model elements displayed.

Figure 21.1. Possible model elements on a collaboration diagram.



21.1.1. Limitations Concerning Collaboration Diagrams in ArgoUML

The collaboration diagram is still rather under-developed in ArgoUML. In particular there is no way to show instance collaborations (based on objects and links) rather than specification collaborations.

The biggest difficulties are with the messages. There are problems with the sequencing of the messages and their display on the diagram. The actions behind them are purely textual in implementation and there is no way to link them back to their associated operations or signals.

21.2. Classifier Role

A classifier role is a specialization of a classifier, used to show its behavior in a particular context. In the UML metamodel `Classifier Role` is a sub-class of `Classifier`. Within a collaboration diagram classifier roles may be used in one of two ways:

- To represent the classifier in a particular behavioral context (the *specification level*); or
- to specify a particular instance of the classifier (the *instance level*).

In this latter form, classifier roles are identical to the instances used in sequence diagrams (see

Chapter 19, *Sequence Diagram Model Element Reference*) and a collaboration diagram shows the same information as the sequence diagram, but in a different presentation.

Caution

A collaboration diagram should not mix classifier roles used as the specifier level and the instance level.

A classifier role is represented on a sequence diagram in ArgoUML as a plain box labeled with the classifier role name (if any) and classifier, separated by a colon (:).

Caution

A classifier role should properly also show object name (if any) preceding the classifier role name and separated from it by a slash (`/`). This allows classifier roles in a specification level diagram to be distinguished from instances in an instance level diagram.

ArgoUML does show the slash, but there is no way to define the instances.

21.2.1. Classifier Role Details Tabs

The details tabs that are active for classifier roles are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 21.2.2, “Classifier Role Property Toolbar” and Section 21.2.3, “Property Fields For Classifier Role” below.

`Documentation`
: Standard tab.

`Presentation`
: Standard tab.

`Source`
: Standard tab, but with no contents.

Caution

A classifier role should not generate any code, so having this tab active is probably a mistake.

`Tagged Values`
: Standard tab. In the UML metamodel, `Classifier Role` has the following standard tagged values defined.

- `persistence` (from the superclass, `Classifier`. Showing the permanence of the state information associated with the classifier role. Values `transitory` (state is destroyed when the classifier role is destroyed) and `persistent` (state is preserved when the classifier role is destroyed).
- `semantics` (from the superclass, `Classifier`). The value is a specification of the se-

mantics of the classifier role.

- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the classifier role is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived classifier roles still have their value in analysis and design to introduce useful names or concepts, and in design to avoid re-computation.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

21.2.2. Classifier Role Property Toolbar


`Go up`

Navigate up through the package structure.


`New reception`

This creates a new reception, navigating immediately to the properties tab for that reception.


`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected classifier role, navigating immediately to the properties tab for that stereotype.


`Delete`

This deletes the classifier role from the model

Warning

This is a deletion from the model *not* just the diagram. To delete an classifier role from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

21.2.3. Property Fields For Classifier Role

`Name`

Text box. The name of the classifier role. By convention classifier role names start with a lower case letter and use bumpy caps to divide words within the name.

Note

ArgoUML does not enforce this naming convention.

`Stereotype`
Drop down selector. Classifier Role is provided by default with the UML standard stereotypes for a classifier (`metaclass`, `powertype`, `process`, `thread` and `utility`).

`Navigate Stereotype`
 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 16.6, “Stereotype”).

`Namespace`
Text box. Records the namespace for the classifier role, which is always the containing Collaboration.

Button 1 double click on the entry will navigate to the collaboration.

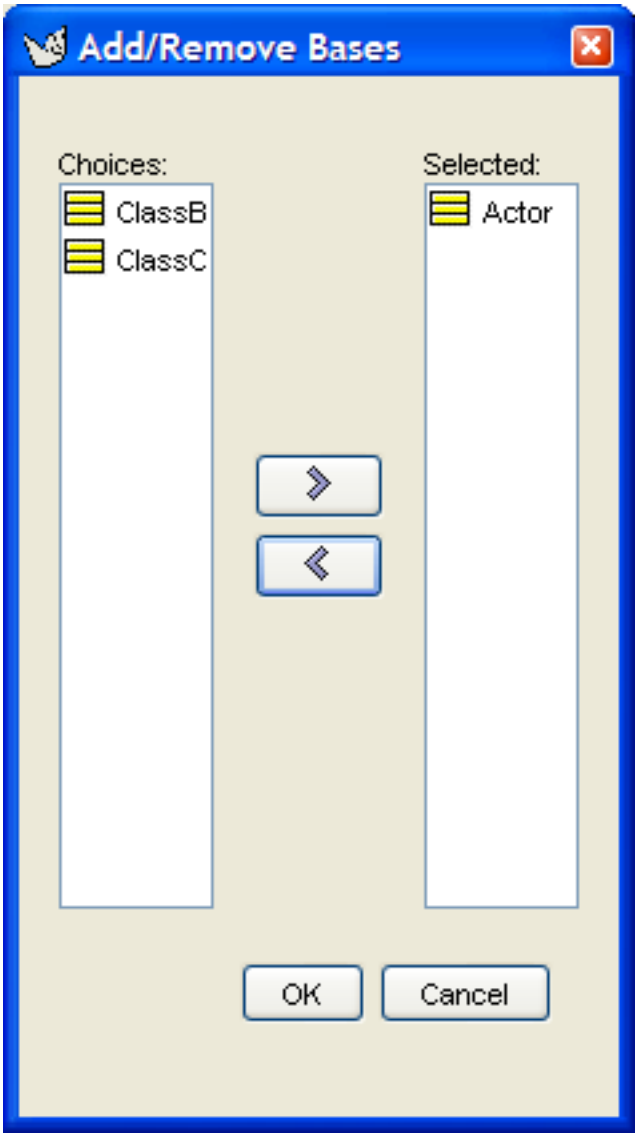
`Multiplicity`
Editable drop down selector. The default value is `*`, which means that there are any number of instances of this classifierrole that play a role in the collaboration. The drop down provides some different multiplicities. E.g. `1..1` would mean that only one instance plays a role in this collaboration.

ArgoUML does not restrict you to the predefined ranges for multiplicity. You can edit this field freely.

`Base`
List. The names of the classifiers of which this is a classifierrole. Button 1 double click navigates to the classifier. Button 2 click gives a pop up menu with the following entries.

- `Add`. Allows adding or removeing classifiers to the list. To this end, a dialog box pops up, as shown in the figure below.

Figure 21.2. The “add context” dialog box



- `Remove`. Allows removeing classifiers to the list, without making use of the dialog box.

`Generalizations`
Text area. Lists any classifierrole that *generalizes* this classifierrole.

Button 1 double click navigates to the generalization and opens its property tab.

`Specializations`
Text box. Lists any specialized classifierrole (i.e. for which this classifierrole is a generalization).

button 1 double click navigates to the generalization and opens its property tab.

`Association End Role`
Text area. Lists the association-end roles that are linked to this classifier role.

Button 1 double click navigates to the selected entry.

`Available Contents`

Text area. Lists the subset of modelelements contained in the base classifier which is used in the collaboration.

Button 1 double click navigates to the modelelement and opens its property tab.

`Available Features`
Text box. Lists the subset of features of the base classifier which is used in the collaboration.

button 1 double click navigates to the feature and opens its property tab.

21.3. Association Role

An association role is a specialization of an association, used to describe an associations behavior in a particular context. In the UML metamodel `Association Role` is a sub-class of `Association`.

An association role is represented on a collaboration diagram in ArgoUML as a line connecting the instances concerned. However on a sequence diagram the representation is modified to reflect the type of action associated with the stimulus carried on the link (see Section 19.3, “Stimulus”).

The association role is labeled with the association role name (if any).

An association role shows its name and the association name according the following syntax:

/ AssociationRoleName : AssociationName

in the same manner as a classifier role. The more generic syntax is:

I / R : C

which stands for an Instance named I originating from the Classifier C playing the role R.

21.3.1. Association Role Details Tabs

The details tabs that are active for association roles are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 21.3.2, “Association Role Property Toolbar” and Section 21.3.3, “Property Fields For Association Role” below.

`Documentation`
Standard tab.

`Presentation`
Standard tab. The values for the bounds of the association role are downlighted, since they are determined by what they connect.

`Source`
Standard tab, but with no contents.

Caution

An association role should not generate any code, so having this tab active is probably a mistake.

`Tagged Values`
Standard tab. In the UML metamodel, `AssociationRole` has the following standard tagged values defined.

- `persistence` (from the superclass, `Association`). Values `transitory`, indicating state is destroyed when an instance is destroyed or `persistent`, marking state is preserved when an instance is destroyed.
- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the association is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived association roles still have their value in analysis to introduce useful names or concepts, and in design to avoid re-computation.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

`Checklist`
Standard tab for an Association Role.

21.3.2. Association Role Property Toolbar


`Go up`

Navigate up through the package structure.


`Delete`

This deletes the association role from the model

Warning

This is a deletion from the model *not* just the diagram. To delete an association role from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

21.3.3. Property Fields For Association Role

`Name`
Text box. The name of the association role, which is shown on the diagram. By convention association role names start with a lower case letter and use bumpy caps to divide words within the name.

Note

ArgoUML does not enforce this naming convention.

`Stereotype`
Drop down selector. Association role is provided by default with the UML standard stereotype from the superclass Association: `implicit`.

`Navigate Stereotype`
 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 18.5, “Stereotype”).

`Namespace`
Text box. Records the namespace for the association role. This is the package hierarchy.

Button 1 double click on the entry will navigate to the item showm.

`Base`
Drop down selector. Records the association that is the base for the association role.

The drop down selector shows all associations that exist between the classifiers that correspond with the connected classifier roles.

`Association End Roles`
Text area. Lists the ends of this association role. An association role can have any number of ends, but two is generally the only useful number (link objects can led to a third end on instance level diagrams, but this is not supported by ArgoUML). For more on association end roles see Section 21.4, “Association End Role”.

The names are listed, unless the association end role has no name, then it is shown as `(Unnamed AssociationEndRole)`.

Button 1 double click on an association end role will navigate to that end.

`Messages`
Text area. Lists the messages that are associated with this association role.

Button 1 double click navigates to the selected entry

21.4. Association End Role

An association end role is a specialization of an association end, used to describe an association end's behavior in a particular context. In the UML metamodel `AssociationEndRole` is a sub-class of `AssociationEnd`.

Two or more association end roles are associated with each association role (see Section 21.3, “Association Role”), although for ArgoUML, the number of ends can only be two.

The association end role has no direct access on any diagram, although its stereotype, name and multiplicity is shown at the relevant end of the parent association role (see Figure 21.1, “Possible model elements on a collaboration diagram.”), and some of its properties can be directly adjusted with button 2 click. Where shared or composite aggregation is selected for one association end role, the opposite end is shown as a solid diamond (composite aggregation) or hollow diamond (shared aggregation).

Note

ArgoUML does not currently (V0.18) support showing qualifiers on the diagram, as described in the UML 1.4 standard.

Caution

An association end role should have the same, or "stricter" attribute values than its base association end. In particular its navigability should be no more general. There is as yet no critic in ArgoUML to offer advice on this rule.

21.4.1. Association End Role Details Tabs

The details tabs that are active for association end roles are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 21.4.2, "Association End Role Property Toolbar" and Section 21.4.3, "Property Fields For Association End Role" below.

`Documentation`
Standard tab.

`Source`
Standard tab. There is no code generated for an association end role.

`Tagged Values`
Standard tab. In the UML metamodel, `AssociationEndRole` has the following standard tagged values defined.

- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the association end role is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Tip

Derived association end roles still have their value in analysis to introduce useful names or concepts, and in design to avoid re-computation. However the tag only makes sense for an association end role if it is also applied to the parent association role.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

21.4.2. Association End Role Property Toolbar

 `Go up`

Navigate up to the association role to which this end role belongs.

 `Go Opposite`

This navigates to the other end of the association role.


`New Qualifier`

This creates a new Qualifier for the selected association-end role, navigating immediately to the properties tab for that qualifier.

Warning

Qualifiers are only partly supported in ArgoUML V0.18. Hence, activating this button creates a qualifier in the model, which is not shown on the diagram. Also, the properties panel for a qualifier equals that of a regular attribute.

`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected association-end role, navigating immediately to the properties tab for that stereotype.


`Delete`

This deletes the selected association-end from the model.

Note

This button is downlighted for binary association roles, since an association needs at least *two* ends. Only for N-ary associations, this button is accessable, and deletes just one end from the association.

21.4.3. Property Fields For Association End Role

`Name`
Text box. The name of the association end role, which provides a *role name* for this end of the association role. This role name can be used for navigation, and in an implementation context, provides a name by which the source end of an association role can reference the target end.

Note

ArgoUML does not enforce any naming convention for association end roles.

`Stereotype`
Drop down selector. Association end role is provided by default with the UML standard stereotypes for AssociationEndRole (`association`, `global`, `local`, `parameter`, `self`).

`Navigate Stereotype`

icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 16.6, “Stereotype”).

`Base`
Text field that shows the name of the corresponding association end. Button 1 double click navigates to the association end.

`AssociationRole`
Text box. Records the parent association role for this association end role. Button 1 double click navigates to the association role.

`Type`
Drop down selector providing access to all standard UML types provided by ArgoUML and all new classes created within the current model.

This is the type of the entity attached to this end of the association role.

`Multiplicity`
Editable drop down text entry. Allows to alter the multiplicity of this association end role (with respect to the other end), i.e. how many instances of this end may be associated with an instance of the other end. The multiplicity is shown on the diagram at that end of the association role.

`All remaining properties`
See Section 18.13.3, “Property Fields For Association End” . Since these are completely equal to the fields of an association end, they are not repeated here.

21.5. Message

A message is a communication between two instances of an association role on a specification level collaboration diagram. It describes an action which will generate the stimulus associated with the message. On a collaboration diagram a message is associated with an association role. In the UML metamodel `Message` is a sub-class of `ModelElement`.

The message is represented on a collaboration diagram in ArgoUML by its sequence number separated by a colon from the expression defining the associated action. It is accompagnied by an arrow pointing in the direction of the communication, i.e. the direction of the AssociationRole. By convention the name of a message is not shown on the diagram. Instead the diagram displays the message sequence number, either as an integer or as a decimal number to show hierarchy.

Warning

The current release of ArgoUML does not retaining message positioning after reloading the project, i.e. as if the positions were not stored in the project file.

21.5.1. Message Details Tabs

The details tabs that are active for messages are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 21.5.2, “Message Property Toolbar” and Section 21.5.3, “Property Fields For Message” below.

`Documentation`
Standard tab.

`Presentation`
Standard tab. The values for the bounds of the message define the bounding box of the message. The `Line` field defines the arrow color. Increasing the `Shadow` size has an esthetically question-

able effect.

Caution

In the V0.18 release of ArgoUML changing the position of the message by editing the values of the `Bounds` field is possible, but will make only a temporary change to the position of the message, as described above.

`Source`
Standard tab, showing the message number and action expression separated by a colon (when UML 1.4 is selected in the drop-down).

Caution

A message probably should not generated any code of itself. That should be left to the action and possibly stimulus associated with it. In any case changes to this tab are ignored.

`Tagged Values`
Standard tab. In the UML metamodel, `Message` has the following standard tagged values defined.

- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the message is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived messages still have their value in analysis and design to introduce useful names or concepts, and in design to avoid re-computation.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

21.5.2. Message Property Toolbar


`Go up`

Navigate up through the package structure.


`New Action`

This creates a new Action (see Section 20.3, “Action”) for the selected object, navigating immediately to the properties tab for that action.


`New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected message, navigating immediately to the properties tab for that stereotype.


`Delete`

This deletes the message from the model

Warning

This is a deletion from the model *not* just the diagram. To delete an message from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

21.5.3. Property Fields For Message

`Name`
Text box. The name of a message is usually its sequence number, either an integer, or a decimal (allowing alternative message hierarchies to be clearly described). ArgoUML will supply an integer sequence number by default.

`Stereotype`
Drop down selector. Message has no stereotypes by default in the UML standard.

`Navigate Stereotype`
 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 16.6, “Stereotype”).

`Interaction`
Text box. Records the Interaction of which the message is a part.

Button 1 double click on the entry will navigate to the interaction.

`Sender`
Text box. Identifies the classifier role which sent this message.

Button 1 double click navigates to the sender classifier role.

`Receiver`
Text box. Identifies the classifier role which receives this message.

Button 1 double click navigates to the receiver classifier role.

`Activator`
Drop down selector. Identifies the message which invokes the behavior that causes the sending of this message.

Button 1 click allows selecting the message.

`Action`
Text box. Lists the action (see Section 20.3, “Action”) this message invokes to raise a stimulus.

Button 1 double click navigates to the selected action, button 2 gives a pop up menu with the following entry.

- `New`. Add a new action.

  This item is downlighted if an action already exists.

`Predecessors`

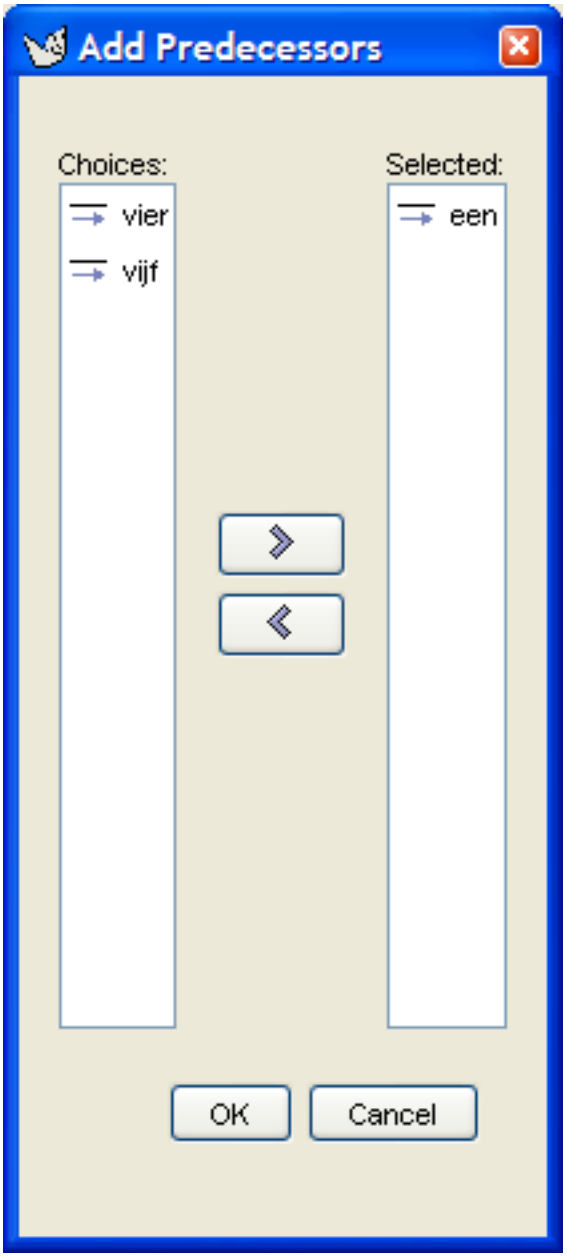
Text area. Identifies the messages, the completion of whose execution enables this message.

Button 1 double click navigates to the selected message, button 2 gives a pop up menu with one entry.

- `Add`. Opens a dialog box that allows to select any number of messages. See figure below.

  This entry is grayed out when no messages exist.

Figure 21.3. The “add predecessors” dialog box



Chapter 22. Activity Diagram Model Element Reference

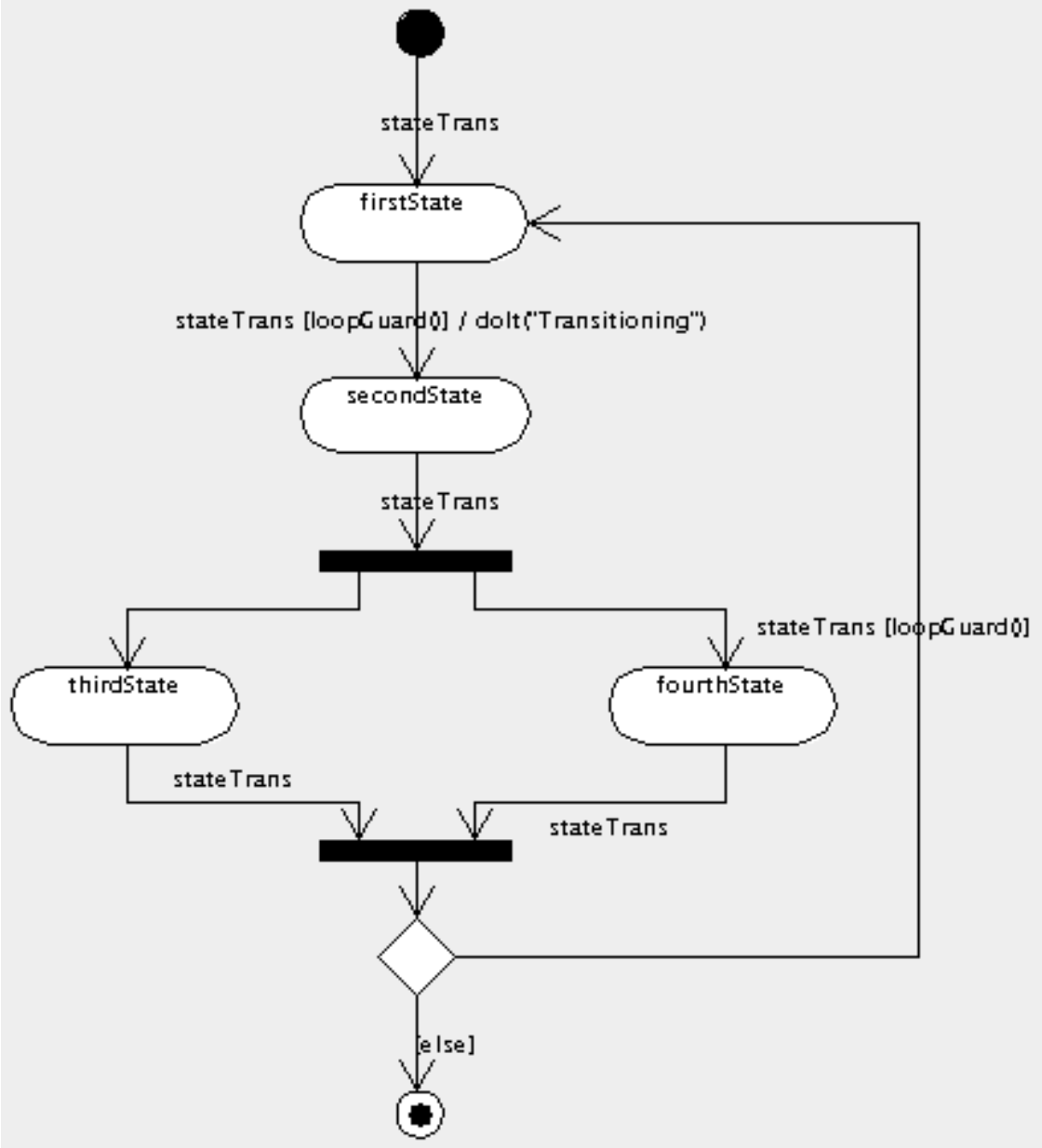
22.1. Introduction

This chapter describes each model element that can be created within an Activity diagram. Note that some sub-model elements of model elements may not actually themselves appear on the diagram.

There is a close relationship between this material and the Properties Tab of the Details Pane (see Section 13.3, “Properties Tab”). That section covers Properties in general, in this chapter they are linked to specific model elements.

Figure 22.1, “Possible model elements on an activity diagram.” shows an Activity Diagram with all possible model elements displayed.

Figure 22.1. Possible model elements on an activity diagram.



22.1.1. Limitations Concerning Activity Diagrams in ArgoUML

Activity diagrams are not fully developed yet in ArgoUML. Some aspects are not fully implemented, or may not behave as expected. In particular lacking are call states, swim lanes, control icons (signals), sub-activities, synch states. Interactions with other classifiers are provided by an object-flow-state which is only partly implemented.

22.2. Action State

An action state represents execution of an atomic action, usually the invocation of an action. Within the UML metamodel, `ActionState` is a sub-class of `SimpleState`. It is a specialized simple state that

only has an entry action, and with an implicit trigger as soon as that action is completed.

Caution

As a consequence any outgoing transitions from an action state should not have explicit triggers defined (ArgoUML will not currently check for this). They may have guards to provide a choice where there is more than one transition.

Note

Unlike an ordinary state, an internal transition, an exit action and a Do activity are not permitted for action states.

An action state is represented on an activity diagram in ArgoUML as a rectangle with rounded corners containing the name of the action state.

Caution

The UML standard specifies that the text shown in the action state on the activity diagram should contain the expression associated with the entry action - which is implemented as such since ArgoUML V0.18. In past versions of ArgoUML (0.16.1 and before), the diagram used to show the action state name. Loading a project created by one of the older versions, causes the project file to be converted to the correct format to conform to the UML standard. This process is designed to be transparent for the user, and the only drawback is, that the activity diagram in the project will not show correctly when reloaded in an old version of ArgoUML again.

22.2.1. Action State Details Tabs

The details tabs that are active for action states are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 22.2.2, “Action State Property ToolBar” and Section 22.2.3, “Property fields for action state” below.

`Documentation`
: Standard tab.

`Presentation`
: Standard tab. The values for the bounds of the action state define the bounding box of the action state.

`Stereotype`
: Standard tab that shows the stereotypes of the action state. In the UML metamodel, there are no stereotypes defined by default for a action state.

`Tagged Values`
: Standard tab. In the UML metamodel, `ActionState` has no standard tagged values defined.

22.2.2. Action State Property ToolBar

 `Go up`

Navigate up through the containment structure. Action states are contained by the (otherwise invisible) top state.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected action state, navigating immediately to the properties tab for that stereotype.

 `Delete from Model`

This deletes the action state from the model

Warning

This is a deletion from the model *not* just the diagram. It is not possible to delete an action state from the diagram, since that concept does not fit the UML standard.

Hence ArgoUML does also not show the `Add to Diagram` pop-up menu for action states.

22.2.3. Property fields for action state

`Name`
Text box. The name of the action state. By convention action state names start with a lower case letter and use bumpy caps to divide words within the name.

Note

ArgoUML does not enforce this naming convention.

`Container`
Text box. The container of the action state. This shows the otherwise invisible composite state at the top of the containment hierarchy.

`Entry-Action`
Text box. Shows the name of the action to be invoked on entry to this action state. According the UML standard, an Action State is obliged to have an Entry-Action.

Button 1 double-click navigates to the shown entry, button 2 gives a pop up menu with two entries.

- `New`. Add a new Entry action of a certain kind. This menu has the following 7 submenus to select the kind of action: Call Action, Create Action, Destroy Action, Return Action, Send Action, Terminate Action, Uninterpreted Action.
- `Delete From Model`. Delete the Entry-Action.

`Deferrable events`
Text box. The deferrable events of the action state.

`Incoming`
Text area. Lists the transitions that enter this action state.

Button 1 double-click navigates to the selected entry.

`Outgoing`
Text area. Lists the transitions that leave this action state.

Button 1 double-click navigates to the selected entry.

22.3. Action

This model element is described in the context of statechart diagrams (see Section 20.3, “Action”).

22.4. Transition

This model element is described in the context of statechart diagrams (see Section 20.8, “Transition”).

Caution

Remember that action states do not have explicit triggers. The transition is implicitly triggered as soon as the entry event of the action state is complete. An explicit trigger should not therefore be set.

The current release of ArgoUML will not check that this constraint is met.

Note

Transitions to and from an ObjectFlowState are dashed, to distinguish *object flow* from *control flow*.

22.5. Guard

This model element is described in the context of statechart diagrams (see Section 20.10, “Guard”).

22.6. Initial State

This model element is described in the context of statechart diagrams (see Section 20.12, “Initial State”).

22.7. Final State

This model element is described in the context of statechart diagrams (see Section 20.13, “Final State”).

22.8. Junction (Decision)

This model element is described in the context of statechart diagrams (see Section 20.14, “Junction”).

22.9. Fork

This model element is described in the context of statechart diagrams (see Section 20.16, “Fork”).

22.10. Join

This model element is described in the context of statechart diagrams (see Section 20.17, “Join”).

22.11. ObjectFlowState

(To Be Written)

Chapter 23. Deployment Diagram Model Element Reference

23.1. Introduction

This chapter describes each model element that can be created within a Deployment Diagram. Note that some sub-model elements of model elements on the diagram may not actually themselves appear on the diagram.

There is a close relationship between this material and the Properties Tab of the Details Pane (see Section 13.3, “Properties Tab”). That section covers Properties in general, in this chapter they are linked to specific model elements.

Within ArgoUML, the deployment diagram is used for both component diagrams (i.e. without instances, showing static dependencies of components) and deployment diagrams (showing how instances of components are handled by instances of nodes at run-time).

Caution

Deployment diagrams are not fully developed yet in ArgoUML. Some aspects are not fully implemented or may not behave as expected. Notable omissions are the possibility to draw new interfaces and proper stereotyping of the various dependency relationships.

Figure 23.1, “Possible model elements on a component diagram.” shows a component diagram with all possible model elements displayed.

Figure 23.1. Possible model elements on a component diagram.

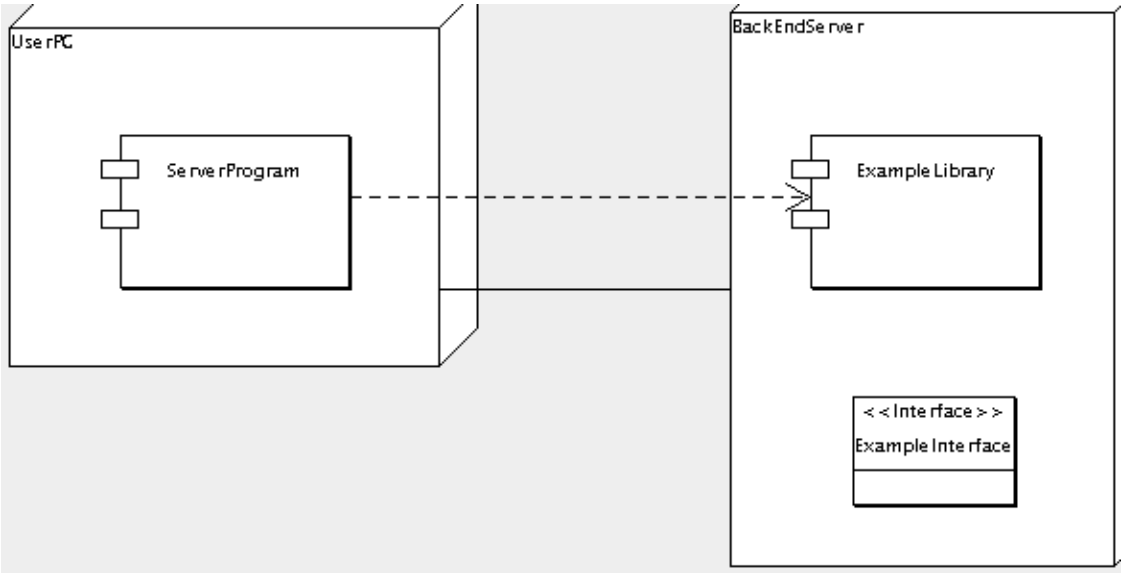
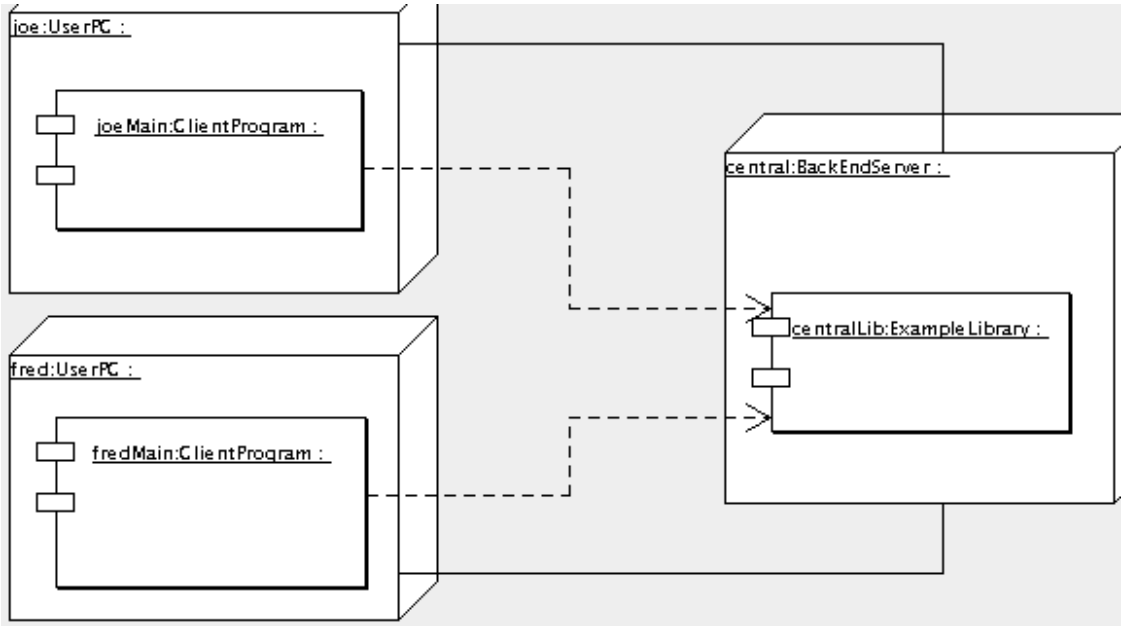


Figure 23.2, “Possible model elements on a deployment diagram.” shows a deployment diagram with all possible model elements displayed.

Figure 23.2. Possible model elements on a deployment diagram.



23.1.1. Limitations Concerning Deployment Diagrams in ArgoUML

The deployment diagram is generally well drawn, but there are only a subset of the relationships that should be shown available, which restricts the ability to show dynamic behavior of deployed code.

It is not possible to create new interfaces directly on this diagram; they can only be added if they are first created in the model (by drawing them on a class diagram).

It is an inconvenience that the alternative representation of an interface (as a small circle) is not supported.

23.2. Node

A node is a run-time physical object on which components may be deployed. In the UML metamodel it is a sub-class of `Classifier`.

A node is represented on a class diagram as a three dimensional box, labeled with its name.

23.2.1. Node Details Tabs

The details tabs that are active for nodes are as follows.

`ToDoItem`
: Standard tab.

`Properties`
: See Section 23.2.2, “Node Property Toolbar” and Section 23.2.3, “Property Fields For Node” be-

low.

`Documentation`
Standard tab.

`Presentation`
Standard tab. The `Bounds:` field defines the bounding box for the node on the diagram.

Warning

Beware that in the 0.18 release of ArgoUML, the bounding box just refers to the front face of the cube. This means that the three dimensional top and side may be ignored, for example when determining the limits of a diagram for saving graphics.

`Source`
Standard tab, but with no contents.

Caution

A node should not generate any code, so having this tab active is probably a mistake.

`Tagged Values`
Standard tab. In the UML metamodel, `Node` has the following standard tagged values defined.

- `persistence` (from the superclass, `Classifier`). Values `transitory`, indicating state is destroyed when an instance is destroyed or `persistent`, marking state is preserved when an instance is destroyed.
- `semantics` (from the superclass, `Classifier`). The value is a specification of the semantics of the node.
- `derived` (from the superclass, `ModelElement`). Values `true`, meaning the node is redundant—it can be formally derived from other elements, or `false` meaning it cannot.

Note

Derived nodes still have their value in analysis to introduce useful names or concepts, and in design to avoid re-computation.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

23.2.2. Node Property Toolbar


`Go up`

Navigate up through the package structure.


`New reception`

This creates a new reception, navigating immediately to the properties tab for that reception.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected node, navigating immediately to the properties tab for that stereotype.

 `Delete`

This deletes the node from the model

Warning

This is a deletion from the model *not* just the diagram. To delete a node from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

23.2.3. Property Fields For Node

`Name`
Text box. The name of the node. The name of a node has a leading capital letter, with words separated by “bumpy caps”.

Note

ArgoUML does not enforce this naming convention.

`Stereotype`
Drop down selector. Node is a type of classifier, and so it has the default stereotypes of a classifier as defined in the UML standard. ArgoUML provides the standard stereotypes for a classifier: `metaclass`, `powertype`, `process`, `thread` and `utility`.

`Navigate Stereotype`
 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 16.6, “Stereotype”).

`Namespace`
Drop down selector. Allows altering the namespace for the node. This is the package hierarchy.

`Modifiers`
Check box, with entries `abstract`, `leaf` and `root`.

- `abstract` is used to declare that this node cannot be instantiated, but must always be specialized. The name of an abstract node is displayed in italics on the diagram.
- `leaf` indicates that this node cannot be further specialized.
- `root` indicates the node can have no generalization.

`Generalizations`
Text area. Lists any node that *generalizes* this node.

Button 1 double click navigates to the generalization and opens its property tab.

`Specializations`
Text box. Lists any specialized node (i.e. for which this node is a generalization.

Button 1 double click navigates to the specialization and opens its property tab.

`Residents`
Text box. Lists any residents (see Section 23.4, “Component”) designed to be deployed on this type of node.

Button 1 double click navigates to the selected entry.

23.3. Node Instance

A node instance is an instance of a node where component instances (see Section 23.5, “Component Instance”) may reside. In the UML metamodel `NodeInstance` is a sub-class of `Instance` and is specifically an instance that is derived from a node.

A node instance is represented on a deployment diagram in ArgoUML as a three dimensional box labeled with the node instance name (if any) and node type, separated by a colon (`:`).

Tip

It is the presence of the colon (`:`) and the underlining of the name and type that distinguishes a node instance from a node.

23.3.1. Node Instance Details Tabs

The details tabs that are active for node instances are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 23.3.2, “Node Instance Property Toolbar” and Section 23.3.3, “Property Fields For Node Instance” below.

`Documentation`
Standard tab.

`Presentation`
Standard tab. The `Bounds:` field defines the bounding box for the node instance on the diagram.

Warning

Beware that in the current release of ArgoUML, the bounding box just refers to the front face of the cube. This means that the three dimensional top and side may be ignored, for example when determining the limits of a diagram for saving graphics.

`Source`
Standard tab, containing just the name of the node instance.

Caution

A node instance should not generate any code, so having this tab active is probably a mistake.

`Tagged Values`
Standard tab.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

`Checklist`
Standard tab for an Instance.

23.3.2. Node Instance Property Toolbar

 `Go up`

Navigate up through the package structure.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected node instance, navigating immediately to the properties tab for that stereotype.

 `Delete`

This deletes the node instance from the model

Warning

This is a deletion from the model *not* just the diagram. To delete an node instance from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

23.3.3. Property Fields For Node Instance

`Name`
Text box. The name of the node instance. By convention node instance names start with a lower case letter and use bumpy caps to divide words within the name.

Note

ArgoUML does not enforce this naming convention.

`Stereotype`
Drop down selector. Node instance has no stereotypes by default in the UML standard.

`Navigate Stereotype`
 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 16.6, “Stereotype”).

`Namespace`
Drop down selector. Records the namespace for the node instance. This is the package hierarchy.

`Stimuli sent`
(To Be Written).

`Stimuli Received`
(To Be Written).

`Residents`
Text box. Lists any residents (see Section 23.4, “Component”) designed to be deployed on this type of node.

Button 1 double click navigates to the selected entry.

`Classifiers`
Text field. A Node instance type can be selected here.

Caution

ArgoUML V0.18 lists many more items in the dropdown list then solely Nodes. Beware to select Nodes only.

23.4. Component

A component type represents a distributable piece of implementation of a system, including software code (source, binary, or executable) but also including business documents, etc., in a human system. Components may be used to show dependencies, such as compiler and run-time dependencies or information dependencies in a human organization. In the UML metamodel it is a sub-class of `Classifier`.

A component is represented on a class diagram as a box with two small rectangles protruding from its left side, labeled with its name.

23.4.1. Component Details Tabs

The details tabs that are active for components are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 23.4.2, “Component Property Toolbar” and Section 23.4.3, “Property Fields For Component” below.

`Documentation`
Standard tab.

`Presentation`
Standard tab. The `Bounds:` field defines the bounding box for the component on the diagram.

`Source`
Standard tab, but with no contents.

Caution

A component should not generate any code, so having this tab active is probably a mistake.

`Tagged Values`
Standard tab.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

23.4.2. Component Property Toolbar

 `Go up`

Navigate up through the package structure.

 `New reception`

This creates a new reception, navigating immediately to the properties tab for that reception.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected component, navigating immediately to the properties tab for that stereotype.

 `Delete`

This deletes the component from the model

Warning

This is a deletion from the model *not* just the diagram. To delete a component from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

23.4.3. Property Fields For Component

`Name`
Text box. The name of the component. The name of a component has a leading capital letter, with words separated by “bumpy caps”.

Note

ArgoUML does not enforce this naming convention.

`Stereotype`
Drop down selector. Component is provided by default with the UML standard stereotypes `document`, `executable`, `file`, `library` and `table`. ArgoUML also provides the standard Classifier stereotypes, `metaclass`, `powertype`, `process`, `thread` and `utility`.

`Navigate Stereotype`
 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 16.6, “Stereotype”).

`Namespace`
Drop down selector. Records and allows altering the namespace for the component. This is the package hierarchy.

`Modifiers`
Check box, with entries `abstract`, `leaf` and `root`.

- `Abstract` is used to declare that this component cannot be instantiated, but must always be specialized.
- `Leaf` indicates that this component cannot be further specialized.
- `Root` indicates the node can have no generalization.

`Generalizations`
Text box. Lists any component that generalizes this component.

`Specializations`
Text area. Lists any derived components, i.e those for which this component is a generalization.

`Client Dependencies`
Text area. Lists outgoing dependencies. Button 1 double click navigates to the dependency.

`Supplier Dependencies`
Text area. Lists incoming dependencies. Button 1 double click navigates to the dependency.

`Residents`
Text box. Lists any residents (see Section 23.4, “Component”) designed to be deployed on this type of node.

Button 1 double click navigates to the selected entry.

23.5. Component Instance

A component instance is an instance of a component (see Section 23.4, “Component”) which may reside on a node instance (see Section 23.3, “Node Instance”). In the UML metamodel `ComponentInstance` is a sub-class of `Instance` and is specifically an instance that is derived from a component.

A component is represented on a class diagram as a box with two small rectangles protruding from its left side, labeled with its name.

A component instance is represented on a sequence diagram in ArgoUML as a box with two small rectangles protruding from its left side labeled with the component instance name (if any) and component

type, separated by a colon (`:`).

Tip

It is the presence of the colon (`:`) and the underlining of the name and type that distinguishes a component instance from a component.

23.5.1. Component Instance Details Tabs

The details tabs that are active for component instances are as follows.

`ToDoItem`
Standard tab.

`Properties`
See Section 23.5.2, “Component Instance Property Toolbar” and Section 23.5.3, “Property Fields For Component Instance” below.

`Documentation`
Standard tab.

`Presentation`
Standard tab. The `Bounds:` field defines the bounding box for the component on the diagram.

`Source`
Standard tab, containing just the name of the component instance.

Caution

A component instance should not generate any code, so having this tab active is probably a mistake.

`Tagged Values`
Standard tab.

Note

The UML `Element` metaclass from which all other model elements are derived includes the tagged element `documentation` which is handled by the *documentation tab* under ArgoUML

`Checklist`
Standard tab for an Instance.

23.5.2. Component Instance Property Toolbar

 `Go up`

Navigate up through the package structure.

 `New Stereotype`

This creates a new Stereotype (see Section 16.6, “Stereotype”) for the selected component instance, navigating immediately to the properties tab for that stereotype.


`Delete`

This deletes the component instance from the model

Warning

This is a deletion from the model *not* just the diagram. To delete a component instance from the diagram, but keep it within the model, use the main menu `Remove From Diagram` (or press the Delete key).

23.5.3. Property Fields For Component Instance

`Name`
Text box. The name of the component instance. By convention component instance names start with a lower case letter and use bumpy caps to divide words within the name.

Note

ArgoUML does not enforce this naming convention.

`Stereotype`
Drop down selector. Component instance has no stereotypes by default in the UML standard.

`Navigate Stereotype`
 icon. If a stereotype has been selected, this will navigate to the stereotype property panel (see Section 16.6, “Stereotype”).

`Namespace`
Drop down selector. Records and allows to change the namespace for the component instance. This is the package hierarchy.

`Stimuli sent`
(To Be Written).

`Stimuli Received`
(To Be Written).

`Residents`
Text box. Lists any residents (see Section 23.4, “Component”) designed to be deployed on this component.

Button 1 double click navigates to the selected entry.

`Classifiers`
Drop down selector. A Component instance type can be selected here.

Caution

ArgoUML V0.18 lists many more items in the dropdown list then solely Components.

Beware to select Components only.

23.6. Dependency

A key part of any component or deployment diagram is to show dependencies. For details see Section 18.14, “Dependency”.

Caution

UML relies on stereotyping of dependencies on component and deployment diagrams to characterize the types of relationship. In the current release of ArgoUML there are limitations in the implementation of dependencies which limit this functionality.

23.7. Class

A component diagram may show the key internal structure of components, including classes within the component. For details see Section 18.6, “Class”.

Caution

Classes can only be added to a component diagram if they already exist in the model (by selecting them in the explorer and executing the "Add to diagram" button 2 command). There is no way to create a new class on a component diagram.

23.8. Interface

A component or deployment diagram may show components or component instances which implement interfaces. For details see Section 18.16, “Interface”.

Caution

The V0.18 release of ArgoUML uses the same representation of an interface as a class diagram. The UML standard suggests that an interface on a component or deployment diagram should just be shown as a small open circle, connected to the component which realizes that interface.

Warning

There is no way to show the linking of an interface to a component or component instance in the V0.18 release of ArgoUML.

23.9. Association

Components may be associated to each other. For details about associations, see Section 18.12, “Association”.

Where classes or interfaces are shown within components on component diagrams, they may be shown

linked by associations.

23.10. Object

Just as components may show the classifiers that make up their internal structure, component instances on deployment diagrams may show the classifier instances that make up their internal structure. In practice the only instance that is of use is an object (an instance of a class). For details see Section 19.2, “Object”.

23.11. Link

Where objects (Node Instances or Class Instances) are shown within component instances on deployment diagrams, their inter-relationships may be shown as links (instances of an association). See Section 19.9, “Link” for details.

Chapter 24. Built In DataTypes, Classes, Interfaces and Stereotypes

24.1. Introduction

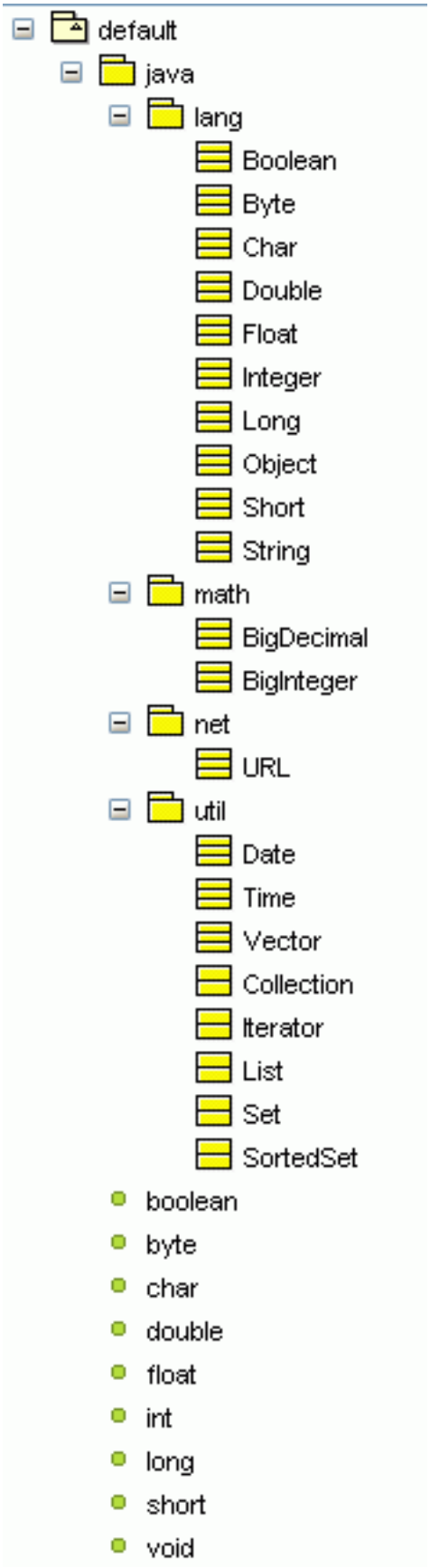
This chapter describes the datatypes, classes, interfaces and stereotypes, which by default, are built in to ArgoUML.

Datatypes, classes and interfaces are generally available for use anywhere a class may be selected in the properties tab. The most common use is for return type and parameter types in method signatures.

24.1.1. Package Structure

ArgoUML datatypes, classes and interfaces are effectively organized as a hierarchy beneath the overall model itself. They are grouped in four packages, `lang`, `math`, `net` and `util`, themselves subpackages of `java`, which is a subpackage of the model itself. Figure 24.1, “Hierarchy of datatypes, classes and interfaces within ArgoUML” shows this structure.

Figure 24.1. Hierarchy of datatypes, classes and interfaces within ArgoUML


default
java
lang
Boolean
Byte
Char
Double
Float
Integer
Long
Object
Short
String
math
BigDecimal
BigInteger
net
URL
util
Date
Time
Vector
Collection
Iterator
List
Set
SortedSet
boolean
byte
char
double
float
int
long
short
void

24.1.2. Exposure in the model

You will not find build-in DataTypes, Classes, and Interfaces exposed within the model by default (i.e. they are not present in the explorer). However, once you select one of the built-in DataTypes, Classes, or Interfaces (in the "Type" combo-box on the property sheet of a parameter of an operation of a class), then it becomes visible: you will find that the DataType, Class, or Interface has appeared in the model, in its correct package stucture for the latter 2.

24.2. Built In Datatypes

These are the built in atomic types. You can change them if you wish. However this is not good practice.

All these can be found in the `java.lang` subpackage of the main model.

Caution

You should be aware that these are Java datatypes. They are not mandated by the UML standard.

These are the standard datatypes. For their definition refer to the Java standard.

- `boolean`
- `byte`
- `char`
- `double`
- `float`
- `int`
- `long`
- `short`
- `void`

Note

`void` is not strictly speaking a type, but the absence of type. ArgoUML knows about void and allows it as an option where a datatype may be selected.

24.3. Built In Classes

These are the common classes, corresponding to classes defined within the standard Java environment. It is up to you if you wish to change them.

These are found in all four subpackages of the `java` subpackage.

For a definition of these classes see the Java language and library definitions.

24.3.1. Built In Classes From `java.lang`

These are the classes within the `java.lang` package.

- `Boolean`
- `Byte`
- `Char`
- `Double`
- `Float`
- `Integer`
- `Long`
- `Object`
- `Short`
- `String`

24.3.2. Built In Classes From `java.math`

These are the classes within the `java.math` package.

- `Big Decimal`
- `Big Integer`

24.3.3. Built In Classes From `java.net`

These are the classes within the `java.net` package.

- `URL`

24.3.4. Built In Classes From `java.util`

These are the classes within the `java.util` package.

- `Vector`
- `Date`
- `Time`

24.4. Built In Interfaces

These are some useful interfaces, corresponding to classes defined within the standard Java environment. Interfaces have many of the properties of classes (like all types) and you can change them if you wish.

All these can be found in the `java.util` subpackage of the main model.

These are the interfaces defined within the `java.util` package. For their definition consult the Java language and library references.

- `Collection`
- `Iterator`
- `List`
- `Set`
- `Sorted Set`

24.5. Built In Stereotypes

UML 1.4 defines a large number of stereotypes of which most are supported by ArgoUML.

Caution

Not all stereotypes defined by UML 1.4 appear in ArgoUML V0.20 due to the fact that they were not yet updated from previous versions of ArgoUML that only supported UML 1.3. Also, there are limitations in the current implementation of some base elements. The table below lists all stereotypes defined in UML 1.4 and if they are supported in ArgoUML or not.

Caution

The UML 1.4 standard also specifies many stereotypes in the chapters “Example Profiles”: one for “Software Development” and one for “Business Modeling”. Due to the specialized nature of these profiles, implementation in ArgoUML is postponed until a yet undetermined moment.

Table 24.1. Stereotypes defined in UML 1.4 and ArgoUML

StereoType	***Base Element***	***ArgoUML support***
access	Permission	yes
appliedProfile	Package	no
association	AssociationEnd	yes
auxiliary	Class	no

StereoType	***Base Element***	***ArgoUML support***
become	Flow	no
call	Usage	yes
copy	Flow	no
create	BehavioralFeature	yes
create	CallEvent	yes
create	Usage	yes
derive	Abstraction	yes
destroy	BehavioralFeature	yes
destroy	CallEvent	yes
document	Abstraction	no
executable	Abstraction	no
facade	Package	yes
file	Abstraction	no
focus	Class	no
framework	Package	yes
friend	Permission	yes
global	AssociationEnd	yes
implementation	Class	no
implementation	Generalization	yes
implicit	Association	yes
import	Permission	yes
instantiate	Usage	yes
invariant	Constraint	no
library	Abstraction	no

StereoType	***Base Element***	***ArgoUML support***
local	AssociationEnd	yes
metaclass	Class	no
metamodel	Package	yes
modelLibrary	Package	no
parameter	AssociationEnd	yes
postcondition	Constraint	no
powertype	Class	no
precondition	Constraint	no
process	Classifier	yes
profile	Package	no
realize	Abstraction	yes
refine	Abstraction	yes
requirement	Comment	yes
responsibility	Comment	yes
self	AssociationEnd	yes
send	Usage	yes
signalflow	ObjectFlowState	yes
source	Abstraction	no
stateInvariant	Constraint	no
stub	Package	yes
systemModel	Package	yes
table	Abstraction	no
thread	Classifier	yes
topLevel	Package	yes

StereoType	***Base Element***	***ArgoUML support***
trace	Abstraction	yes
type	Class	yes

Glossary

A

Activity Diagram

A UML diagram capturing the dynamic behavior of a system or sub-system. See Section 6.10, “Activity Diagrams (To be written)” for more information.

Action

Behavior associated with *States* or *Transitions* in *State Diagram*. These actions are invocations of *Methods* and appear on *Sequence* and *Collaboration Diagrams*.

Actor

A representation of an agent (animate or inanimate) on a *Use Case Diagram* external to the system being designed.

Analysis

Analysis is the process of taking the “customer” requirements and re-casting them in the language of, and from the perspective of, a putative solution.

Association Class

A class that characterizes the association between two other classes.

Association

A relationship between two classes in a Class Diagram or between Use Cases or Use Cases and Actors in a Use Case Diagram.

Attribute (of a Class or Object)

An attribute of a class or object is a specification of a data element encapsulated by that object.

C

CASE

Computer Aided Software Engineering.

Class

The encapsulation of the data associated with a model element (its *attributes*) and the actions associated with the model element (its *methods*).

A class specifies the characteristics of a model element. An *object* represents an instance of the model element.

Classes and objects in UML are represented on *Activity Diagrams*, *Class Diagrams*, *Collaboration Diagrams* and *Sequence Diagrams*.

Class Diagram

A UML Diagram showing the structural relationship between classes. See Section 5.2, “Class Diagrams (To be written)” for more information.

Collaboration

The process whereby several objects cooperate to provide some higher level behavior that is greater than the sum of the behaviors of the objects.

Collaboration Diagram

A UML Diagram showing the dynamic behavior as messages are passed between objects. Equivalent to a *Sequence Diagram*. Which representation is appropriate depends on the problem under consideration.

Collaborator

An object that participates in a Collaboration.

Comprehension and Problem Solving

A design visualization theory within cognitive psychology. The theory notes that designers must bridge a gap between their mental model of the problem or situation and the formal model of a solution or system.

This theory suggests that programmers will benefit from:

1. Multiple representations such as program syntactic decomposition, state transitions, control flow, and data flow. These allow the programmer to better identify elements and relationships in the problem and solution and thus more readily create a mapping between their situation models and working system models.

2. Familiar aspects of a situation model, which improve designers' abilities to formulate solutions.

Concept Class Diagram

A Class Diagram constructed during the Analysis Phase to show the main structural components of the problem identified in the Requirements Phase. See Chapter 5, *Analysis* for more information.

Critic

A process within ArgoUML that provides suggestions as to how the design might be improved. Suggestions are based on principles within three theories of cognitive psychology, *reflection-in action*, *opportunistic design* and *comprehension and problem solving*.

E

Extend Relationship

A relationship between two Use Cases, where the *extended* Use Case describes a special variant of the *extending* Use Case.

G

Generalization Relationship

A relationship between one *generalizing* Use Cases and one or more

generalized Use Cases, where the *generalized* Use Cases are particular examples of the *generalizing* Use Case.

GUI

Graphical User Interface.

H

Hierarchical Statechart Diagram

A *Statechart Diagram* that contains subsidiary statechart diagrams within individual *States*.

I

Include Relationship

A relationship between two Use Cases, where the *included* Use Case describes part of the functionality of the *including* Use Case.

Iterative Design Process

A design process where each all phases (requirements, analysis, design, build, test) are tackled partially in a series of iterations. See Section 3.2.1, “Types of Process” for more information.

J

Java

A fully object oriented programming language introduced by Sun Microsystems. More strongly typed than C++, it compiles to an interpreted code, the Java Virtual Machine (JVM). The JVM means that Java code should run on any machine that has implemented the JVM.

The most significant component of Java was integration of the JVM into web browsers, allowing code (Applets) to be download and run over the web.

ArgoUML is written in Java.

M

Mealy Machine

A *Statechart Diagram* where actions are associated with *States*.

Method (of a Class or Object)

A method of a class or object is a specification of behavior encapsulated by that object.

Moore Machine

A *Statechart Diagram* where actions are associated with *Transitions*.

O

Object

An instance of a *Class*.

Classes and objects in UML are represented on *Activity Diagrams*, *Class Diagrams*, *Collaboration Diagrams* and *Sequence Diagrams*.

OCL

Object Constraint Language. A language for describing constraints within UML.

OMG

The Object Management Group. An international industry standardization body. Best known for CORBA and UML.

OOA&D

Object Oriented Analysis and Design. An approach to software problem analysis and design based on objects, which encapsulate both data and code. See See Section 1.1.1, “Object Oriented Analysis and Design” or any standard textbook on Software Engineering.

UML is a notation to support OOA&D.

Opportunistic Design

A theory within cognitive psychology suggesting that although designers plan and describe their work in an ordered, hierarchical fashion, in actuality, they choose successive tasks based on the criteria of cognitive cost. Simply stated, designers do not follow even their own plans in order, but choose steps that are mentally least expensive among alternatives.

P

Pane

A sub-window within the main window of the ArgoUML user interface.

R

Realization Use Case

A *Use Case* where the *Use Case Diagram* and *Use Case Specification* are in the language of the solution domain, rather than the problem domain.

Reflection-in-Action

A theory within cognitive psychology which observes that designers of complex systems do not conceive a design fully-formed. Instead, they must construct a partial design, evaluate, reflect on, and revise it, until they are ready to extend it further. As developers work hands-on with the design, their mental model of the problem situation improves, hence improving their design.

Requirement Capturing

Requirement capturing is the process of identifying what the "customer" wants from the proposed system. See Chapter 4, *Requirements Capture* for a fuller description.

Responsibility

S ome behavior for which an object is held accountable. A responsibility denotes the obligation of an object to provide a certain behavior.

S

Scenario

A specific sequence of actions that illustrates behavior.

Sequence Diagram

A UML Diagram showing the dynamic behavior as messages are passed between objects. Equivalent to a *Collaboration Diagram*. Which representation is appropriate depends on the problem under consideration. See Section 5.4, "Sequence Diagrams (To be written)" for more information.

SGML

Standard Graphical Markup Language. Defined by ISO 8879:1986.

Simula 67

A procedural programming language intended for simulation. Noted for its introduction of *objects* and *coroutines*.

State

Within a *Statechart Diagram* a one of the possible configurations of the machine.

Statechart Diagram

A UML Diagram showing the dynamic behavior of an active *Object*. See Section 5.6, "Statechart Diagrams (To be written)" for more information.

Stereotypes and Stereotyping

Any model element within UML can be given a *stereotype* to indicate its association with a particular role in the design. A stereotype `spqr` is generally indicated with the notation `<<spqr>>`.

A stereotype defines a Namespace within the design. Examples of stereotypes are `<<business>>` and `<<realization>>` for Use Cases, used to distinguish between Use Cases at the requirements phase defined in terms of the problem domain, and Use Cases at the analysis phase defined in terms of the solution domain.

Supplementary Requirement Specification

The document capturing non-functional requirements that cannot be associated with Use Cases.

SVG

Scalable Vector Graphics format. A standard representation of graphics diagrams that use vectors. ArgoUML can export diagrams in SVG.

System Sequence Diagram

A *Sequence Diagram* used in the *Analysis* Phase showing the dynamic behavior of the overall system. See Chapter 5, *Analysis* for more information.

System Statechart Diagram

A *Statechart Diagram* used in the *Analysis* Phase showing the dynamic behavior of an active top level system objects. See Chapter 5, *Analysis* for more information.

T

To-Do List

A feature of ArgoUML allowing the user to record activities that are yet to be completed.

Transition

The change between *States* in a *Statechart Diagram*..

U

UML

Universal Modeling Language. A graphical notation for OOA&D processes, standardized by the OMG. ArgoUML supports UML 1.4. UML 2.0 is in the final stages of standardization and should be complete during 2006.

Use Case

A UML notation for capturing requirements of a system or subsystem. See Section 4.3, “Output of the Requirements Capture Process” for more information.

Use Case Diagram

A UML diagram showing the relationships between Actors and Use Cases. See Section 4.3, “Output of the Requirements Capture Process” for more information.

Use Case Specification

The document capturing the detailed requirements behind a Use Case.

V

Vision Document

The top level document describing what the system being developed is to achieve.

W

W3C

The World Wide Web Consortium, www.w3c.org

[http://www.w3c.org]. An international standardization body for all things to do with the World Wide Web.

Waterfall Design Process

A design process where each phase (requirements, analysis, design, build, test) is completed before the next starts. See Section 3.2.1, “Types of Process” for more information.

X

XMI

XML Model Interchange format. A format for file storage of UML models. Currently incomplete, since it does not carry all graphical layout information, so must be supplemented by files carrying that information.

XML

eXtensible Markup Language. A simplified derivative of SGML defined by W3C

Appendix A. Supplementary Material for the Case Study

A.1. Introduction

The case study requires various material (mostly documents) that live alongside the design diagram

A.2. Requirements Documents (To be written)

To be written...

A.2.1. Vision Document (To be written)

To be written...

A.2.2. Use Case Specifications (To be written)

To be written...

A.2.2.1. UC Specification 1 (To be written)

To be written...

A.2.3. Supplementary Requirements Specification (To be written)

To be written...

Appendix B. UML resources

B.1. The UML specs (To be written)

To be written...

B.2. UML related papers (To be written)

To be written...

B.2.1. UML action specifications (To be written)

To be written...

B.3. UML related websites (To be written)

To be written...

Appendix C. UML Conforming CASE Tools

C.1. Other Open Source Projects (To be written)

To be written...

C.2. Commercial Tools (To be written)

To be written...

Appendix D. The C++ Module

The ArgoUML C++ Module (C++ Mod.) provides C++ code generation functionalities and C++ notation within ArgoUML. It works the same way as the other languages' modules.

D.1. Modeling for C++

The C++ programming language has constructs that aren't contained by default in UML. Examples are pointers, global functions and variables, references and operator overloading. To enable us to apply these constructs in our models and be capable of taking advantage of it for code generation and C++ notation in UML diagrams, the C++ module uses conventions in the use of the extension features of UML, tagged values and stereotypes.

Since UML and C++ are object oriented, there is an obvious correspondence between the UML model elements and C++ structural constructs, e.g, the UML `Class` is related to the C++ `class`. These obvious relations will not be described here, since it is assumed that an ArgoUML user that wants to model for C++ has basic knowledge of both C++ and UML.

Tagged values are one of the main means by which we can define code generation behavior. They have a name - the tag - and a value, and are applied to model elements.

The tagged values in use for the C++ module have two categories:

- free format values - any `String` is valid, except the empty `String`
- formated values - the value must obey some restrictions, e.g., be one of `true` or `false` (abbreviated to `true || false`)

For `Boolean` tagged values, only the values "`true`" or " `false`" are applicable. If a `Boolean` tagged value does not exist or is invalid for one model element, a default value is assumed by the code generator. In the bellow documentation the default value is marked.

Free format tagged values are only significant if present and if the value isn't an empty `String`. When the value must follow some sort of format, that is explicitly stated. In this case, there is the chance that the value is invalid. If the value is invalid, no assumptions are made; the generator will trace the problem and ignore the tagged value.

D.1.1. `Class` tagged values

`constructor`
: `true` - generates a default constructor for the `class`.

  `false` (default) - no default constructor is generated, unless it is explicitly modeled with the «`create`» stereotype.

`header_incl`
: Name of the file to include in the header.

Note

If we desire to have multiple headers included this way, just use multiple tagged values with `header_incl` as the tag.

Other tagged values used for C++ modeling may also be used this way. This note won't be repeated in those cases.

`source_incl`
Name of the file to include in the source (`.cpp` file).

`typedef_public`
`<source type> <type_name>` - creates `typedef` line in the public area of the `class` with `typedef <source type> <type name>`.

`typedef_protected`
Same as `typedef_public`, but, in `protected` area.

`typedef_private`
Same as `typedef_public`, but, in the `private` area.

`typedef_global_header`
Same as `typedef_public`, but, in the global area of the header.

`typedef_global_source`
Same as `typedef_global_source`, but, in the source file.

`TemplatePath`
`Directory` - will search in the specified directory for the template files "header_template" and "cpp_template" which are placed in top of the corresponding file. The following tags in the template file are replaced by model values: |FILENAME|, |DATE|, |YEAR|, |AUTHOR|, |EMAIL|. If no such tag is specified, the templates are searched in the subdirectory of the root directory for the code generation.

`email`
`name@domain.country` - replaces the tag |EMAIL| of the template file.

`author`
`name` - replaces the tag |AUTHOR| of the template file.

Note

You may simply use the Author property in the documentation property panel.

D.1.2. `Attribute` tagged values

UML `Attributes` are mapped to `class member variables`.

`pointer`
`true` - the type of the member variable will be a pointer to the attribute type.

For example, if you have the UML `Attribute`: `name: std::string`, with the `pointer` tagged value set to true, the generated member variable would be: `std::string* name;`

`false` (default) - no pointer modifier is applied.

`reference`
`true` - the type of the member variable will be a reference to the attribute type.

`false` (default) - no reference modifier is applied.

`usage`
`header` - will lead for class types to a pre-declaration in the header, and the include of the remote class header in the header of the generated class.

`MultiplicityType`
`list || slist || vector || map || stack || stringmap` - will define a multiplicity as the corresponding STL container, if the `Multiplicity` range of the attribute is variable (for fixed size ranges this setting is ignored).

`set`
`private || protected || public` - creates a simple function to set the attribute by a function (call by reference is used for class-types, else call by value); place the function in the given visibility area.

`get`
`private || protected || public` - as for `set`.

D.1.3. `Parameters`

D.1.3.1. Variable passing semantics

If a `Parameter` for an `Operation` is marked as `out` or `inout` the variable will be passed by reference (default) or pointer (needs tagged value `pointer` - see above), otherwise by value.

Return values in UML are simply `Parameters` marked as `return`, therefore everything here applies to them, except where explicitly noted.

Warning

Note that UML allows multiple return values. This is possible to support in C++ as out parameters, but, currently the generator doesn't supports it.

This problem is being handled in issue #3553 - handle multiple return parameters [http://argouml.tigris.org/issues/show_bug.cgi?id=3553].

D.1.3.2. `Parameter` tagged values

`pointer`
`true || false` (default) - same as for `Attributes`.

`reference`
ditto

D.1.4. Preserved sections

With each code generation, special comments around the member function definitions will be generated like this:

```
function Testclass::Testclass()
// section -64--88-0-40-76f2e8:ec37965ae0:-7fff begin
```

```
{
}
 // section -64--88-0-40-76f2e8:ec37965ae0:-7fff end
```

All code you put within the "begin" and "end" lines will be preserved when you generate the code again. Please do not change anything within these lines because the sections are recognized by this comment syntax. As the curly braces are placed within the preserved area, attribute initializers are preserved on constructors.

This also works if you change Method Names after the generation.

```
  void newOperation(std::string test = "fddsaffa")
  // section 603522:ec4c7ff768:-7ffc begin
{
}
  // section 603522:ec4c7ff768:-7ffc end
```

If you delete an Operation in the model. The next time the class is generated, the lost code - i.e., the whole member function definition - will be added as comment to the end of the file.

Appendix E. Limits and Shortcomings

As all products, ArgoUML has some limits. Those important to the user are listed in this section.

E.1. Diagram Canvas Size

Due to the underlying diagram editing software, the canvas size for diagrams is limited to 6000 units in height and width.

E.2. Missing functions

Appendix F. Open Publication License

F.1. Requirements On Both Unmodified And Modified Versions

The Open Publication works may be reproduced and distributed in whole or in part, in any medium physical or electronic, provided that the terms of this license are adhered to, and that this license or an incorporation of it by reference (with any options elected by the author(s) and/or publisher) is displayed in the reproduction.

Proper form for an incorporation by reference is as follows:

> Copyright (c) <year> by <author's name or designee>. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, vX.Y or later (the latest version is presently available at http://www.opencontent.org/openpub/ [http://www.opencontent.org/openpub/]).

The reference must be immediately followed with any options elected by the author(s) and/or publisher of the document (see section VI).

Commercial redistribution of Open Publication-licensed material is permitted.

Any publication in standard (paper) book form shall require the citation of the original publisher and author. The publisher and author's names shall appear on all outer surfaces of the book. On all outer surfaces of the book the original publisher's name shall be as large as the title of the work and cited as possessive with respect to the title.

F.2. Copyright

The copyright to each Open Publication is owned by its author(s) or designee.

F.3. Scope Of License

The following license terms apply to all Open Publication works, unless otherwise explicitly stated in the document.

Mere aggregation of Open Publication works or a portion of an Open Publication work with other works or programs on the same media shall not cause this license to apply to those other works. The aggregate work shall contain a notice specifying the inclusion of the Open Publication material and appropriate copyright notice.

SEVERABILITY. If any part of this license is found to be unenforceable in any jurisdiction, the remaining portions of the license remain in force.

NO WARRANTY. Open Publication works are licensed and provided “as is” without warranty of any kind, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose or a warranty of non-infringement.

F.4. Requirements On Modified Works

All modified versions of documents covered by this license, including translations, anthologies, compil-

ations and partial documents, must meet the following requirements:

1. The modified version must be labeled as such.
2. The person making the modifications must be identified and the modifications dated.
3. Acknowledgement of the original author and publisher if applicable must be retained according to normal academic citation practices.
4. The location of the original unmodified document must be identified.
5. The original author's (or authors') name(s) may not be used to assert or imply endorsement of the resulting document without the original author's (or authors') permission.

F.5. Good-Practice Recommendations

In addition to the requirements of this license, it is requested from and strongly recommended of redistributors that:

1. If you are distributing Open Publication works on hardcopy or CD-ROM, you provide email notification to the authors of your intent to redistribute at least thirty days before your manuscript or media freeze, to give the authors time to provide updated documents. This notification should describe modifications, if any, made to the document.
2. All substantive modifications (including deletions) be either clearly marked up in the document or else described in an attachment to the document.
3. Finally, while it is not mandatory under this license, it is considered good form to offer a free copy of any hardcopy and CD-ROM expression of an Open Publication-licensed work to its author(s).

F.6. License Options

The author(s) and/or publisher of an Open Publication-licensed document may elect certain options by appending language to the reference to or copy of the license. These options are considered part of the license instance and must be included with the license (or its incorporation by reference) in derived works.

A. To prohibit distribution of substantively modified versions without the explicit permission of the author(s). “Substantive modification” is defined as a change to the semantic content of the document, and excludes mere changes in format or typographical corrections.

To accomplish this, add the phrase “Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.” to the license reference or copy.

B. To prohibit any publication of this work or derivative works in whole or in part in standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from the copyright holder.

To accomplish this, add the phrase “Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.” to the license reference or copy.

F.7. Open Publication Policy Appendix:

(This is not considered part of the license.)

Open Publication works are available in source format via the Open Publication home page at http://works.opencontent.org/ [http://works.opencontent.org/].

Open Publication authors who want to include their own license on Open Publication works may do so, as long as their terms are not more restrictive than the Open Publication license.

If you have questions about the Open Publication License, please contact David Wiley [mailto:dw2@opencontent.org], and/or the Open Publication Authors' List at opal@opencontent.org [mailto:opal@opencontent.org], via email.

To *subscribe* to the Open Publication Authors' List: Send E-mail to opal-request@opencontent.org with the word “subscribe” in the body.

To *post* to the Open Publication Authors' List: Send E-mail to opal@opencontent.org or simply reply to a previous post.

To *unsubscribe* from the Open Publication Authors' List: Send E-mail to opal-request@opencontent.org with the word “unsubscribe” in the body.

Appendix G. The CRC Card Methodology

A CRC card is ostensibly an index card that is used to represent classes, their responsibilities, and the interactions between them. The term CRC card is also used to refer to a methodology for object oriented modeling based on their use.

Kent Beck and Ward Cunningham introduced CRC cards in a paper "A Laboratory for Teaching Object-Oriented Thinking" that was presented at the OOPSLA (Object-Oriented Programming, Systems, Languages & Applications) conference in 1989. A tutorial on the subject can be found at http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/crc_b/. The CRC card methodology was originally designed as a teaching tool but has proved useful as a modeling tool as well.

The three parts of the CRC acronym were felt by the authors of the paper to represent the essential dimensions of object oriented modeling. The term Responsibilities refers to the contract that the class under discussion offers to the rest of the world (Interface and Contract are similar concepts). Responsibilities model the things that a class can do. Services, Methods, or Operations will result from these. The term Collaborators refers to the classes whose services the class under discussion will use. Kent Beck tried unsuccessfully to use ther term Helpers instead of Collaborators to indicate classes that were supporting the class under discussion. It is widely believed that the terminology was chosen because CRC are the initials of Ward Cunningham's son.

Why use CRC cards?

- They are portable. No computers are required so they can be used anywhere. Even away from the office.
- They allow the participants to experience first hand how the system will work. No computer tool can replace the interaction that happens by physically picking up the cards and playing the role of that object.
- They are a useful tool for teaching people the object-oriented paradigm.
- They can be used as a methodology themselves or as a front end to a more formal methodology such as Booch, Wirfs-Brock, Jacobson, etc. Although CRC cards were created for teaching, they have proven useful for much more.
- They have become an accepted method for analysis and design. The biggest contributing factor to their success is the fact that they provide an informal and non threatening environment that is productive to working and learning.

G.1. The Card

The exact format of the card can be customized to the preferences of the group, but the minimal required information is the name of the class, it's subclasses and superclasses, it's responsibilities and the collaborators for each of those responsibilities. The back of the card can be used for a description of the class. During the design phase attributes of the class can be recorded on the back as well. One way to think of the card is that the front contains the public information, and the back contains the encapsulated, implementation details. As a class is defined a card is made for that class with its name entered. When a class is assigned to an individual that has only a class name on it, the individual (or the group) selects an initial set of responsibilities for the class. This initial set should be whatever (if anything) is immediately obvious.

G.2. The Group

Whether they are implicitly or explicitly defined the requirements for the system need to be familiar to the people participating in the group.

The ideal group size for a CRC card session is five or six people. This size generally allows everyone to productively participate. In groups of larger size productivity is cut by more disagreements and the amount of participation by each is lower. If there are more than six people, one solution is to have the extra people be present strictly as observers.

The group five or six people in the core group should be composed of developers, domain experts, and an object-oriented technology facilitator.

G.3. The Session

Before starting a session a part of the problem needs to be selected for the session to focus on. Essentially, this means picking the set of classes that are to be used.

Pick the scenarios that are to be walked through that use the classes picked above. Start with scenarios that are part of the systems normal operation first, and then exceptional scenarios, like error recover, later.

Assign each class to a member of the group. Each person should be responsible for at least one class. They are the owner of that class for the session. Each person records the name of their class on a card. One class per card.

Walk-throughs are the heart of the CRC card session. To walk through a scenario address each action in it one at a time. First decide which class is responsible for this function. The owner of the class then picks up his card and holds it up in the air. When a card is up in the air it is an object and can do things. The owner announces that he needs to fulfill his responsibility. The responsibility is refined into smaller tasks if possible. These smaller tasks can be fulfilled by the object is appropriate or they can be fulfilled by interacting with other objects (collaborators). If no other appropriate class exists, you may need to make one and assign it to someone. This is the fundamental procedure of the scenario execution.

G.4. The Process

CRC Cards are used in the Analysis and Design phases. The process for these phases differ primarily in how the classes and scenarios are chosen.

In the Analysis phase the classes and scenarios are in the problem space and generally derive from the requirements. In the Design phase solution space classes and scenarios are added. Additionally in the Analysis phase the very first session starts with no classes or scenarios to select from so a special session creates them.

Index

The use of the index in the document is done a little at random and cannot be trusted. Please help in suggesting new index entries!

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X