VisualAge Smalltalk

# UML Designer User's Guide

*Version 5.5*

**Note**

Before using this document, read the general information under "Notices" on page iii.

**August 2000**

This edition applies to Version 5.5 of the VisualAge Smalltalk products, and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product. The term "VisualAge," as used in this publication, refers to the VisualAge Smalltalk product set.

Portions of this book describe materials developed by Object Technology International Inc. of Ottawa, Ontario, Canada. Object Technology International Inc. is a subsidiary of the IBM Corporation.

If you have comments about the product or this document, address them to: IBM Corporation, Attn: IBM Smalltalk Group, 621-107 Hutton Street, Raleigh, NC 27606-1490. You can fax comments to (919) 828-9633.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

# Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, USA 10594.

IBM may change this publication, the product described herein, or both.

## Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:
- VisualAge
- UML Designer

The following terms are trademarks of other companies:
- Rational (Rational Software Corporation)
- Microsoft (Microsoft Corporation)

Windows is a trademark of Microsoft Corporation.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

# About this document

This document describes how to use the UML Designer feature. UML Designer is a tool for capturing and organizing requirements and other high-level design information as models, which can then be transformed into implementation using VisualAge for Smalltalk, or into Java source code (.java files). You can also use UML Designer to analyze and document existing Smalltalk applications.

This document is not intended to document object-oriented analysis and design. It assumes you are familiar with VisualAge, Smalltalk, OO programming, and Unified Modeling Language (UML) notation. If you are new to UML, see "References" on page vi for some recommended references.

This document is divided into the following sections:

- **"Part 1. Modeling concepts" on page 1** explains some of the basic concepts behind modeling with UML Designer. It describes the available model elements and the various transforms available to generate model elements and implementation code.
- **"Part 2. Using the UML Designer tools" on page 25** gives general information about using the UML Designer browsers and diagrammers.
- **"Part 3. Building models with UML Designer" on page 65** explains the process of building a model by following a simple example through a "forward" process of requirements, analysis, and design. If you want to get a quick start using UML Designer, you can read this chapter first, referring to the earlier chapters if you want more information.

## Conventions used in this book

This book uses several conventions that you might not have seen in other product manuals.

These highlighting conventions are used in the text:

| Highlight style | Used for | Example |
|---|---|---|
| **Boldface** | New terms the first time they are used | VisualAge uses **construction from parts** to develop software by assembling and connecting reusable components called **parts**. |
| | Items you can select, such as push buttons and menu choices | Select **Add Part** from the **Options** pull-down. Type the part's class and select **OK**. |
| *Italics* | Special emphasis | Do *not* save the image. |
| | Titles of publications | Refer to the *VisualAge Smalltalk User's Guide*. |
| | Text that the product displays | The status area displays *Category: Data Entry*. |
| | VisualAge programming objects, such as *attributes, actions, events, composite parts,* and *script names* | Connect the window's *aboutToOpenWidget* event to the *initializeWhereClause* script. |

| Highlight style | Used for | Example |
|---|---|---|
| `Monospace font` | VisualAge scripts and other examples of Smalltalk code | |
| | Text you can enter | For the customer name, type `John Doe` |

```
doSomething
  | aNumber aString |
  aNumber := 5 * 10.
  aString := 'abc'.
```

## References

For more information about Unified Modeling Language (UML), refer to the following documents:

- *UML Distilled* by Martin Fowler with Kendall Scott
- *Unified Modeling Language User Guide* by Grady Booch, James Rumbaugh, and Ivar Jacobson
- *Unified Modeling Language Reference Manual* by James Rumbaugh, Grady Booch, and Ivar Jacobson
- *UML Semantics* and *UML Notation Guide*, available from Rational Software Corporation at www.rational.com.

## Tell us what you think

The VisualAge Smalltalk web page has an online comment form. Please take a few moments to tell us what you think about this book. The only way for us to know if you are satisfied with our books or if we can improve their quality is through feedback from customers like you.

# Contents

# Part 1. Modeling concepts

# Chapter 1. Introduction to UML Designer

UML Designer is a tool for capturing and organizing requirements and other high-level design information as models, which can then be transformed into implementation using VisualAge for Smalltalk. You can also use UML Designer to analyze and document existing Smalltalk applications.

A **model** is a means of analyzing, representing, and documenting a system. The implementation of an object-oriented system can itself be regarded as an executable model. But modeling can also help explain a design in terms of higher-level abstractions (such as requirements and analysis objects) than the implementation code itself.

UML Designer is tightly integrated with the VisualAge for Smalltalk programming environment, with all models stored in the VisualAge for Smalltalk repository. This integration provides several significant benefits:

- Models are treated as software components. The VisualAge for Smalltalk repository provides configuration management and version control of all design artifacts in the model, including import, export, and reconciliation capabilities. Individual design elements, as well as entire models, can evolve through many versions, and previous versions can always be retrieved.
- Model elements can be related to their implementations, and implementations can be explained in terms of higher levels of abstraction. Traceability links make it possible to navigate between model and implementation entities; they also make it possible to propagate changes to the design model to accommodate modifications discovered during implementation — so-called “round-trip engineering”.
- The transformation from model to implementation is relatively straightforward, and the Smalltalk language makes possible an object-oriented implementation.

UML Designer provides design artifacts and tools that are flexible and can be used in many different ways. It prescribes no specific OO methodology; it is intended to support those elements of OO analysis and design that are common to the major methodologies and have generally proved most useful to working programmers. UML Designer supports an evolutionary approach to software development, where the design undergoes continual revision even after implementation has begun.

## Features

UML Designer provides capabilities to build models and to present them. With the UML Designer browsers and diagram editors, you can interactively create models and can publish them as hardcopy or HTML documents.

- **Modeling.** Distinct design elements are provided for each stage of the modeling process: requirements capture, analysis, and implementation. Elements are organized into models representing subsystems, and traceability links make it possible to maintain connections between design elements and eventual implementation elements.
- **Browsing.** Specialized browsers make it possible to organize requirements, use cases, and other OO design artifacts as named design elements. Each element can have a natural-language description and hypertext links to other related elements.

- **Diagramming.** Three graphical editors support the creation of diagrams using the Unified Modeling Language (UML) notation. Diagrams are, in effect, tightly integrated *views* of the underlying model, and they automatically reflect changes to design elements.
- **Publishing.** UML Designer can generate documentation from models and the design elements they contain. Supported output formats include HTML and RTF.

## Evolutionary approach

UML Designer provides tools that support an evolutionary approach to modeling. Traditional modeling techniques have typically assumed a "waterfall" approach—moving forwards from requirements to implementation—but this is not generally the preferred approach for OO development, which tends to be iterative. In an OO project, it is equally likely that you will need to start with an existing implementation or prototype and use modeling to reverse-engineer the analysis for documentation purposes.

UML Designer is equally suited to either the "forward" or "backward" approach, or (most typically) a mix of the two.

## Semantic models

UML Designer uses three different semantic models, each representing a different level of abstraction: requirements, analysis, and design. Each model also corresponds roughly to a distinct phase in a traditional software development process, although in reality you will likely move freely between them. Each model has its own set of model elements representing artifacts appropriate to that level of abstraction:

**Requirements model**

The requirements model provides elements you can use to capture the requirements and boundaries of the system you are designing. The requirements model focuses on the purposes and use cases for the system, and usually also describes some elements that lie outside the bounds of the system.

Requirements and use cases are formulated with the assistance of the system's intended users, and the users must also be able to verify the requirements. Therefore, the analysis in the requirements model is informal, text-based natural language.

Requirements model elements include requirements, use cases, actors, things, and responsibilities.

**Analysis model**

The analysis model moves forward from requirements by adding more detail and providing a rigorous specification of the system's required behavior.

The analysis centers around *protocols*, elements that describe behavior in terms of abstract interfaces. Protocols are implementation-independent, but they use a strict type-inheritance model that ensures rigor and internal consistency. Protocols can be generated from things and responsibilities identified during the requirements phase, generated from existing Smalltalk classes, or created manually.

Analysis model elements include protocols, message specifications, parameters, and return values.

**Design model**

The design model includes elements representing the actual system as implemented in Smalltalk. You can use these elements to annotate the Smalltalk implementation with traceability information to track how the classes conform to the protocols defined in the analysis model.

Design model elements include classes, instances, and method calls.

UML Designer includes transforms that can automatically generate model elements at one level of abstraction based on elements at another. For example, you can generate a protocol and its messages based on a thing and its responsibilities. These transforms also include the ability to generate stub implementation code in either Smalltalk or Java.

You can also create links between model elements at different levels of abstraction to trace how they evolve. These traceability links make it possible to track how your implementation conforms to your design decisions, and how your design satisfies your requirements.

# Chapter 2. Model elements

This chapter describes the model elements supported under each of the UML Designer semantic models. These elements and their relationships to one another are defined in the UML Designer **metamodel**, which can itself be represented using model elements and diagrams.

## Common model elements

### Model

A **model** is a collection of related design elements; these elements can include requirements, use cases, objects, diagrams, and other artifacts of design. In the VisualAge team library, a model is stored within the context of a Smalltalk application.

All of the modeling work connected with designing a system is within the context of a model, or a set of related models; a model can have another model as a prerequisite. Model elements in a prerequisite model are visible, meaning they can be accessed (and linked to) from the model specifying the prerequisite. Models can have “uses” and “used by” relationships to other models.

### Group

A **group** is an element you can use to collect related model elements together. Groups do not correspond to any implementation; they are simply a convenient means of organizing model elements according to arbitrary criteria. Groups are useful for filtering the elements displayed in a browser or included in a publication.

A group is also used as the underlying model element representing the system boundary in a use case diagram.

### Stereotype

A **stereotype** is a UML-defined element you can use to classify an element as belonging to a user-defined subclass of an existing element. Stereotypes are useful for indicating usage distinctions between elements of the same type, but with different intent. Stereotypes are generally important for tools and code generation.

### Diagram

A diagram is a graphical view of a model, exposing elements that convey important information about a design. A diagram exists independently of any diagrams it contains, but any changes to the underlying model are automatically reflected in affected diagrams.

### Publication

A **publication** is a model element used to generate formatted output from a model (or some subset of a model) suitable for printing or online viewing. You can create publication elements either automatically or manually. A publication, in effect, describes a textual view of a set of model elements.

A publication element contains one or more **topic** elements, each of which can contain other topics. Each topic contains a text element derived from the textual contents of a model element. The text of a topic can contain hypertext links to the text of other topics.

For more information about publishing, see “Chapter 15. Publishing models” on page 103.

# Requirements model elements

## Requirement

A **requirement** describes a function the system must perform in order to meet its fundamental business objectives. Requirements also help to define the boundaries of the system; some functions, though important, might turn out to be outside the scope of the system being developed. Requirements are formulated by the users of the system and describe what the system must do.

There are no strict rules governing what you can include as a requirement, nor what level of detail is appropriate. Indeed, the list is very likely to change over time, as you gain a better understanding of the problem domain and the system boundaries. You can use Requirement elements to capture any information that is useful for setting the system’s objectives and that you want to capture for documentation and tracking purposes. Ideally, requirements should be short, succinct, and focused on the essential purpose of the system. Often, the requirement’s title alone will suffice; other times, you might want to provide further explanation, which you can enter as hypertext.

A requirement should avoid any unnecessary descriptions of *how* the system will be implemented; instead, it should describe *what* the system must do for the user in order to work successfully. If you have quantifiable requirements that must be met, such as response times or scalability objectives, you can include these as well.

## Use case

A **use case** is a specific case of usage, tracing a particular task through from start to finish. Rather than concentrating on *how* the system functions, use cases describe *what* the system must do from the user’s perspective. A use case begins with some stimulus from someone or something outside the system (an **actor**, which we will discuss in more detail shortly).

Though more detailed than requirements, use cases are still quite informal. A use case should be a natural-language description, not pseudocode, and it should avoid any unnecessary assumptions about implementation. A use case should do the following:

- It should help to capture the purpose of the system from the user’s perspective.
- It should define the system boundaries by identifying external agents (actors).

The style and level of detail to use in writing use cases are matters of judgment and experience. The amount of detail you should include depends upon the novelty of the system and your familiarity with the problem domain. The level of detail can also vary from one use case to another: some might describe high-level interactions of real-world elements, while others might describe low-level interactions of actual system objects (when such objects are known). But try to avoid mixing levels of abstraction within a single use case.

Above all, remember that use cases are not formal; they are written in natural language rather than adhering to any rigorous, readily programmable semantics. This can lead to some imprecision or ambiguity, but it has the advantage of making the requirements readily understandable to, and verifiable by, the users of the system.

## Scenario

A **scenario** is an instance of a use case. In other words, a scenario traces a particular execution of a use case from start to finish, with specific conditions and values. While a use case might encompass several possible outcomes depending upon conditions, a scenario describes only one outcome. Consequently, there might be multiple scenarios associated with a single use case.

## Concept

A concept is any significant term or idea that is doesn’t necessarily qualify as an actor or a thing. This category can include any aspect of the system you want to capture, such as real-world objects outside the system or domain-specific terminology. You might later decide that the idea described by a concept is also an actor or a thing, in which case you can create links to indicate this relationship.

## Actor

An actor is an entity outside the system that provides a stimulus setting a use case in motion, or receiving the output from a use case. An actor is not actually part of the system, but is some real-world entity that interacts with the system. Usually, actors are human users, although they can also be other software or hardware entities that initiate actions.

## Thing

A thing (also called a **domain object**) is an entity inside the system. Things are candidates to become objects in the eventual implementation.

A thing can be transformed into a protocol. See “Chapter 3. Transforms and code generation” on page 13 for more information.

## Responsibility

A **responsibility** is a duty of a thing or actor, something it must do in order for a use case to complete successfully. Each thing or actor can have many responsibilities, and each can collaborate with other things or actors in their responsibilities. A responsibility of a thing is a candidate ultimately to become one or more methods of an implementation object or a relationship between objects.

A responsibility can be transformed into a message specification. See “Chapter 3. Transforms and code generation” on page 13 for more information.

## Use case diagram

A use case diagram is a UML-compliant diagram that gives a visual representation of the system being designed, its actors and use cases, and their relationships. For more information about use case diagrams, see “Use case diagrammer” on page 46.

# Analysis model elements

## Protocol

A **protocol** is a specified object interface, a named set of message specifications defining what messages an object must understand, what their parameters are, and what their return values will be. A protocol defines a **type** rather than a class; it does not say anything at all about implementation, only external behavior. Input parameters and return values are specified in terms of other types, which must also be defined by protocols.

A class is said to **conform** to a protocol if it implements all of the messages defined by that protocol and adheres to the specified types for input and output values. A class can conform to more than one protocol, so multiple inheritance of types is possible.

In addition, protocols can **refine** other protocols. (This is similar to, but distinct from *inheritance* among classes, which is implementation-based.) A refining protocol can add additional message specifications, but it cannot remove any. It can also refine the input and output types of the message specifications defined by the supertype, but only in specific ways:

- The input parameter types can be *less specific* than those defined by the supertype, accepting anything accepted by the supertype and more.
- The return type can be *more specific* than the one defined by the supertype, returning only a subset of the possible types returned by the supertype.

A protocol can be generated from a thing, or it can be retrieved from an existing class. See "Chapter 3. Transforms and code generation" on page 13 for more information.

### Message specification

A **message specification** is an element contained within a protocol. It defines a single message signature, including parameters and return values. It can also specify exceptions.

A message specification can be generated from a responsibility, or it can be retrieved from an implemented method. See "Chapter 3. Transforms and code generation" on page 13 for more information.

### Parameter

A **parameter** element is part of a message specification. It defines the name and type of a single message parameter. Type is specified in terms of a defined protocol. The parameter can also specify aliasing (whether the parameter is the same as the return value).

### Return value

A **return value** element is part of a message specification. It specifies the name and type of a message return value. Type is specified in terms of a defined protocol. The return value can also specify aliasing (whether the return value is the same as one of the message parameters).

### Exception

An **exception** element is part of a message specification. It specifies a named error condition that a message can raise. Identifying possible error conditions helps to ensure that they are handled.

### Class diagram

A **class diagram** is a UML-compliant diagram that shows the relationships between classes, instances, and protocols in your model. For more information about class diagrams, see "Class diagrammer" on page 47.

---

## Design model elements

### Class design

A **class design** is a model element that represents a class. A class design can be (but does not have to be) connected to a real Smalltalk class. The class design, rather than a real Smalltalk class, can then be connected to other model elements. In a class diagram, each class figure is attached to a class design rather than to an actual class.

Class designs provide an indirect coupling between your model and the actual Smalltalk implementation. During design, you might not yet be ready to start creating actual Smalltalk classes; instead, you can create class designs (which are comparatively lightweight) without any underlying Smalltalk classes. (However, you must create a real Smalltalk class if you want to create inheritance relationships between class designs.)

Furthermore, you might not yet know what your actual classes will be named, or you might want to use names other than the ones your actual classes will have (this might be the case if you are documenting legacy classes). A class design can have a different name from that of the Smalltalk class it is associated with; even if the names are different, a class design will still reflect any changes made to the underlying class.

Class designs also serve as a repository for class-related design information that is not normally captured in a Smalltalk code. For example, a class design can have a conformance link to one or more protocols, indicating that the attached class should implement all of the methods defined by the protocols. By conforming to protocols, class designs also specify information such as parameter and return types, which otherwise are not specified by a Smalltalk class definition. (On the other hand, class designs do not duplicate any information stored in the class itself, such as its methods or instance variables.)

In effect, class designs provide a bridge between analysis elements (like protocols) and implementation classes. They provide traceability links from implementation classes back to the model elements from which they are derived.

A class design combines the interface of its Smalltalk class with those of any protocols to which it conforms. Its methods, therefore, fall into two categories:

- Methods defined in a protocol to which the class design conforms. These are called **specified methods**. A specified method might or might not actually be implemented in the underlying class, if any.
- Methods defined in the underlying Smalltalk class the class design is linked to, if any. These are called **implemented methods**. An implemented method might or might not be specified by any protocols the class design conforms to.

Similarly, the attributes of a class design can be either specified or implemented (or both).

A class design can be retrieved from an existing class. See “Chapter 3. Transforms and code generation” on page 13 for more information.

## Instance

An **instance** element represents a named instance of a class design, from which it inherits its attributes. Essentially, an instance represents a sample object with state information and specific values for its attributes. Instances are useful for building examples and sequence diagrams.

## Sequence diagram

A **sequence diagram** is a visual representation of a series of interactions between the objects in your system. Unlike a class diagram, which is a representation of a static model of the system, a sequence diagram is a representation of the dynamic interaction of your system, serialized over time. For more information about sequence diagrams, see “Sequence diagrammer” on page 51.

### Method call

A **method call** appears within a sequence diagram and represents a message send from one object to another, and the passing of control from the sender to the receiver.

### Method instance

A **method instance** (or activation) appears within a sequence diagram and represents the active execution of a method, beginning with a message send and ending with a return.

### Method return

A **method return** appears within a sequence diagram and represents the return of control from a method that has finished executing.

# Chapter 3. Transforms and code generation

Each of the three UML Designer semantic models (requirements, analysis, and design) represents a different level of abstraction, and each has its own set of model elements. Although these elements represent distinct objects, in many cases there is a logical correspondence between an element at one level of abstraction and an element at the next level. For example, a Thing element (requirements model) typically corresponds to a Protocol element (analysis model), which in turn corresponds to a class (design and implementation model).

UML Designer provides transform capabilities that can map between model elements at a different levels of abstraction. Transforming takes the selected element, creates its corresponding element or elements, and automatically creates a traceability link between them.

Transforms exist for both the “forward” mapping (requirements → analysis → design) and the “backward” mapping (reverse engineering). “Forward” transforms include:

- Generating protocols from things (going from requirements to analysis); this includes generating protocol message specifications, associations, and attributes from responsibilities
- Code generation (going from analysis to design and implementation); this includes generating classes and methods from class designs and protocols

“Backward” transforms include:

- Retrieving protocols from Smalltalk classes and class designs; this includes retrieving protocol message specifications from Smalltalk class methods
- Creating things from protocols

## Going from requirements to analysis

The first phase in a “forward” development process, capturing requirements, yields elements such as requirements, use cases, things, and actors. Of these, things are most important from a transform perspective, because things represent domain objects (and potential implementation classes). By transforming things into protocols, you can move from requirements capture to analysis.

The **Generate Protocol** transform maps things to protocols and responsibilities to protocol message specifications. The default mapping of things to protocols is one-to-one: transforming a thing into a protocol results in the creation of a Protocol element that, by default, has the same name as the thing. For example, a thing called *Car* would result in a protocol called <*Car*>. You can rename the generated protocol while still maintaining its traceability link back to the original thing.

The default mapping of a responsibility of a thing depends upon the idiom of the responsibility; a responsibility and its participants will be mapped, according to its idiom, into one or more message specifications and parameters. In this way, you can go from the informality of responsibilities to the relative rigor of message specifications.

You can transform responsibilities collectively or one at a time. You can also modify the generated message specifications afterward.

## Responsibility idioms

An **idiom** specifies a mapping of a responsibility into one of several predefined implementations as message specifications. For a given responsibility, you can choose one of four idioms:

**Action**
A responsibility to perform an arbitrary action or operation. This is a general-purpose, nonspecific idiom that describes something the thing *does*.

**Reference**
A responsibility to keep a value that refers to another thing. This idiom describes something the thing *knows*.

**Value** A responsibility to keep a value as an attribute. This idiom describes something the thing *keeps*.

**Identifier**
A responsibility to keep a value that can be used to uniquely identify the thing. This idiom describes an attribute the thing can be *identified by*.

For example a *Customer* thing might have the following responsibilities:

| Idiom | Responsibility |
|---|---|
| Action | pay bill |
| Reference | sales rep |
| Value | account balance |
| Identifier | customer number |

An idiom provides guidance to UML Designer regarding how the responsibility should be transformed into protocol message specifications and attributes. For each idiom, there is a corresponding set of messages that would typically be used to implement the responsibility. (Essentially, these are simple **patterns**: designs for implementing common programming requirements.) Each idiom results in a different combination of messages and attributes in the generated protocol, based on what would typically be used for such a responsibility.

*Table 1. Summary of idioms and how they affect protocol generation*

| **Idiom** | **Messages generated** | **Participants** | **Attributes generated** |
|---|---|---|---|
| Action | one message based on responsibility name | optional; each participant becomes a message parameter | None |
| Identifier | getter and setter methods based on responsibility name | Optional; if no participant specified, assumes similarly named protocol or <*Object*> | One attribute, type specified by participant |
| Reference | • If max cardinality=1, getter and setter methods<br>• If max cardinality=many, collection API (*add*, *remove*, and getter methods) | Required (one or more) | • One attribute, type specified by participant (or <*Collection*> if max cardinality=many)<br>• Also generates an association between the protocols |
| Value | • If max cardinality=1, getter and setter methods<br>• If max cardinality=many, collection API (*add*, *remove*, and getter methods) | Optional; if no participant specified, assumes similarly named protocol or <*Object*> | One attribute, type specified by participant (or <*Collection*> if max cardinality=many) |

## Action

**Action** is the simplest idiom, and describes an arbitrary, user-defined responsibility. It is the default idiom for a new responsibility.

The **Action** idiom specifies a simple, one-to-one transform from a responsibility to a protocol message specification. For each **Action** responsibility, UML Designer generates a single message. The name of the message is the responsibility's implementation name, which by default is derived from the responsibility name; for example, *pay bill* becomes *payBill*. (You can change the implementation name if you prefer a different name.)

Participants are optional for an **Action** responsibility. If there are any participating things in a responsibility, each participant becomes a message parameter whose type is the default type derived from the participant. Message parameters must be specified as protocols, so if a protocol does not yet exist for a participant, UML Designer automatically generates an empty protocol for it.

You can specify a maximum cardinality of **Many** or **1**. If you specify **Many**, the default generated message name will be plural.

**Question for Nick:** Does cardinality have any other effect on an Action responsibility?

No attributes or associations are generated for an **Action** responsibility.

## Reference

A **Reference** responsibility specifies a value that the thing knows that is itself another thing. In effect, this specifies an association for the implementing protocol.

At least one participant is required for a **Reference** responsibility; the participant identifies the thing the responsibility refers to. In the generated protocol, UML Designer generates an attribute to hold the reference; its type is that of the implementing protocol for the participating thing. (If you specify more than one participant, any of the specified types are permitted.)

In addition to the attribute, UML Designer generates a set of messages to get and set the attribute's value. The specific behavior varies depending upon the maximum cardinality you specify (**Many** or **1**):

- If you specify a maximum cardinality of **1**, the protocol corresponding to the participant is used as the generated attribute's type. (If multiple participants are specified, there is still only one attribute, which can be of any of the specified types.) In addition, UML Designer generates getter and setter messages for the attribute, using the appropriate protocols as the allowed types for their parameters and return values.
- If you specify a maximum cardinality of **Many** (the default), the generated attribute will be a collection, which can contain multiple items of the allowed types. In this case, UML Designer generates a getter method for the collection, along with *add* and *remove* messages for the elements in the collection.

In addition, for a **Reference** responsibility, UML Designer creates an association between the generated protocol and any participating protocols. This is only an association between protocols; it does not automatically become an association between classes during the design phase.

### Value

A **Value** responsibility describes a value that the thing knows. In some ways, this is similar to a **Reference**, but in this case the thing being pointed to is not necessarily interesting in its own right.

For each **Value** responsibility, UML Designer generates a protocol attribute to hold the value, as well as a set of message specifications to get and set the attribute's value.

Specifying a participant is optional for a **Value** responsibility. If you specify a participant, the corresponding protocol is used as the type of the generated attribute and of the message parameters and return values. If no protocol exists for the participant, UML Designer automatically generates one.

If you do not specify a participant, UML Designer checks for an existing thing with a similar name to that of the responsibility. If it finds a match, it uses the corresponding protocol as the type of the attribute. If it does not find a match, it assumes <*Object*>. (See "Finding protocols for participants" on page 17 for more information.)

In addition to the attribute, UML Designer generates a set of messages to get and set the attribute's value. The specific behavior varies depending upon the maximum cardinality you specify (**Many** or **1**):

- If you specify a maximum cardinality of **1** (the default), the protocol corresponding to the participant is used as the generated attribute's type. (If multiple participants are specified, there is still only one attribute, which can be of any of the specified types.) In addition, UML Designer generates getter and setter messages for the attribute, using the appropriate protocols as the allowed types for their parameters and return values.
- If you specify a maximum cardinality of **Many**, the generated attribute will be a collection, which can contain multiple items of the allowed types. In this case,

UML Designer generates a getter method for the collection, along with *add* and *remove* messages for the elements in the collection.

No associations are generated for a **Value** responsibility.

### Identifier

An **Identifier** responsibility describes a value that the thing knows, and by which it can be uniquely identified.

For each **Identifier** responsibility, UML Designer generates a protocol attribute to hold the identifier, as well as getter and setter messages to access the attribute's value.

Specifying a participant is optional for an **Identifier** responsibility. If you specify a participant, the corresponding protocol is used as the type of the generated attribute and of the message parameters and return values. If no protocol exists for the participant, UML Designer automatically generates one.

If you do not specify a participant, UML Designer checks for an existing thing with a similar name to that of the responsibility. If it finds a match, it uses the corresponding protocol as the type of the attribute. If it does not find a match, it assumes <*Object*>. (See "Finding protocols for participants" for more information.)

**Note:** An identifier is, by definition, of a single specified type; if you specify more than one participant for an **Identifier** responsibility, only the first is used when transforming to a protocol.

No associations are generated for an **Identifier** responsibility.

## Java versus Smalltalk conventions

In the system settings, you can specify that you want to generate Java code instead of Smalltalk classes. If you select this option, UML Designer uses different conventions when generating accessor messages from a responsibility.

The differences between the Smalltalk conventions and the Java conventions are as follows:

| Smalltalk conventions | Java conventions |
|---|---|
| • *add* and *remove* messages specifications return the added or removed object<br>• colons are used to indicate keyword parameter positions | • *add* and *remove* message specifications do not return the added or removed element<br>• colons are not used to indicate keyword parameter positions |

## Finding protocols for participants

When you transform a responsibility into a message specification, the types of the generated parameters and return values must be specified as protocols. Depending upon your selections, UML Designer can use several different methods of choosing which protocol to use for a parameter or return value.

If a responsibility specifies a participating thing, UML Designer uses the implementing protocol of the thing as the type for the corresponding message parameter.

However, participants are optional for responsibilities using the **Value** or **Identifier** idioms, even though some of the generated messages take parameters. If you do not specify a participant, UML Designer chooses a default parameter type as follows:

1. If you have specified English link labeling in the system settings, UML Designer looks for a noun in the responsibility implementation name. For example:
   - *name* in *customerName*
   - *date* in *dateOfBirth*
   - *indicator* in *ownerIndicator*

   If you have not specified English link labeling, UML Designer uses the entire implementation name. (For more information about the system settings, see “System settings” on page 35.
2. Using the noun or implementation name, UML Designer then looks for a Thing whose name matches. If it finds one, and the thing has an implementing protocol, it uses that protocol. For example:
   - *name* matches *Name*, a Thing element in the Kernel model; its implementing protocol is<*String*>.
   - *indicator* matches *Indicator*, a Thing element in the Kernel model; its implementing protocol is <*Boolean*>.
3. If no Thing element matches, UML Designer then looks for a protocol whose name matches the noun or implementation name. If it finds one, it uses that protocol.

   For example, *Date* matches the protocol <*Date*>.
4. If Relationships Browser cannot find a matching thing or protocol, it uses <*Object*>.

## Going from analysis to design

The analysis phase of a “forward” development process yields protocols and message specifications. These elements can in turn be used to generate implementation code.

Although much of the actual implementation of your program logic is still up to you, some of the structure of objects and methods can often be deduced from protocols and their message specifications. When you transform a protocol into a class design, UML Designer can also generate an actual Smalltalk or Java class with attributes and stub methods based on the messages of the protocol. Depending on the idioms of the protocol messages, it might also be able to generate default implementation code for the method.

UML Designer can generate both class and instance methods, although if you are generating Java there are some limitations on class attributes. Class methods are specified by a class conformance relationship between the protocol and the implementation class; instance methods are specified by an instance conformance relationship. (You can specify class or instance conformance when you transform a protocol to a class design.)

### Message idioms

As with the responsibilities of things, protocol message specifications are characterized by idioms; however, message idioms are distinct from responsibility idioms. When you transform a protocol into a class, message idioms control the stub method implementations generated for the protocol's messages.

Although message idioms are different from responsibility idioms, there are correspondences between them. When you generate a protocol from a thing, UML Designer automatically assigns an idiom to each generated message specification, using the responsibility's idiom to determine which idiom to assign to the messages. (You can also manually select the idiom for a message specification.)

| **Idiom of responsibility** | **Idioms of generated messages** |
|---|---|
| Action | General Message |
| Identifier | • Getter<br>• Setter |
| Reference | If max cardinality=1:<br>• Getter<br>• Setter<br>If max cardinality=Many:<br>• Getter<br>• Add<br>• Remove |
| Value | If max cardinality=1:<br>• Getter<br>• Setter<br>If max cardinality=Many:<br>• Getter<br>• Add<br>• Remove |

For each message idiom, UML Designer generates a different stub implementation when transforming to a class. This section gives examples of both the Smalltalk and Java code generated for each message idiom.

## General Message

The **General Message** idiom is the most straightforward message idiom. It does not make any assumptions about implementation, so it is appropriate for any message. The method generated for a **General Message** idiom is effectively empty, allowing you to add whatever logic you need. However, the generated method includes comments that identify the parameter and return types for the method as guidelines during implementation.

The following code shows the structure of the Smalltalk method generated from a **General Message** protocol message with one parameter.

```
doSomething: p1

        "Do something for the receiver

        PARAMETERS
                p1 : <Object>
        RETURNS
                <Object>"
        "Put user defined code here."
```

Following is the Java code generated from a **General Message** protocol message with one parameter.

```
/**
 * Do something for the receiver
 *
 * @param aString
 *
 */
public void doSomething (String aString) {

/* Put user defined code here. */


}
```

## Add

The **Add** idiom describes a message that adds an item to a collection. This idiom is automatically assigned to the “add” messages generated for a responsibility with maximum cardinality of many. An **Add** message must have one parameter representing the object to be added. A protocol containing an **Add** message must also contain an attribute for the collection (this attribute is transformed into an instance variable).

The following code shows the structure of the Smalltalk method generated from an **Add** protocol message.

```
addMessage: p1
        "Add the argument, p1, to the receiver's collection of aVariable.

        PARAMETERS
                p1 : <Object>
        RETURNS
                <Object>"

        self aVariable add: p1.

        ^p1.
```

Following is the Java code generated from an **Add** protocol message.

```
/**
 * Add the argument, p1, to the receiver's collection of aVariable.
 *
 * @param p1
 *
 */
public void addMessage (Object p1) {
this.aVariable.addElement(p1);

}
```

## Getter

The **Getter** idiom describes a message that returns the value of an attribute. A **Getter** message cannot take any parameters, and the protocol must contain an attribute to contain the attribute whose value is returned (this attribute is transformed into an instance variable).

The following code shows the structure of the Smalltalk method generated from a **Getter** protocol message.

```
getterMessage
        "Answer the receiver's aVariable.

        PARAMETERS
                -none-
        RETURNS
                <Object>"

        ^aVariable.
```

Following is the Java code generated from a **Getter** protocol message.

```
/**
 * Answer the receiver's aVariable.
 *
 */
public void getGetterMessage () {
return this.aVariable;

}
```

## Remove

The **Remove** idiom describes a message that removes an item from a collection. This idiom is automatically assigned to the "remove" messages generated for a responsibility with maximum cardinality of many. A **Remove** message must have one parameter representing the object to be removed. A protocol containing a **Remove** message must also contain an attribute for the collection (this attribute is transformed into an instance variable).

The following code shows the structure of the Smalltalk method generated from a **Remove** protocol message.

```
removeMessage: p1
        "Remove the argument, p1, from the receiver's collection of aVariable.

        PARAMETERS
                p1 : <Object>
        RETURNS
                <Object>"

        self aVariable remove: p1.

        ^p1.
```

Following is the Java code generated from a **Remove** protocol message.

```
/**
 * Remove the argument, p1, from the receiver's collection of aVariable.
 *
 * @param p1
 *
 */
public void removeMessage (Object p1) {
this.aVariable.removeElement(p1);

}
```

## Setter

The **Setter** idiom describes a message that sets the value of an attribute. A **Setter** message requires one parameter, and the protocol must contain an attribute to contain the attribute whose value is being set (this attribute is transformed into an instance variable).

The following code shows the structure of the Smalltalk method generated from a **Setter** protocol message.

```
setterMessage: p1
        "Set the receiver's aVariable to the argument p1.

        PARAMETERS
                p1 : <Object>
        RETURNS
                <Object>"

        aVariable := p1.

        ^p1.
```

Following is the Java code generated from a **Setter** protocol message.

```
/**
 * Set the receiver's aVariable to the argument p1.
 *
 * @param p1
 *
 */
public void setSetterMessage (Object p1) {
this.aVariable = p1;

}
```

## Reverse engineering

If you have existing Smalltalk code, you can use UML Designer transforms to generate analysis and requirements elements. These “backward” transforms can retrieve class designs from classes, protocols from class designs, and things from protocols. These transforms can be useful if you want to document an existing system.

**Note:** UML Designer does not currently support reverse engineering Java code.

### Retrieving class designs

Retrieving classes creates Class Design elements based on existing Smalltalk classes. This transform can create a class design for a single class, a selected group of classes, or all the classes in an application. This transform is available from the Relationships Browser and from the Class Diagrammer; see “Retrieving multiple classes at once” on page 97 for more information.

The class or classes you want to retrieve must be visible from within your model, which means they must be either in the model application or in a prerequisite application. For each selected class, UML Designer creates a Class Design element, with a traceability link back to the class as the real implementing class for the class design.

You can also create a class design and manually link it to an existing class, either from the Relationships Browser or the Class Diagrammer.

### Retrieving protocols

For a class design that has a real implementing class, you can retrieve one or more protocols. (You can also do this automatically while retrieving classes.) You can retrieve a single protocol specifying all the methods of the class, or multiple protocols, each specifying the methods in a particular category.

When you retrieve a protocol from a class design, UML Designer creates a new Protocol element containing a message specification for each of the class design’s methods. It also creates a traceability link between the protocol and the conforming class design. You can also select methods for retrieval by category; this is useful if you want to retrieve only some of the methods, or if you want to retrieve the methods into several distinct protocols. UML Designer creates conformance links between the class design and all of the protocols to which it conforms.

When you retrieve a protocol, UML Designer attempts to assign types to the generated message parameters based on the names given to the parameters in the Smalltalk code, if a matching protocol exists in the model or a prerequisite model. For example, a method parameter called *aString* would result in a protocol message parameter of type <*String*>.

## Retrieving things

You can also reverse engineer requirements elements by retrieving things from protocols. For each selected protocol, UML Designer creates a Thing element with a traceability link to the protocol. The generated Thing element is empty (no responsibilities are generated).

# Part 2. Using the UML Designer tools

# Chapter 4. Using the browsers

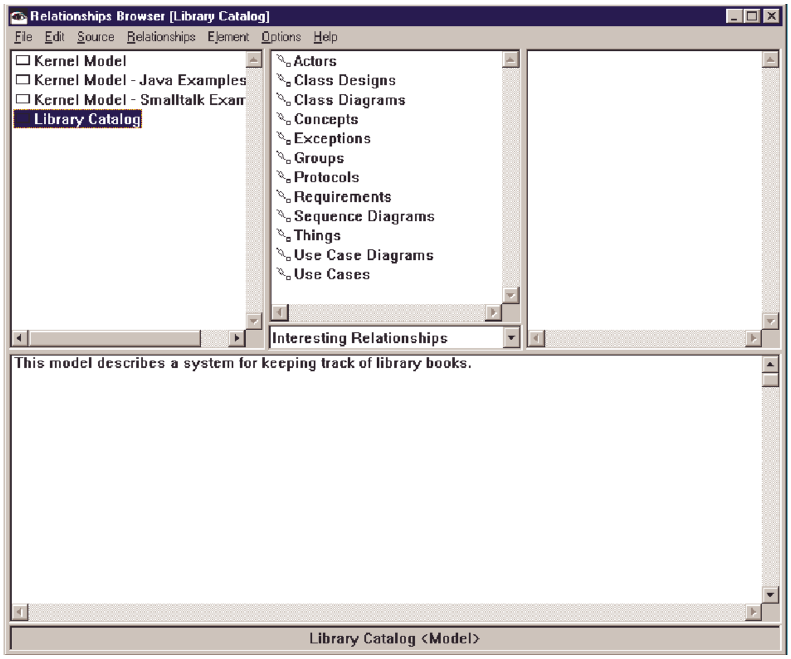
UML Designer provides three browsers you can use to view, navigate, and edit your model elements:

- The Relationships Browser
- The Path Browser
- The Hierarchy Browser

All three browsers display model elements and their contents, but each displays the relationships between elements differently.

## The Relationships Browser

The Relationships Browser is the heart of UML Designer; you can use it to browse and edit all of your model elements, and it has sophisticated filtering capabilities that can help you work with the kinds of elements and relationships you’re interested in.



This model describes a system for keeping track of library books.

With the default settings, the Relationships Browser has three panes. The leftmost pane shows the available source elements; the source element is the starting point for navigating in the browser. If you opened the browser from the Transcript window, the leftmost pane lists all of the models in your image, and you can select any one of them as the current source element.

When you select a source element, the middle pane of the Relationships Browser then shows a list of possible relationships between that element and other elements. For example, if you select a model in the leftmost pane, the middle pane shows a list of relationships between the model and the elements it contains, such as class designs, diagrams, use cases, and requirements. These relationships are the predefined relationship types defined by the UML Designer metamodel.

The drop-down list at the bottom of the middle pane selects the relationship filter used to control which of the possible relationships are shown. Some types of model elements have a large number of possible relationships, so this filter is useful for viewing a subset of them. The default relationship filter is **Interesting Relationships**, which shows only the most useful relationships; most of the time, you can leave the filter set to this setting. Other settings can show other subsets of the possible relationships, or all of the possible relationships.

**Note:** The relationship filter is cumulative with other UML Designer filtering options, which are available from the menu bar. See “Filtering” on page 32 for more information.

The bottom pane of the Relationships Browser is the hypertext pane. This pane contains any descriptive text that applies to the model, relationship, or model element currently selected. You can use the hypertext pane to write explanatory text describing each element; the text can include hypertext links to other related model elements.

When you select one of the relationships from the middle pane, the rightmost pane shows all of the model elements that result from following the selected relationship from the selected source element. For example, if you select *Library Catalog* in the leftmost pane and *Class Diagrams* in the middle pane, the rightmost pane lists all of the class diagrams defined in the *Library Catalog* model.

## Spawning a new browser

From any UML Designer browser, you can open a new browser with a selected element or elements as the source elements; this is called **focusing**. Focusing is useful for following a long series of navigations, or getting a different view of an element (for example, opening a Hierarchy Browser from a Relationships Browser). You can focus on any element displayed in a browser (a source element or a target element). There are several ways to do this:

- Double-click on an element to open the default Focus browser for that element. (For most elements, the default is the Relationships Browser, although for some it is different; for example, the default browser for a Publication element is the Hierarchy Browser.) The selected element is source element in the new browser.
- Select one or more elements and then select **Focus** from the pop-up menu. This also opens the default Focus browser for the selected elements, but this way you can include multiple elements in the Source pane of the new browser.
- Select one or more elements and then select **Open With** from the pop-up menu. A cascaded menu appears from which you can select from all of the possible UML Designer browsers (and any applicable Smalltalk browsers). Use this method if you want to use a browser other than the default Focus browser (for example, if you want to open a Path Browser on a protocol).

### Spawning a new browser on a relationship

You can also open a new browser to display the results of following a specified relationship. There are two ways you can browse a relationship:

- To browse the destinations of a relationship, select a relationship and then select **Browse→Destinations** from the pop-up menu. A browser opens showing the destinations of the selected relationship in the source pane.
- To browse the connections for the relationship, select a relationship and then select **Browse→Connections** from the pop-up menu. A browser opens showing the connections to the destination in the source pane.

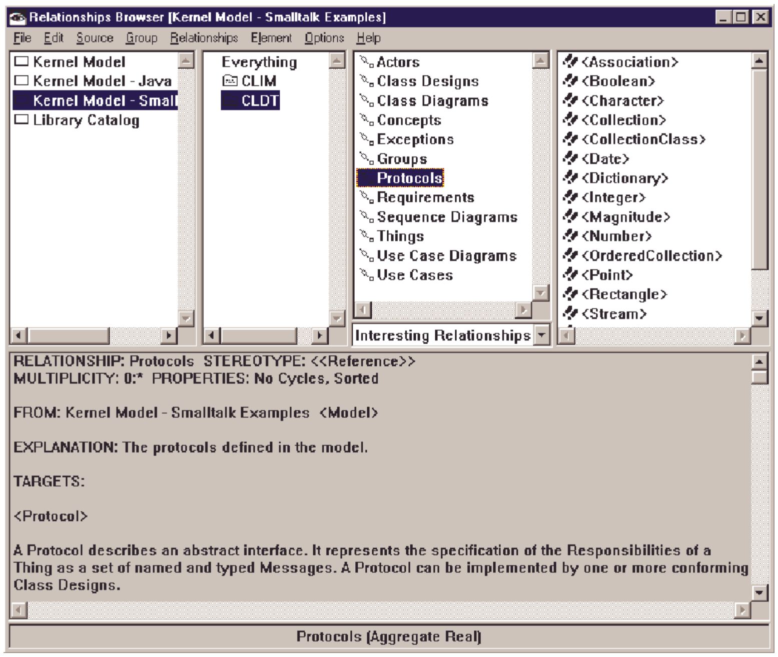
You can select connection elements for navigating, deleting, and editing just as you can other model elements. This makes it possible to browse and edit the properties of an association without opening a diagrammer.

## Browsing groups

If you use groups to organize your model elements, you can have the Relationships Browser display groups in an additional, separate pane. This makes it easier to use groups to work on a subset of the elements in your model.

To show groups in the Relationships Browser, follow these steps:

1. Select **System Settings** from the **Options** menu.
2. In the System Settings window, go to the **Browser** page and select **Show Groups**.
3. Close and reopen the Relationships Browser.



```
RELATIONSHIP: Protocols  STEREOTYPE: <<Reference>>
MULTIPLICITY: 0:*  PROPERTIES: No Cycles, Sorted

FROM: Kernel Model - Smalltalk Examples  <Model>

EXPLANATION: The protocols defined in the model.

TARGETS:

<Protocol>

A Protocol describes an abstract interface. It represents the specification of the Responsibilities of a
Thing as a set of named and typed Messages. A Protocol can be implemented by one or more conforming
Class Designs.
```

The additional second pane lists the groups in the current model. Select a group to limit the browser to displaying only elements in the selected group; if you want to see all elements, select the default group *Everything*. If you select a group, any elements you create will be added to the selected group automatically.

## Browsing refinement and inheritance

The Relationships Browser can optionally display refinement relationships (including inheritance) in a hierarchical tree view. To enable this option, select **Show Refined By button** on the **Browser** page of the system settings. (See “System settings” on page 35 for more information.)

If you select this option, refining elements show up in the browser as nested children in either the source pane or the destination pane. If an element has children, a plus sign (+) button appears beside the element in the list. Select this button to expand or collapse the list and display the refining elements.

There are several kinds of refining elements:

- Prerequisites of a model

- Refining protocols or messages
- Subclassing class designs
- Extending use cases

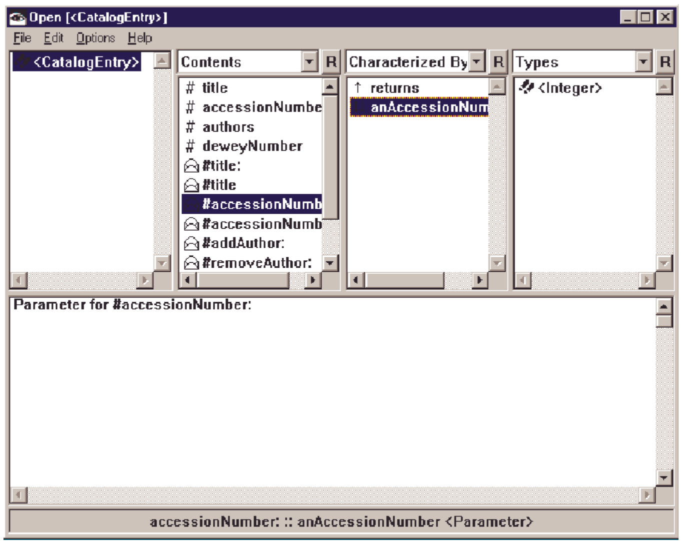
## The Path Browser

The Path Browser is similar to the Relationships Browser, and it shows essentially the same information. However, it offers a more concise layout and a larger context. The Path Browser is helpful when you need to work with nested elements or complex relationships, because it allows you to view multiple layers of navigation within a single browser.

The Path Browser shows the successive navigation of not just one relationship, but of up to four relationships, starting from the selected source element. For example, you can use the Path Browser to browse a protocol, its message specifications, the parameters of a selected message, and the type of a selected parameter, all in the same browser.

There are two ways to open the Path Browser:

- In the Transcript window, select **Path Browser** from the **UML Designer** menu.
- In any UML Designer browser or diagrammer, select a model element and then select **Open With→Path Browser** from the pop-up menu.



Parameter for #accessionNumber:

accessionNumber: :: anAccessionNumber <Parameter>

The leftmost pane of the Path Browser shows the source element. If you opened the Path Browser from the Transcript window, the source element is a model (you can select any available model from the list). If you opened the Path Browser from a UML Designer browser or diagrammer, the source element is the element you selected before opening the Path Browser.

Each successive pane of the Path Browser shows the elements satisfying the selected relationship to the selected element in the previous pane. For example, if you select a model in the leftmost pane and the **Contents** relationship in the second pane, the second pane shows all of the elements contained in the model. You can then select an element in the second pane and use the third pane to follow another navigation (for example, you could browse the messages of a protocol).

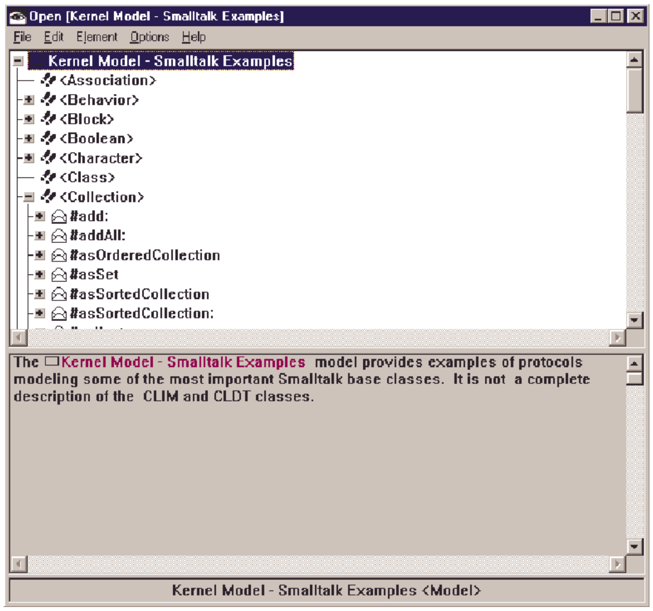
The relationships available in each pane of the Path Browser are the same as the relationships available in the Relationships Browser, and all of the same filtering options are available; select the **R** push button beside the relationship drop-down list to select a filter. As with the Relationships Browser, the visible relationships are also affected by UML Designer global filtering options (see “Filtering” on page 32 for more information).

## The Hierarchy Browser

The Hierarchy Browser presents model elements in a hierarchical tree view, with elements arranged according to their containment relationships. This view is similar to what you might see in a file browser. The Hierarchy Browser does not show the actual relationships as elements; instead, it shows only the source and destination model elements, arranged hierarchically from parent to child. The Hierarchy Browser is particularly useful for browsing and editing nested elements such as Publication elements. (See “Chapter 15. Publishing models” on page 103 for more information.

There are two ways to open a Hierarchy Browser:

- In the Transcript window, select **Hierarchy Browser** from the **UML Designer** menu.
- In any UML Designer browser or diagrammer, select a model element and then select **Open With→Hierarchy Browser** from the pop-up menu.



The Kernel Model - Smalltalk Examples model provides examples of protocols modeling some of the most important Smalltalk base classes. It is not a complete description of the CLIM and CLDT classes.

The Hierarchy Browser has two panes; they can be arranged either vertically or horizontally, depending upon the settings on the **Browser** page in the system settings (see “System settings” on page 35 for more information). The first pane (either top or left) shows the hierarchical tree view of model elements. The top-level element is the element you selected before opening the Path Browser; if you opened the browser from the Transcript window, all of the existing models are listed as top-level elements.

If an element has any important relationships to other elements (such as containment, conformance, or collaboration), a plus sign (+) appears beside the element in the list. Click on the **+** to expand the list and see the related elements. If an element is linked to by multiple others, it might appear multiple times in the list.

---

# Filtering

The Relationships Browser and the Path Browser have powerful filtering functions that control the model elements and level of detail displayed. By using filtering, you can limit the browsers to displaying only the model elements you're interested in, based either on your relative level of advancement or your current task.

There are two ways of filtering the browser contents:

- Filtering by task
- Filtering by browser level

You can use both kinds of filtering at the same time.

## Filtering by task

To filter by task, select **Tasks** from the **Options** menu. This displayed a cascaded menu from which you can select one of the following tasks:

**All** Not actually a task, **All** specifies no filtering. This is the default setting and causes all relationships and elements to appear (subject to other selected filters).

**Requirements**
Select the **Requirements** task filter to see only relationships and elements in the requirements model (such as things and use cases).

**Analysis**
Select the **Analysis** task filter to see only relationships and elements in the analysis model (such as protocols).

**Design**
Select the **Design** task filter to see only relationships and elements in the design model (such as class designs).

**Diagramming**
Select the **Diagramming** task filter to see only relationships and elements related to creating and publishing diagrams (such as diagram elements).

**Organizing**
Select the **Organizing** task filter to see only relationships and elements related to organizing your model elements (such as groups).

## Filtering by browser level

The Relationships Browser and the Path Browser can display relationships and elements according to five different levels of detail. To change the browser level, select **Levels** from the **Options** menu; this displays a cascaded menu from which you can select the browser level you want to use. This setting globally affects all Relationships Browsers and Path Browsers.

The higher the browser level, the more relationships appear in the Relationships Browser and Path Browser. These levels are cumulative: each browser level includes all of the relationships displayed at the lower levels, plus additional relationships. Generally, you should use the lowest browser level that includes all of the relationships you want to use, in order to avoid unnecessary clutter.

**Note:** Filtering by browser level works in conjunction with other UML Designer filtering mechanisms (such as filtering by task or selecting a relationship filter in the Relationships Browser). To see all of the relationships included at the current browser level, you must do the following:

- Select **Tasks→All** from the **Options** menu to disable task filtering.
- Select **All Relationships** in the Relationships Browser.

## Level 1: Basic

Browser level 1 includes only the basic structural relationships of the model. This includes relationships that lead to semantically interesting direct children of the source element (containment relationships); it also includes associations of the source element.

It does not include cross-linking relationships, relationships that lead only to ″view″ elements such as diagrams and publications, or cross-referencing and traceability relationships (see Level 2 for more information).

For example, Level 1 includes:

- Actors
- Things and responsibilities
- Protocols and messages
- Simple associations

## Level 2: General

Browser level 2 adds additional semantic model relationships, as well as relationships to ″view″ elements such as publications and diagrams. It also includes semantically important **cross-linking** relationships (direct references other than containment and hypertext relationships, such as a “satisfies” link from a use case to a requirement), as well as a few **cross-reference** relationships (direct and indirect connections that help in understanding the model, such as traceability and “used by” links). This level is the default when you install UML Designer.

For example, Level 2 includes:

- Diagram elements
- Exceptions
- Conformance relationships
- Collaboration relationships

## Level 3: Cross-reference

Browser level 3 adds additional relationships that show how model elements depend upon one another. This can be helpful with a large or complex model. For example, Level 3 includes:

- Dependencies between models
- Constraints
- Hypertext references between elements

## Level 4: Meta

Browser level 4 adds some additional relationships that make it possible to navigate some of the metamodel elements associated with your model element. This includes the filters themselves, each a relationship in the metamodel (you can select a filter to see the relationships available with that filter).

### Level 5: Advanced

Browser level 5 includes all of the public UML Designer relationships. This includes relationships that show the available elements of a particular type in the model, according to the scoping criteria of the element and its parent relationships.

For example, level 5 includes:

- Available messages
- Available return values
- Available responsibilities
- Available publication topics

---

# Checking consistency

Some circumstances can cause a model to have inconsistencies, such as missing link destinations, out-of-scope references, or missing prerequisites. For example, inconsistencies can be caused by loading a back-level edition or making changes to a model with standard ENVY browsers rather than the UML Designer browsers.

UML Designer provides a tool to check for inconsistencies. To check consistency, select the model or element you want to check and then select **Check Consistency** from the pop-up menu. (You can also select **Check Consistency** from the Source or Element menu of the Relationships Browser, depending upon where the element you want to check appears. Checking an element also checks all of its children.

When the consistency check finishes, a window appears listing any errors and asking whether you want to browse them. You have two choices:

- Select **Yes** to open a Hierarchy Browser from which you can repair the errors manually.
- Select **No** to have UML Designer attempt to fix the errors automatically. If any cannot be repaired automatically, UML Designer will then open a Hierarchy Browser to display the remaining errors so you can repair them manually.

## Browsing inconsistencies

If you select **Yes**, a Hierarchy Browser opens listing each error, with the involved elements listed as children of the error. The text pane of the browser gives details about the error and suggests what you might do to fix it.

Depending upon the nature of the error, there may be two different ways of correcting it.

- To correct the error by deleting elements, select the error in the browser and then select **Make Consistent By Deleting** from the pop-up menu. (This option is available only if deleting an element is a viable option for correcting the problem and will not create additional errors.)
- To attempt to repair the error automatically, select the error and then select **Make Consistent By Repairing** from the pop-up menu. UML Designer will attempt to repair the error; an error message appears if it is unable to do so.

## Repairing inconsistencies

Following are the inconsistencies you are most likely to encounter:

| Error Message | Explanation | Possible Repairs |
|---|---|---|
| Missing Object On Diagram | A diagram points to one or more missing objects. | • Reload the missing objects.<br>• Delete the figure. |
| Already Deleted | An object has been marked as deleted but is still in the system in an inconsistent state. | Reload the object. |
| Missing Destination | One of the objects in a relationship is missing. | • Load the missing object.<br>• Unlink from the missing object. |
| Missing Parent | The parent of an object is missing. | Reload the parent. |
| Orphan | An object is not referenced by any other object (it has no parent), possibly because of damaged links. | Repair any damaged links. If no links are damaged, delete the orphaned object. |
| Out Of Scope | A relationship references an element that is not visible (not in the same model or one of its prerequisites). | • Move the referenced element so it becomes visible.<br>• Change the prerequisites.<br>• Remove the link. |
| Missing Back Reference Relationship | One element points to another, but the second does not point back to the first. | Select **Make Consistent By Repairing** to automatically create the inverse link. |
| Modified But Not Saved | An element contains changes that have not been saved. | Save the element. |
| No Released Edition | An element has not been released to its parent. This can happen if you version elements with ENVY browsers. | Select **Make Consistent By Repairing** to release the currently loaded edition. |
| Association Attachment Link Error | The source or destination of an association is incorrect. | • Correct the association.<br>• Delete the association. |

# System settings

In any of the three UML Designer browsers, you can select **Options→System Settings** to display the UML Designer System Settings window. From this window you can control options that affect how the browsers and diagrams display information and other configurable UML Designer settings. Each tab in the System Settings window controls a different aspect of the UML Designer environment:

**UI** Controls general user-interface options. This includes default filtering for the Relationships Browser and Path Browser, as well as general options for diagramming and UML Designer dialogs.

**Browser**
Controls the appearance of the UML Designer browsers. This includes

whether the **Groups** pane appears in the Relationships Browser, whether to show refining elements, and the layout of the Hierarchy Browser.

**Color/Font**
Controls the appearance of the screen fonts used in the UML Designer browsers and diagrammers. You can specify fonts for three different kinds of emphasis: hypertext links, figure captions, and missing object captions.

**General**
Controls options related to default link labels and consistency checking. See “Association labeling” on page 41 for more information.

**Syntax**
Controls the syntax used to parse and display protocol messages. You can select any one of the following options:

- Standard UML syntax
- Smalltalk-style syntax
- Java-style syntax

**Code Generation**
Controls options related to transforms that generate implementation classes. These options affect:

- The default prerequisite for any new models
- The specification of protocol messages generated from responsibilities
- The availability and appearance of transform dialogs
- The target language for generated code (Java or Smalltalk)

# Chapter 5. Using the UML Designer diagrammers

UML Designer supports the creation of three types of UML-compliant diagrams:
- Use case diagrams
- Class diagrams
- Sequence diagrams

Each type of diagram presents models in a different way, and each has a separate diagramming tool. However, the three diagrammers (and the three types of diagrams) share many common characteristics.

## Diagrams

Each model element in a UML Designer diagram is represented by a **figure**, a graphical representation based on UML notation. A figure is attached to an underlying model element, from which it takes most of its properties. Normally, each model element can be represented by only one figure in a single diagram.

A figure also has additional characteristics of its own that control its visual appearance. For example, the figure controls which aspects of the underlying model element appear on the diagram, as well as purely visual attributes such as color.

From a diagrammer, you can edit both the visual attributes of the figures; you can also open browsers (using **Open With**) and make changes to the underlying model elements.

A diagram has two types of figures:
- A **node** figure represents a structural model element (such as an actor, use case, or class design).
- A **connector** figure joins two node figures and represents a link or association between model elements. Each connector figure is **owned** by one of the node figures it connects to.

**Note:** The term **link** here refers to a semantic connection between two objects (in UML terms, a link is an instance of an association). This is distinct from a **hypertext link**, which is not a formal semantic connection but simply a navigational reference. Hypertext links are not represented on diagrams.

Figures can also have **adornments**, which are labels and other decorations that carry semantic meaning on the diagram. The visual attributes of a figure (its font, color, and position on the diagram) exist independently of the underlying model.

Adornments help to make a class diagram more understandable, and they convey additional information about the model elements, including additional UML semantics; for example, they can show the navigability and multiplicity of an association. You can specify whether or not most adornments appear on a figure; however, the properties described by adornments exist in the model, whether or not the adornments themselves are displayed.

Using the UML Designer diagrammers, you can build models in two different ways:

- You can add figures for existing model elements. This technique is useful for documenting an existing model.
- You can add new figures and create new model elements for them at the same time. You can use this technique to build a new model graphically.

## Using the diagrammers

In order to build a diagram, you must first create a Diagram element in your model. To create a diagram, do one of the following:

- In the Relationships Browser (or another UML Designer browser), select a model and then select **New→Class Diagram**, **New→Sequence Diagram**, or **New→Use Case Diagram** from the pop-up menu.
- In the middle pane of the Relationships Browser, select the type of diagram you want to create and then select **New** from the pop-up menu.

To open a diagram for editing, double-click on the Diagram element in the browser.

### Adding a node figure

There are several ways to add a node figure to a diagram:

- Manually creating a new node figure and attaching it to a model element (selected from a list or named directly)
- Copying a node figure from another diagram along with its attached model
- Copying a model element from a browser and pasting it into the diagram as a figure
- Selecting **Hide/Show Relationships** to show an existing or candidate relationship to an element not already on the diagram

Once you have added a figure, you can move it on the drawing surface to make your diagram more readable.

#### Manually creating a node figure

To manually create a node figure and attach it to a model element, follow these steps:

1. Select the figure you want from the tool bar. (If you prefer, you can select **Create Node Figure** from the **Tools** menu and then select the figure you want from the cascaded menu.)
2. Position the mouse pointer where you want the figure to appear (in an empty area of the drawing surface) and then click mouse button 1.

   When first added, an unattached figure is labeled with a question mark (**?**).
3. If you know the name of the model element you want to attach the figure to, hold down the Alt key and click mouse button 1 on the figure label to directly edit the label. UML Designer automatically attaches the figure to the element that matches the name you specify.
4. If you want to select from a list of available elements, select **Attach** from the pop-up menu of the figure. In the window that appears, select the model element you want to attach to the figure. (You can also specify that you want to create a new element.)

   **Note:** This list does not normally include any model elements already represented on the diagram.

### Copying a node figure from another diagram

To copy an existing node figure from another diagram, follow these steps:

1. Select the figure you want to copy in the source diagram.
2. Select **Copy** from the **Edit** menu, or from the pop-up menu.
3. Go to the target diagram and place the mouse pointer in the location where you want to paste the figure.
4. Select **Paste Here** from the pop-up menu. (You can also select **Paste** from the **Edit** menu and then move the figure where you want it.)

The figure is copied along with its visual properties (color and font), and the new copy is attached to the same model element as the original figure. (If the attached element is in a different model from that of the diagram, the model name appears in brackets.)

### Copying a model element to a node figure

To copy a model element and paste it as a new node figure, follow these steps:

1. In the Relationships Browser or another UML Designer browser, select the element you want to create a figure for.
2. Select **Copy** from the **Element** menu, or from the pop-up menu.
3. Go to the target diagram and place the mouse pointer in the location where you want to paste the figure.
4. Select **Paste Here** from the pop-up menu. (You can also select **Paste** from the **Edit** menu and then move the figure where you want it.)

The new figure is automatically attached to the model element you selected. (If the attached element is in a different model from that of the diagram, the model name appears in brackets.)

### Adding a node figure with Hide/Show Relationships

You can use the **Hide/Show Relationships** choice on the pop-up menu of a node figure to create connector figures for existing or new relationships (see “Adding a connector with **Hide/Show Relationships**” on page 40 for more information). If the element at the other end of the relationship is not already represented by a node figure, UML Designer automatically creates one and adds it to your diagram.

## Adding a connector figure

There are several ways to add a connector figure to a diagram:

- Manually creating a connector figure between two node figures
- Copying a connector figure from another diagram
- Copying a relationship from a browser and pasting it as a connector figure
- Displaying an existing or candidate connection by selecting **Hide/Show Relationships** on the pop-up menu of a node figure

To reroute the path of a connector figure in a class diagram or use case diagram, select the figure and then drag its selection handles.

### Manually creating a new connector figure

To add a connector figure to a diagram, follow these steps:

1. Select the figure you want from the tool bar. (If you prefer, you can select **Select Relationship Figure** from the **Tools** menu and then select the figure you want from the cascaded menu.)

2. Position the mouse pointer on the figure representing the source (owner) of the relationship. For example, a class design owns its conformance relationships to protocols; either node can be the owner of a simple association.
3. Click mouse button 1 to connect the source end of the relationship.
4. Move the mouse pointer to the figure representing the destination of the relationship.
5. Click mouse button 1 to complete the relationship.

## Copying a connector figure from another diagram

To copy an existing connector figure from another diagram, follow these steps:

1. Select the connector figure you want to copy in the source diagram.
2. Select **Copy** from the **Edit** menu, or from the pop-up menu.
3. Go to the target diagram.
4. Select **Paste Here** from the pop-up menu or **Paste** from the **Edit** menu.

The connector figure is copied along with its adornments. If both endpoints already appear on the diagram, the connector automatically connects them; if either or both are missing, they are copied along with the connector figure.

## Copying a relationship to a connector figure

To copy a relationship between model elements and paste it as a new connector figure, follow these steps:

1. In the Relationships Browser, select the relationship you want to create a figure for. (You can select a relationship by browsing connections; see “Spawning a new browser on a relationship” on page 28 for more information.)
2. Select **Copy** from the **Source** menu, or from the pop-up menu.
3. Go to the target diagram.
4. Select **Paste Here** from the pop-up menu or **Paste** from the **Edit** menu.

A new connector figure is created for the relationship you selected. If either or both of the elements involved in the relationship are not yet represented in the diagram, node figures for them are created as well.

## Adding a connector with Hide/Show Relationships

You can create connector figures for existing or candidate relationships. To do this, follow these steps:

1. Select the node figure you want to connect to.
2. Select **Hide/Show Relationships** from the pop-up menu.

   A window appears listing any existing relationships (associations and links) involving the selected element, followed by any identified candidate relationships. A candidate relationship is identified with a plus sign (**+**).

   A candidate relationship is a relationship that doesn’t yet exist in the model, but can be created automatically. Candidate relationships include commonly used “default” relationships. By selecting a candidate relationship, you can quickly create the relationship, a connector figure for the relationship, and in some cases, a node figure for the other endpoint. For example, you can create (and link to) a new instance of a class design.
3. Select the relationship for which you want to create a connector figure from the **Hidden Relationships** list. (You can also select multiple relationships at once.)
4. Select **>>** to add the relationship to the **Shown Relationships** list.
5. Select **OK**.

If you select a relationship that requires a new model element (for example, a conformance link to a protocol that doesn't yet exist), UML Designer prompts you for confirmation before creating the new element. If either endpoint can be the owner of a relationship (as with a simple association), UML Designer prompts you for confirmation of the direction.

After you provide any necessary confirmations, the new connector figure (and new node figure, if necessary) appear on the diagram. You can now rearrange the diagram to place the new figures where you want them.

## Direction and ownership of associations

For any connector, one of its endpoints (the nodes to which it is attached) is its **owner** (or **source)**. All model information about the connector is stored and versioned as part of the owning model element. By default, the owning end of a connector is indicated in a diagram by a solid semicircle, although you can disable this indicator in the system settings.

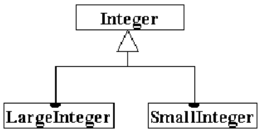
For most connectors, ownership is determined by the semantics of the relationship; for example, a conformance link is always owned by the conforming class design. For simple associations, however, ownership is arbitrary, and either endpoint can be the owner. The source element (the element you start with when creating the association) is automatically designated the owner of the association. (This is true for associations created in either the browsers or the diagrammers.) Once you create an association, you cannot change the direction of ownership except by deleting and re-creating it starting with the new source element.

**Note:** Direction of ownership is distinct from the UML concept of navigability, which is represented on connector figures by arrowheads. Navigability represents a UML semantic describing whether an association can be traversed in a given direction in the implementation.

## Lamination

**Lamination** refers to the merging of multiple connector figures into one in order to reduce clutter. Lamination is useful in situations where multiple connector figures share the same source or destination. Currently, lamination is only available for inheritance and conformance relationships.

Lamination is automatic when you add multiple connector figures using the **Hide/Show Relationships** menu option.

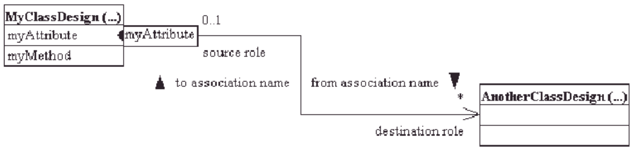


## Association labeling

Text labels on associations are one type of adornment. Labels can show the name of an association (for each direction), as well as the role names of the elements in the association. Any or all of the following labels might appear on an association:

- The "to" association name, which initially defaults to the name of the destination element
- The "from" association name, which initially defaults to the name of the source element (the owner of the association)

- The source role name
- The destination role name



There are a couple of ways you might decide to name an association:

- One style is to name the association with a verb describing the relationship, making it possible to read the association (usually from left to right) as a sentence.
- Another style is to name the association with a noun referring to the destination entity, with plurality appropriate for the multiplicity of the association. This is the default for the labels UML Designer automatically generates.

If you want the automatically generated labels to reflect the multiplicity of your associations, select the **Autoplural** option on the **General** page of the system settings.

## Display properties

There are numerous display properties you can control for each diagram figure. The available options vary between different kinds of figures.

### General display properties for node figures

The following options are available on most node figures:

- Select **Figure Properties** to open a dialog from which you can specify fonts and colors for the figure.
- Select **Display Options→Move To Front** or **Display Options→Move To Back** to control whether a figure appears in front of or behind any other figures it overlaps.
-

  Select **Display Options→Use Figure Specific Name** if you want to give the figure a name different from that of the attached model element.

  If you select **Use Figure Specific Name**, you can then change the name of the figure by selecting **Rename** from the pop-up menu. (If **Use Figure Specific Name** is not selected, this option renames the figure and the attached model element.)
- Select **Display Options→Show Stereotype** to show the stereotype of the attached element above the figure name. (This option is available only if the attached element is linked to a stereotype.)
- Select **Display Options→Show Model If Different** to show the name of the model that contains the attached element, if different from the model containing the diagram.
- Select **Display Options→Show Properties** to show the UML properties string for the figure.

### General display properties for connector figures

The following options are available on most connector figures:

- Select **Figure Properties** to open a dialog from which you can specify fonts and colors for the figure.
- Select **Role→Edit From** or **Role→Edit To** to edit the role names of the source and destination endpoints.
- Select **Role→Show From** or **Role→Show To** to control whether the source and destination role labels are displayed.
- Select **Navigation** to specify the navigability of the connector. You can specify any one of the following navigability options:
  - **None** (no navigability)
  - **From** (navigable toward the source element)
  - **To** (navigable toward the destination element)
  - **Both** (navigable in either direction)

  Select **Navigation→Show Navigation Arrow** to control whether navigability is indicated on the diagram by an arrowhead.
- Select **Show All Labels** or **Hide All Labels** to control whether any labels appear on the connector figure.

## Alignment

In addition to manually moving figures on a diagram, you can also use the Format menu to automatically arrange your figures for readability or neatness.

- Select **Align Horizontal** to arrange two or more selected figures in a straight horizontal line. You can align the top edges, bottom edges, or centers of the figures.
- Select **Align Vertical** to arrange two or more selected figures in a straight vertical line. You can align the left edges, right edges, or centers of the figures.
- Select **Align To Grid** to arrange one or more selected figures to the alignment grid. (The grid is not visible, but it provides a quick way to align figures to preset rows and columns.)
- Select **Distribute By Edges** to arrange three or more figures so the space between them is equal. You can select to arrange the figures horizontally (select **Left To Right**) or vertically (select **Top To Bottom**).
- Select **Distribute By Centers** to arrange three or more figures so the space between their centers is equal. You can select to arrange the figures horizontally (select **Left To Right**) or vertically (select **Top To Bottom**).

You can also interactively align figures by selecting **Format** from the Format menu. This opens the Format window, from which you can select the alignment options you want to apply to the selected figure or figures.

## Deleting figures

There are two ways to delete a figure from a diagram.

- To delete the figure but leave the attached model element, select the figure and then select **Delete Figure** from the pop-up menu (or press the Backspace key).

  If you use this option, you can re-add the deleted figure, and only its visual properties are lost.
- To delete the figure and its attached model element, select the figure and then select **Delete Figure And Model** (or press the Del key).

  If you use this option, you cannot re-add the figure unless you also re-create (or reload) the element it represents.

## Diagram synchronization

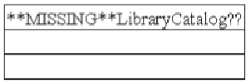
UML Designer automatically keeps each diagram synchronized with its underlying model. The diagram is updated if an element is deleted or changed, or if certain properties are changed (for example, the types of message parameters shown on a class figure). You can also force a resynchronization at any time by selecting **Refresh Browser** from the **Diagram** menu.

If you make changes to a model element while editing a diagram, your changes are immediately committed to the underlying model, even if you do not save the diagram. To undo changes to a model element, load the previous edition of the element (see “Loading elements” on page 56 for more information). Changes that affect only the visual properties of a diagram (such as figure placement, fonts, and colors) are saved only when you save the diagram. (If you make any such changes, UML Designer prompts you to save the diagram when you close it.)

**Note:** The appearance of a diagram can change if you change display properties (such as fonts) in the system settings, or if you change an aspect of an element that affects how it appears (such as the length of its name). When this happens, the diagrammers attempt to readjust the diagram appropriately to accommodate the changes. In some cases, some connector figures may appear to become unconnected to their endpoint node figures. If this happens, just drag any loose endpoints to the correct locations.

### Missing objects

When you open a diagram, the figures are checked against the underlying model elements. If an element has been deleted or moved, its figure indicates a missing object:



To repair a ″missing object″ error, do one of the following:

- If you want to keep the figure, re-create or reload the missing element.
- If you no longer need the figure, delete it.
- If the element has been moved to another model, use the **Reattach** menu choice to attach the figure to the relocated element.

## Panning and zooming

The visible area of the diagrammer’s drawing surface can be larger or smaller than your actual diagram. You might want to zoom in on a complex diagram in order to make it easier to read, or you might want to zoom out in order to see a large diagram in its entirety. There are several ways to control what area of your diagram is visible in the diagrammer.

**Note:** You can always get back to the default pan/zoom settings by selecting **Reset Pan/Zoom** from the **Tools** menu. To center the diagram without changing the zoom settings, select **Center**.

### Zooming by percentage

To set the zoom percentage directly, select **Zoom Percentage** from the **Tools** menu and then select a zoom percentage, from 10 (one-tenth normal size) to 200 (twice normal size). You can also select **Fit To Window** to automatically set the zoom percentage to the maximum value that allows the diagram to fit within the current diagrammer window.

### Dynamic pan/zoom

To dynamically change the pan/zoom settings and immediately see the results, select **Pan/Zoom** from the tool bar or from the **Tools** menu. The mouse pointer changes to a crosshair to indicate pan/zoom mode.

To pan the diagram, click and drag with mouse button 1. To zoom the diagram, click and drag with mouse button 2 (drag up or left to zoom out, right or down to zoom in). To reset to default pan/zoom settings, press the space bar.

### Overview window

Select **Overview** from the **Diagram** menu to open a small window showing a large-scale overview of the entire diagram, with a superimposed rectangle representing the currently visible area.

To pan with the overview window, click mouse button 1 within the rectangle and drag it to the part of the diagram you want to view. To zoom, use mouse button 1 to drag one of the selection handles at the corners of the rectangle to change the size of the visible area. These changes are dynamically reflected in the main diagrammer window.

## Creating GIF files

To create a GIF graphic of your diagram, follow these steps:

1. Set panning and zooming so the diagram appears in the diagrammer window exactly as you want it to appear in the GIF file. Only the visible areas of the diagram are saved to GIF. (Select **Zoom Percentage→Fit To Window** from the Tools menu to automatically bring the entire diagram into view.
2. Select **Save As Gif File** from the **Diagram** menu.
3. Specify the location and file name for the GIF graphic.

   **Note:** If you plan to use this GIF in a publication, remember that the publication uses the file name last used for the diagram at the time the publication is generated. If you later use a different file name for the diagram, you will need to regenerate the publication in order to pick up the new GIF file in the output document.
4. Select **OK** to create the GIF file.

## Printing

There are several ways to print a diagram:

-

  Select **Print** from the Diagram menu to print the entire diagram at full size on the default printer. If the diagram is too large to fit on a single page, it will be printed on multiple pages which can then be assembled to show the complete diagram.

  Select **Print Preview** to see how the diagram will be printed using this option.
- Select **Print Window** to print the portion of the diagram currently visible in the diagrammer window at the displayed zoom percentage.
- Select **Print Scaled** to print the entire diagram, scaled to fit on a single page.

You can control the following printing-related options:

- Select **Diagram Print Setup** to select a printer, or to control printer-specific options (such as orientation).

- Select **Headers And Footers** to control the headers and footers that appear on the printed pages.
- Select **Show Page Boundaries** to see page boundaries in the diagrammer window. This can help you arrange a multipage diagram.
- Select **Include Blank Pages** to control whether blank pages should be included for parts of a multipage diagram where no figures appear.

## Use case diagrammer

A use case diagram shows a visual representation of your use cases and actors and the relationships among them. In addition, a use case diagram shows the system boundary (represented in UML Designer by a Group element).

The following tools are available in the Use Case Diagrammer:

| | |
|---|---|
| ↖ | **Selection**: Allows selection and manipulation of existing figures. |
| ✥ | **Pan/zoom**: Controls the size and position of the viewing area. |
| □ | **System**: Creates a System figure, representing the grouping of elements that make up the system. |
| ⬯ | **Use case**: Creates a use case figure. |
| 웃 | **Actor**: Creates an actor figure. |
| 🗈 | **Annotation**: Creates a figure representing a note, comment, or constraint. |
| \| | **Association**: Creates a "uses" link between use cases. |
| ↱ | **Extension**: Creates an "extends" link between use cases. |
| ⋯ | **Constraint annotation**: Creates a link between a constraint annotation and a design element. |
| Sticky ☐ | **Sticky**: Allows creation of multiple figures without reselecting on the tool bar. |

Each system figure in a use case diagram represents a system or subsystem and is attached to a Group model element. You cannot attach a system figure to a group that already contains use cases.

Placing a use case figure within a system figure also adds the use case element to the corresponding group. If you move the use case figure outside the system figure, the use case element is removed from the group. A use case figure outside a system figure has a dotted border; a use case inside a system figure has a solid border.

Select **Hide/Show Relationships** from the pop-up menu of a use case figure to select from a list of existing and candidate relationships to other model elements. Candidate relationships between use cases and actors, indicated by a plus sign (**+**), are chosen based on the hypertext links that occur in the text of the use case. If you select a candidate relationship, UML Designer prompts you to confirm the direction of the relationship before creating it.

## Class diagrammer

A class diagram is a visual representation of the class and protocol objects in your system and their relationships to one another. For example, a class diagram can show:

- classes and methods
- protocols and message specifications
- class instances
- associations between classes, protocols, or instances
- protocol conformance
- instantiation
- dependencies
- refinement and subclassing

The Class Diagrammer provides sophisticated capabilities for filtering which aspects of your model elements you want to appear on the diagram, as well as the level of detail displayed. For example, you can control which attributes and methods are shown for class and protocol figures, and whether type information is included (and in what syntax).

The following tools are available in the Class Diagrammer:

**Selection**: Allows selection and manipulation of existing figures.

**Pan/zoom**: Controls the size and position of the viewing area.

**Protocol**: Creates a figure representing a protocol.

**Class**: Creates a figure representing a class design.

**Object** : Creates a figure representing an object instance.

**Annotation**: Creates a figure representing a note, comment, or constraint.

**Inheritance**: Creates a figure representing an inheritance relationship between classes.

**Conformance** : Creates a figure representing a conformance relationship between a class and a protocol.

**Dependency:** Creates a figure representing a dependency relationship between a class and a protocol.

**Association**: Creates a figure representing a simple association relationship between classes.

**Aggregation**: Creates a figure representing an aggregation relationship between classes.

**Constraint annotation**: Creates a link between a constraint annotation and a design element.

**Sticky** : Allows creation of multiple figures without reselecting on the tool bar.

### Class figure display properties

In addition to the general display properties, the following display properties are available for class and instance figures:

- Select **Display Options→Name Only** to show only the name of the class. This option hides any additional information (such as method names).

If you select **Name Only**, you can also select **Display Options→Use Small Margins** to make the figure as compact as possible.

These options is also available on instance figures.

- Select **Display Options→Method Name Only** to show only the names of the methods of the class. This option hides additional information about methods, such as parameters and return types.
- Select **Display Options→Show Scope** to show scope indicators for methods and attributes. Scope indicates whether a method or attribute is public, private, or protected.
- Select **Display Options→Show Visibility** to show visibility indicators for methods and attributes. Visibility is affected by whether the methods are specified in a protocol or implemented in a real class.
- Select **Display Options→Attribute Name Only** to show only the names of attributes and not their types, scope, or visibility.
- Select **Display Options→Method Name Only** to show only the names of methods and not their parameters, return values, scope, or visibility.

## Filtering

The methods displayed in a class figure are **method models**; these are temporary model elements, each representing a method. The list of method models is built from the implemented methods of the real class, as well as the message specifications in any protocols the class conforms to.

Likewise, the displayed attributes are **attribute models**, based on the variables of the real class together with attributes specified in any protocols the class conforms to.

A method or attribute defined in a protocol is **specified**; a method or attribute implemented in a real class is **implemented**. There are therefore four possible combinations:

| | **Implemented** | **Unimplemented** |
|---|---|---|
| **Specified** | Protocol and class | Protocol only |
| **Unspecified** | Class only | Neither |

In addition, a method or attribute is characterized by **scope** (whether it is defined for the class or its instances) and **visibility** (public, private, or protected).

You can filter which methods appear on a class figure based on whether it is specified or implemented, based on its scope, or arbitrarily. The following options are available from the pop-up menu:

- Select **Filtering Options** to open a window from which you can specify which categories of attributes and methods you want to appear in the figure. You can filter based on the following criteria:
  - Specified/unspecified
  - Implemented/unimplemented
  - Scope (instance/class)
  - Visibility (public/private/protected)
- Select **Methods** to limit which methods appear in the figure. You can show only specified methods, a limited number of methods, or all methods. Any hidden methods are indicated with an ellipsis (**...**).

- Select **Attributes** to limit which attributes appear in the figure. You can show only specified attributes, a limited number of attributes, or all attributes. Any hidden attributes are indicated with an ellipsis (**...**).

## Connector display options

In addition to the general display properties for connector figures, the following properties are available from the pop-up menus of connector figures on a class diagram:

- For a unidirectional connector (such as a conformance or refinement link), select**Name→Show** to show a name label for the connector. If you want to change the name of the connector, select **Name→Rename**.
- For a bidirectional connector (such as a simple association):
  - Select **Name→Show From** to show the name of the association from the source element's perspective. To change this name, select **Name→Rename From**.
  - Select **Name→Show To** to show the name of the association from the destination element's perspective (the inverse association). To change this name, select **Name→Rename To**.

  In addition to the association names, you can also display the stereotype of the association in each direction by selecting **Name→Show Stereotype From** and **Name→Show Stereotype To**. These options are available only if the corresponding name labels are displayed.
- For an association, select **Multiplicity→Show From** or **Multiplicity→Show To** to control whether multiplicity labels appear on the source and destination ends of the connector.

  To change the multiplicity of the association, select **Multiplicity→Edit From** or **Multiplicity→Edit To**. You can also select from several commonly used multiplicity values:
  - **0..1 To 1**
  - **0..1 To 0..***
  - **0..* To 0..1**
  - **0..* To 0..***
- For an association, select **Show From Qualified Attributes** or **Show To Qualified Attributes** to show qualifiers on the association. These options are available only if you have linked constrained attributes in the relationship properties. (See "Relationship properties" for more information.)

## Relationship properties

To edit the properties of a relationship, double-click on the connector figure or select **Focus** from its pop-up menu. This opens a window from which you can edit the properties of the relationship represented by the connector figure. Note that these properties apply to the relationship model, rather than to the connector figure (although some of them might affect the visual appearance of the figure).

The relationship properties window has three pages: one for general properties, and one for each role (direction) of the relationship. Depending upon the type of relationship, some of the options might not be available.

For associations, you can use the properties window to specify **constraints** on the relationship. A constraint is a specific rule that restrains the allowed behavior of the relationship, such as cardinality (multiplicity). Constraints should be observed in your implementation.

## Main page

The following settings are available on the Main page of the relationship properties window:

**Link Name**
The name of the relationship for the indicated direction. (The name you specify appears only if the Name label is displayed.)

**Text**
The text for the relationship in the indicated direction. This text will appear in any publications that include the relationship.

## Role page

The following settings are available on each Role page of the relationship properties window:

**Id**
Specifies a unique internal identifier for the role.

**Role Name**
Specifies the role name that will appear on the diagram if role name labels are displayed.

**Stereotype**
Specifies any user-defined stereotype that applies to the relationship. You can select any available stereotype from the drop-down list. To define a new stereotype, add a Stereotype element to the model; you will then need to close and reopen the relationship settings in order to see the new stereotype.

**UML Stereotype**
Specifies the UML-defined stereotype that applies to the relationship. You can select from a list of four UML-defined stereotypes.

**Is Unique**
Specifies whether each instance of this relationship must be uniquely identifiable.

**Is Immutable**
Specifies whether an instance of this relationship can be altered once it is created.

**Composition**
Indicates whether the relationship represents an aggregation or composition association. Aggregation is indicated on the connector figure by an open diamond; composition is indicated by a solid diamond.

**Aggregation** indicates that the aggregated element is a child of the aggregating element. **Composition** is a stronger form of aggregation, indicating that the child element is an inherent part of the composing element and cannot be meaningfully separated.

**Min/Max**
Specifies the multiplicity for this end of the association. (This is the same as selecting **Multiplicity→Edit From** or **Multiplicity→Edit To** from the pop-up menu of the connector figure.

Multiplicity is defined in termed of a minimum cardinality and a maximum cardinality. The minimum cardinality (0–*n*) defines the minimum number of instances required; the maximum cardinality (0–*n* or * for many) defines the maximum number of instances allowed.

**Ordering**

Specifies whether the set of instances of a relationship are considered ordered or sorted:

- **Ordered**: The instances are ordered in the sequence in which they are added.
- **Sorted**: The instances are ordered according to some kind of sorting (such as alphabetically).
- **Unspecified**: The ordering of the instances is not specified.

**Cycles** Specifies whether cyclic references are allowed using this relationship:

- **No Cycles**: Cyclic references are not allowed. If element A references element B, element B may not reference element A.
- **Indirect**: Indirect cycles are allowed, but direct cycles are not. In other words, if element A references element B, B may not reference A; however, B may reference another element that references A.
- **No Check**: Both direct and indirect cycles are allowed.

**Delete Check**

Specifies whether this relationship has deletion constraints:

- **Cascade**: If the source is deleted, the destination should also be deleted.
- **Control**: The destination may not be deleted as long as the source refers to it.
- **No Check**: No deletion constraints apply.

**Implementation**

Specifies properties that do not affect the semantics of the relationship but do affect your implementation:

- **Is By Reference**: The relationship is implemented by value rather than by reference (for example, by holding a collection of values or keys).
- **Is Navigable**: The relationship can be traversed in the direction of the role. That is, given an instance of the source element, it is possible to obtain an instance of the destination element through some implementation of the relationship (this can be either physical, such as a collection or database table, or derived).
- **Is Derived**: The relationship is computed indirectly from other relationships or attributes. A derived association is indicated on a diagram by a backslash (\) in front of its name label.

**Attributes**

Opens a window from which you can specify any constrained attributes (qualifiers) for this end of the association. You can select any attribute defined in the class or in any protocol it conforms to; you can also create new attributes from this window. Qualifications appear in a box at the end of the connector figure, if the option is selected to display qualified attributes.

## Sequence diagrammer

A sequence diagram is a visual representation of a sequence of interactions between objects. A sequence diagram generally does not show all of the possible interactions in your system; instead, it illustrates the interactions associated with the implementation of a particular use case or scenario. A sequence diagram can help you validate your design by confirming that it accommodates your use cases. It can also be a useful for documenting a system in which messages must be sent in a particular order to work correctly.

The objects depicted in a sequence diagram are instances rather than classes. Each object figure in a sequence diagram must be attached to an Instance model element.

**Note:** Instance elements can reside in a different model from their class designs. When you create a new instance element from a diagrammer, the new instance is located in the same model as the diagram, which might not be the same model as the class design being instantiated.

Each instance on a sequence diagram is represented by an instance figure at the top of the diagram with a dotted **lifeline** extending downward. Message sends are indicated as horizontal lines between object lifelines. Each message send can have conditions and has a corresponding return value.

An object is considered active from the time it receives a message to the time it returns control to the sender. This is indicated on the diagram by an **activation**, a vertical bar along the active object's lifeline between the message send and return lines. Only an active object can send messages.

When you first open the Sequence Diagrammer, the drawing surface is empty except for the system activation along the left edge. The system activation represents the actor or object that initiates the sequence of messages shown in the sequence diagram.

The following tools are available in the Sequence Diagrammer:

**Selection** : Allows selection and manipulation of existing figures.

**Pan/zoom**: Controls the size and position of the viewing area.

**Object** : Creates a figure representing an object instance.

**Method call**: Creates a figure representing a message send from one object to another.

**Annotation**: Creates a figure representing a note, comment, or constraint.

**Constraint annotation**: Creates a link between a constraint annotation and a design element.

**Sticky** : Allows creation of multiple figures without reselecting on the tool bar.

## Creating instances

To add an instance figure to a sequence diagram, follow these steps:

1. Select **Object** from the tool bar.
2. Use mouse button 1 to place the new figure in an empty area of the diagram. The object figure automatically appears at the top of the diagram, so only the horizontal position is important.
3.

Select **Attach** from the pop-up menu of the new object figure. A window appears listing the available class designs.

4. Select the class of the object. and then select **OK**.

It is important to remember that objects in a sequence diagram are instances, not classes. Therefore, when you select a class design to attach, UML Designer automatically creates a new anonymous instance of the selected class and attaches the figure to the new instance.

If you want to use a named instance rather than an anonymous instance, deselect **Is Anonymous** on the instance figure's pop-up menu. You can then select **Rename** to change the name of the instance. (If you rename an anonymous instance, **Is Anonymous** is automatically deselected.

## Creating method calls

To create a method call from one object to another, follow these steps:

1. Select **Method call** from the tool bar.
2. For a call to an instance method, click mouse button 1 on the sender's lifeline within an activation.

   For a call to a class method, hold down the Alt key and click mouse button 1 on the sender's lifeline within an activation.
3. Click mouse button 1 on the receiver's lifeline.

   **Note:** For a call to *self*, click mouse button 1 on the same object's lifeline for both sender and receiver.
4. In the dialog that appears, specify the message you want to send:
   - In the **Classes** list, select the class that defines the method you want to call. The list includes the class of the receiver as well as any classes it inherits from.

     **Note:** The list might include some candidate class designs, indicated by a plus sign (**+**). These class designs do not yet exist in the visible models, but it will be created automatically if you select one of their methods. This happens if the real Smalltalk class of the selected object inherits from ancestors which do not yet have class designs.
   - In the **Protocols** list, select the protocol that specifies the method you want to call. The list includes all protocols to which the selected class conforms. (Select **<unspecified>** if the method you want to call is not specified in a protocol.)
   - In the **Messages** list, select the message you want to send.
5. Select **OK**. The new method call and its return appear as a method call figure. In addition, the receiver's lifeline now shows activation between the message send and return.

### Moving and resizing method call figures

After you create a method call figure, you can then move it and resize the activation to make room for other method calls.

To move a method call figure, select the activation and then drag it up or down with mouse button 1. This moves the entire method call, including the activation and return call.

To resize an activation, select the activation and then drag its bottom selection handle. This changes the amount of space between the method send and the return; you can do this to make room for more method calls.

### Conditions and iteration

A method call can have a **condition**, an annotation (written in natural language or pseudocode) that limits the conditions under which the message is sent.

To add or change the condition of a method call, select the message send figure (the arrow going from sender to receiver) and then select **Edit Condition** from the pop-up menu. Type the text of the condition (for example, *x>0*) and press Enter.

**Iteration** indicates that the message is sent multiple times to multiple receivers (for example, iterating over a collection). To indicate iteration, select the message send figure and then select **Is Iterated** from the pop-up menu. An asterisk (*) appears beside the message send label to indicate iteration.

# Chapter 6. Configuration management and version control

UML Designer design artifacts such as actors, use cases, and diagrams are stored in the VisualAge repository, which provides versioning and management capabilities. You can use the normal VisualAge (ENVY) team programming capabilities for library management tasks such as importing, exporting, and replication.

As with Smalltalk code, the library maintains a fine-grained edition history of individual elements, allowing recovery of previous versions and configurations, as well as identification of differences between editions. Version reconciliation is similar to that used for Smalltalk code; however, UML Designer provides two new browsers that are specifically designed to help with managing editions of your models.

This section describes the special considerations that apply to team development using UML Designer.

## Editions and versions

Version control of models is similar to version control of Smalltalk code, but with some differences to accommodate the extra subcomponents of the UML Designer metamodel. Generally speaking, you can think of a model as an application with a single class representing the model; in fact, a model is stored as an ENVY application, with model elements stored within private storage classes.

For version control purposes, the elements in the model behave somewhat similarly to the methods of a class, in that they are editioned and released automatically when you save changes. Unlike methods, though, some model elements can also have child elements. For example, a protocol contains message specifications, which in turn contain parameters. To see the parent-child relationships among elements, use the Contents filter in the Relationships Browser.

In order to make changes, you must have an open edition of the model. Each time you save changes to a model element, a new edition is created for that element, and the parent element points to the new edition. A model (like a class) is versioned as a whole. Normally, only one user owns a model and its open editions, but different users can have different editions open.

Each edition of a model or element points to the released edition of each element it contains. An element is automatically released to its parent when it is created, loaded, or changed. You can also use the UML Designer Editions Browser to manually release a different edition.

### Browsing and loading

To load a model that is not currently in your image, select **Available** from the **Source** menu of the Relationships Browser (or from the pop-up menu in the list of models). This option opens a standard ENVY applications browser from which you can choose the application containing the model you want.

To load a different edition of a model that you already have loaded, do one of the following in the Relationships Browser:

- To use the standard ENVY browsers, select **Application→Browse Editions** from the **Source** menu, or from the pop-up menu in the list of models. This opens a browser from which you can select the edition you want to load. This method is usually fastest, because it loads the entire application at once rather than one element at a time.
- Select **Editions** from the **Source** menu, or from the pop-up menu in the list of models. This option opens a UML Designer Editions Browser from which you can select the edition you want to load. This method shows more detail than the standard ENVY browser, so it can be useful if you're not sure which edition you want.

**Note:** Any time you load a model using standard ENVY browsers (rather than the UML Designer Editions Browser), you must manually refresh any open UML Designer browsers in order to see the newly loaded model. There are two ways to do this:

- To refresh a single browser, select **Refresh Browser** from the **File** menu of the browser you want to refresh.
- To close all open browsers, select **Reset All Browsers** from the **UML Designer** menu of the Transcript window. You can then reopen the browsers you need.

## Loading elements

Loading an edition of an element also loads the released editions of its children. You can also manually load other editions of the elements. You can do this in either of two ways:

- In the Relationships Browser, select a loaded element and then select **Editions** from the pop-up menu. This opens a UML Designer Editions Browser from which you can select the edition you want to load.
- In the Relationships Browser, select a relationship (such as **Requirements**) and then select **Available** from the pop-up menu. This opens a browser from which you can select the element and edition you want to load.

**Note:** You should always use the UML Designer Editions Browser or Hierarchical Change Browser (rather than the standard ENVY browsers) to browse and load editions of model elements. These browsers are designed to display model elements. See “Using the UML Designer browsers” on page 57 for more information.

## Crash recovery

To recover from a crash, restart your image. If you have made changes to your model more recently than the last time you saved your image, load the latest edition of the model application. Loading the model automatically loads the released child elements as well.

## Composite objects

Certain types of elements, called **composite objects**, are stored within their parents, limiting the ways in which they can be browsed and loaded. The following elements, all part of message specifications, are composite objects:

- Parameter
- Parameter text
- Return value
- Return value text
- Attribute

Because these elements are embedded within their parents, you cannot browse editions for them individually. To see the changes of a composite object, browse its parent object (a message specification). When you browse a message specification, its composite relationships are expanded and displayed inline so you can see the contents of these objects.

You can use the Hierarchical Change Browser to selectively load different editions of composite objects individually. See “Hierarchical Change Browser” on page 58 for more information.

## Team development

Ownership of UML Designer components is at the model level. Therefore, it is not currently possible to share ownership of a model (that is, all elements of a model have the same owner). In order for multiple developers to work on the same project, you must use one or both of the following strategies:

- Separate the project into multiple models, using prerequisites to provide appropriate visibility among them.
- Have each developer work on a separate edition of the model, reconciling the differences between the editions later. (The Hierarchical Change browser can help you do this.)

---

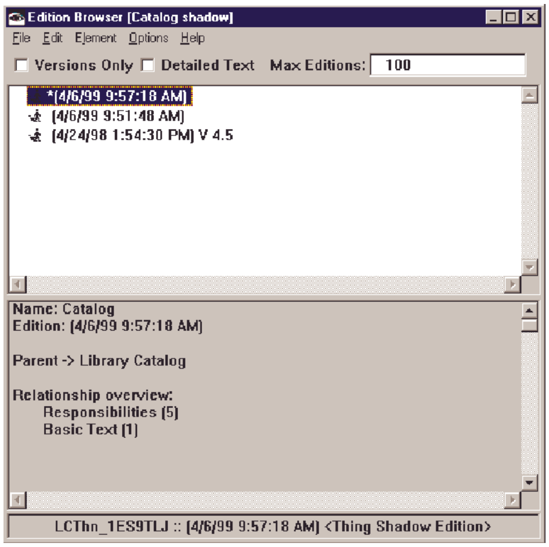
# Using the UML Designer browsers

UML Designer provides two new browsers intended to help you manage version control of your models: a modified Edition Browser, and the Hierarchical Change Browser.

## Edition Browser

The UML Designer Edition Browser enables you to see the available editions of a selected model element, along with summary information about each. You can use this browser to load a different edition of a model element.

To open the Edition Browser, select a model element in the Relationships Browser and then select **Editions** from the pop-up menu.



☐ Versions Only ☐ Detailed Text Max Editions: 100

```
*(4/6/99 9:57:18 AM)
(4/6/99 9:51:48 AM)
(4/24/98 1:54:30 PM) V 4.5
```

```
Name: Catalog
Edition: (4/6/99 9:57:18 AM)

Parent -> Library Catalog

Relationship overview:
    Responsibilities (5)
    Basic Text (1)
```

```
LCThn_1ES9TLJ :: (4/6/99 9:57:18 AM) <Thing Shadow Edition>
```

The top pane of the Editions Browser shows a list of the available editions and versions of the selected model element. An asterisk (*) indicates the currently loaded edition.

The bottom pane of the Editions Browser shows information about the edition currently selected in the top pane. This information includes:
- Name
- Parent
- Links to other elements
- Contents

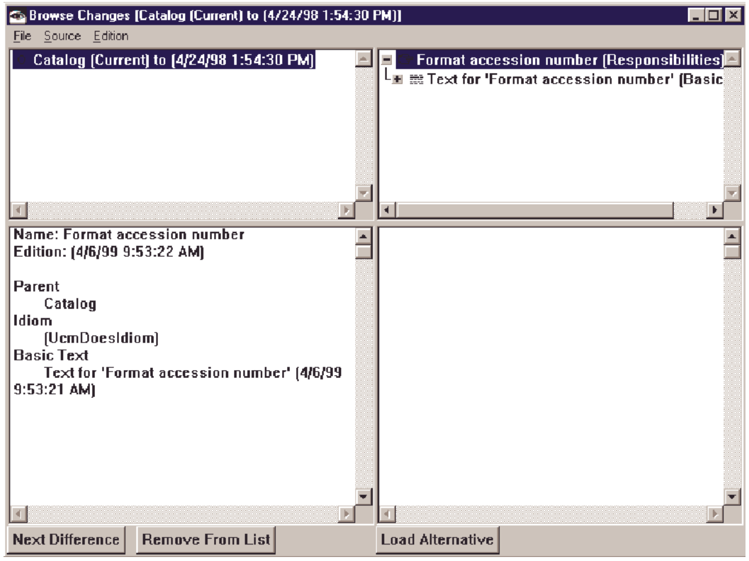
### Browser options

You can use several browser options to control the information that appears in the Edition Browser.
- Select **Versions Only** to include only named versions (no editions) in the browser.
- Use **Detailed Text** to control how much information the browser shows about the element's links to other elements:
  - If **Detailed Text** is selected, the browser lists each link separately, including the type of relationship and the name of the linked element.
  - If **Detailed Text** is not selected, the browser lists only the number of links of each type.
- Use **Max Editions** to control how many editions the browser should show (starting with the most recent and working backward). For an element with a large number of older editions, this option can help to speed up library access, particularly if **Detailed Text** is selected.

## Hierarchical Change Browser

The Hierarchical Change Browser is a specialized UML Designer browser you can use to compare and reconcile the differences between editions of an element. This browser is particularly useful in a situation where different developers have been working on different editions of the same model, and you now want to merge the editions together into a single consistent edition.

To open the Hierarchical Change Browser, select an element in the Relationships Browser and then do one of the following:
- Select **Browse Changes→Previous Edition** to compare the currently loaded edition with the previous edition.
- Select **Browse Changes→Another Edition** to compare the currently loaded edition with any other edition. You will be prompted to select the edition to compare to.



Catalog (Current) to (4/24/98 1:54:30 PM)

Format accession number (Responsibilities)
Text for 'Format accession number' (Basic

Name: Format accession number
Edition: (4/6/99 9:53:22 AM)

Parent
Catalog
Idiom
(UcmDoesIdiom)
Basic Text
Text for 'Format accession number' (4/6/99 9:53:21 AM)

The top left pane of the Hierarchical Change Browser shows which two editions are being compared.

The top right pane of the browser lists, in a hierarchical tree view, all of the elements that differ between the two editions being compared. The hierarchical view arranges the elements according to their parent-child relationships (for example, message specifications appear below the protocol they belong to).

Any element that changed between the two editions being compared appears in the list. This change can either be a change in the element’s own contents, or a change in the contents of a subelement it contains. For example, if the text of a use case has changed, the use case appears in the list with the text listed below it.

## Browsing and reconciling differences

When you select an element from the list, the bottom two panes of the browser compare the contents of the two editions. The bottom left pane lists the contents of the current edition; the bottom right pane lists the contents of the alternate edition. These contents include the element’s links to other elements (including contained elements); for a text element, the browser also shows the actual text so you can compare the changes. Use the scroll bars to scroll through the lists.

To find differences between the two editions, click on the **Next Difference** button. Each time you click on **Next Difference**, the browser highlights the next location where the contents of the two panes differ. This can be an element that appears in one edition and not in the other, or an element that has changed between the two editions.

In order to reconcile two editions, for each difference you must decide which of the editions you prefer. If you want to stay with the currently loaded edition, you do not need to do anything; however, you might want to click on **Remove From List** to hide the highlighted entries (this does not change anything in the actual elements, but it helps to reduce clutter in the browser).

If you want to switch to the alternate edition, click on **Load Alternative**. This changes the currently loaded edition so it matches the alternate edition (with respect to the selected difference).

# Chapter 7. Importing and exporting models

## Object Extender import/export

The VisualAge Object Extender feature provides tools you can use to model, map, and generate classes for persistent application objects. UML Designer can import model information from, or export it to, Object Extender models. (Similarly, Object Extender can import from or export to UML Designer models; see the VisualAge Smalltalk ObjectExtender User's Guide and Reference for more information.)

When you import model information from Object Extender, UML Designer converts each selected model class into a protocol and a conforming class design, using the same name as the Object Extender model class. If there is an associated Smalltalk class generated from Object Extender, that class will also be linked as the real implementing class of the class design.

For each attribute in the model class, UML Designer creates an attribute in the generated protocol. If the attribute is marked as a required value in the model class, UML Designer uses the **Key Attribute** idiom for the generated attribute.

In the generated class designs and protocols, UML Designer maintains any inheritance relationships among the Object Extender model classes. If the model class has no parent model class, the generated UML Designer class design and protocol are children of the Object Extender default persistent class root (configured in the Object Extender Model Browser's generation options). The Object Extender Base model, which contains class designs and protocols for the most important Object Extender root classes, is automatically set as a prerequisite for any model containing elements imported from Object Extender.

UML Designer also preserves any associations among the imported model classes, provided both source and target of each association are being imported.

### Importing model elements from Object Extender

To import model elements from Object Extender into a UML Designer model, follow these steps:

1. In the Relationships Browser, select the model you want to contain the imported elements. (Create a new model if necessary.)
2. Select **Transforms→Import From Object Extender** from the pop-up menu.

   A prompter appears listing the available Object Extender models.
3. Select the Object Extender model that contains the model classes you want to import. Then select **OK**.
4. When prompted, select the Object Extender model classes you want to import. (Remember that if you want to preserve associations between classes, you must import both the source and destination classes). Then select **OK**.

After the import operation completes, the target UML Designer model contains a class design and a protocol for each selected Object Extender model class. You can now use these elements in your model as with any other UML Designer elements.

### Exporting model elements to Object Extender

To export class designs to Object Extender, follow these steps:

1. In the Relationships Browser, select the model containing the class designs you want to export.
2. Select **Transforms→Export To Object Extender** from the pop-up menu.

   A prompter appears listing the available Object Extender models.
3. If you want the exported model classes to be placed in an existing model, select the target model and then select **OK**.

   If you want to create a new Object Extender model for the exported model classes, select **<New>** and then select **OK**. You can then specify the name of the new model.
4. When prompted, select the UML Designer class designs you want to export. (Remember that if you want to preserve associations between classes, you must export both the source and destination classes.) Then select **OK**.

After the export operation completes, the target Object Extender model contains a model class for each exported UML Designer class design. See the VisualAge Smalltalk ObjectExtender User's Guide and Reference for more information about model classes imported from UML Designer.

## XMI import/export

XMI (XML Metadata Interchange) is an open standard for exchanging object programming and design information between application development tools and repositories. Based on XML (Extensible Markup Language), XMI provides an industry-standard format for sharing design information based on UML modeling definitions. UML Designer can import model information from, or export it to, XMI file streams.

The UML Designer XMI support uses XMI Version 1.0, which is based on the UML 1.1 metamodel. To view the UML 1.1 XMI Document Type Definition, inspect the following Smalltalk expression in a workspace:`XmiUmldImport dtdSource`.

Not all XMI modeling elements are supported by UML Designer. The following table shows which UML Designer elements can be imported from, or exported to, XMI streams; it also shows how the UML Designer elements are mapped to corresponding XMI elements. (A complete XMI stream will include additional tagging; these are only the primary elements.)

| **UML Designer model element** | **XMI element tag (primary only)** |
|---|---|
| Model | `Model_Management.Model` |
| Group | `Model_Management.Package` |
| Class Design | `Foundation.Core.Class` |
| Protocol | `Foundation.Core.Interface`<br>`Foundation.Core.DataType`<br>`Foundation.Data_Types.Enumeration`<br>`Foundation.Data_Types.Primitive` |
| Message | `Foundation.Core.Operation` |
| Parameter | `Foundation.Core.Parameter` |
| Return Value | `Foundation.Core.Parameter` |
| Attribute | `Foundation.Core.Attribute` |
| Association | `Foundation.Core.Association` |

| **UML Designer model element** | **XMI element tag (primary only)** |
|---|---|
| Inheritance | `Foundation.Core.Generalization` |
| Group Membership | `Model_Management.ElementReference` |
| Actor | `Behavioral_Elements.Use_Cases.Actor` |
| Use Case | `Behavioral_Elements.Use_Cases.UseCase` |
| Stereotype | `Foundation.Extension_Mechanisms.Stereotype` |
| Related Text | `Foundation.Extension_Mechanisms.TaggedValue`<br>`(with tag = 'documentation')` |

## Exporting from UML Designer to XMI

To export UML Designer model elements to an XMI file, follow these steps:

1. In any UML Designer browser, select the source model.
2. Select **Import/Export Tools→XMI export** from the pop-up menu. (This option is also available from the **Source** menu of the Relationships Browser, or the **Element** menu of the Hierarchy Browser.)
3. When prompted, specify the location and file name for the exported XMI file.
4. Select **Save** to save the XMI file.

## Importing from XMI to UML Designer

To import elements from an XMI file into a UML Designer model, follow these steps:

1. Select **Import/Export Tools→XMI import (new model)** from the pop-up menu. (This option is also available from the **Source** menu of the Relationships Browser, or the **Element** menu of the Hierarchy Browser.)
2. When prompted, select the file containing the XMI stream you want to import.
3. When prompted, specify the name, namespace prefix, and application name of the new UML Designer model that will contain the imported elements.
4. Select **OK** to begin the import operation. Importing might take several minutes for a large model.

**Note:** An XMI stream can contain multiple model elements of the same type and name, but in different namespaces. However, UML Designer has only one namespace for each model. To resolve name conflicts, UML Designer appends a unique suffix to any conflicting element name. The suffix is automatically generated based on the model's namespace prefix and a number.

# Part 3. Building models with UML Designer

# Chapter 8. Capturing requirements

The first step in a “forward” development process is to capture the requirements for the system being developed. (For more information about requirements, see “Requirement” on page 8. For our example, we want to build a library catalog system. Our users describe the system they want as follows:

*The system must maintain a catalog of the available books in the library. Books must be separately indexed by title, author, and Dewey Decimal System number. The system must support registering newly acquired books and catalog queries to find existing books.*

Based on this description (and perhaps conversations with users), we can then begin to derive a list of discrete requirements for the system, and we can use UML Designer to capture them.

## Starting a new model

To start working in UML Designer, select **Relationships Browser** from the **UML Designer** menu in the Transcript window. The Relationships Browser opens.

The Relationships Browser shows the available models, the design elements in each model, and the relationships that exist among them. From here, you can create new design elements, browse and modify their relationships, and launch the other UML Designer browsers and editors. (For more information about the Relationships Browser, see “The Relationships Browser” on page 27.)

When you first install UML Designer, three models are automatically included:

- *Kernel Model* includes design elements for some implementation-independent common data types.
- *Kernel Model—Java Examples* includes design elements for Java data types.
- *Kernel Model—Smalltalk Examples* includes design elements for Smalltalk data types.

Since you’re likely to need some of the basic data types in any system you design, UML Designer, by default, makes the *Kernel Model* and *Kernel Model—Smalltalk Examples* prerequisites for any models you create. (If you want to use Java data types, you will need to manually add *Kernel Model—Java Examples* as a prerequisite.) This means that the basic Smalltalk classes in the *Kernel* models are automatically visible from any other model.

To begin modeling a new system, you must first create a new model in the Relationships Browser:

1. Select **Create Model** from the **Source** menu, or from the pop-up menu in the leftmost pane in the Relationships Browser.
2. When prompted, specify the following information about the new model:
    - The specification name of the model. This can be any name you want to use to identify the model. Our example will be a system to catalog library books, so type *Library Catalog*.
    - The namespace prefix for the model. This is an arbitrary string that will be used as a prefix to uniquely identify the model and its elements in the

Smalltalk repository. You can accept the default value for this prefix, or you can add your initials or some other unique identifier if you share the same repository with other users.
    - The Smalltalk application name for the model. This is the name that will be used for the application containing the model elements. You will not normally need to access this application directly, so in most cases you can accept the default application name, which is the model name appended to the namespace prefix.
3. Select OK. The new model appears in the list in the Relationships Browser.

## Adding Requirement elements

By looking at the above description from our users, we might derive the following individual requirements for the system:

- Keep a catalog of available books
- Index books by title, author, and Dewey number
- Support catalog queries

To add these requirements to the model, follow these steps:

1. Select *Library Catalog* from the list of models in the leftmost pane of the Relationships Browser. The middle pane shows a list of possible relationships to model elements the model might contain.
2. In the middle pane, select **Requirements**. The rightmost pane shows a list of requirements already defined for the model; since we haven't created any yet, the list is empty.
3. Select **New** from the **Relationships** menu, or from the pop-up menu for the selected relationship.
4. When prompted for the requirement name, type *Keep a catalog of available books.*
5. Select **OK**. The new requirement appears in the list.
6. Repeat the same process to add the other two requirements listed above.

After you have added all three requirements to the list, you can provide additional descriptive text for each requirement. To do this, select the requirement you want to describe, and then use the bottom text pane of the Relationships Browser to type the requirement text. This text pane can contain any kind of description you want to capture for the requirement; it can also contain hypertext links to other model elements. Our example's requirements are fairly straightforward and probably do not need much more explanation at this stage. Instead, let's move on to use cases.

# Chapter 9. Writing and analyzing use cases

Once you have captured your initial requirements, you can write use cases for the system, using Use Case elements to record them. For more information about use cases, see “Use case” on page 8.

## Adding Use Case elements

Based on the requirements for the library catalog system, we can identify three use cases, each describing a particular user task:

- Adding a new book to the catalog
- Removing a book from the catalog
- Querying the catalog to find a book

To add these use cases to the *Library Catalog* model, follow these steps:

1. Select **Use Cases** in the middle pane of the Relationships Browser.
2. Select **New** from the **Relationships** menu, or from the pop-up menu in the middle pane.
3. Type *Adding a new book to the catalog.*
4. Select **OK**. The new use case appears in the list in the rightmost pane.
5. Repeat the same procedure to add the other two use cases listed above.

A use case requires a carefully written description, rather than just a brief title. We must now use the hypertext pane (at the bottom of the Relationships Browser) to add detail to each use case. To edit the text of a use case, select the use case from the list and type the text in the hypertext pane. You can use the hypertext pane’s pop-up menu to save the text, just as in any Smalltalk browser.

Add the following text to the first use case, *Adding a new book to the catalog*:

*A new book arrives at the library, and a librarian creates a new catalog entry with the book’s author, title, and Dewey number. The catalog entry is then assigned a serial number and registered in the catalog. The catalog uses the book’s Dewey number to print a bookplate and spine label for the book.*

When you have finished, save the text by selecting **Save** from the pop-up menu or pressing Ctrl+S.

Optionally, you can add descriptive text for the other use cases you created. These aren’t necessary for the rest of the example, because we will focus on the *Adding a new book* use case.

## Making links between elements

Use cases can be related to requirements. They elaborate upon what the requirements entail, and in some cases they can uncover additional requirements. Each use case *satisfies* one or more requirements; if a use case does not satisfy any requirement, either a new requirement should be added, or the use case is outside the scope of the system and should be dropped.

In UML Designer, you can create links between model elements in order to show this “satisfies” relationship, as well as other relationships. Such links contribute to traceability. **Traceability** is the ability to move between elements at different levels of abstraction. For example, you can track which use cases satisfy each requirement, or which requirements are satisfied by each use case. This traceability helps to clarify *why* the elements in the model are what they are and how they became that way.

## Adding a "satisfies" link

To add a “satisfies” link between a use case and a requirement, follow these steps:

1. Select **Use Cases** in the middle pane of the Relationships Browser.
2. Select *Adding a new book to the catalog* from the list.
3. Select **Link→Satisfied Requirements** from the pop-up menu of the use case. A window appears listing the available requirements.
4. Select *Keep a catalog of available books* from the list.
5. Select **>>** to add the selected requirement to the list of linked requirements.
6. Select **OK**.

To see the effect of this link, double-click on *Adding a new book to the catalog* in order to open a browser on that requirement. In the middle pane of the browser, select the **Satisfied Requirements** relationship. You should see a list including *Keep a catalog of available books*.

Optionally, you can add “satisfies” links from the other use cases you created.

# Analyzing a use case

Once you have a use case description, you can analyze the use case to begin identifying some of the elements that will play a part in the system. One way to start is to look for all of the nouns in the use case description to see which ones seem to refer to important elements of the system. Each time you find an important idea in the use case, you can classify it as one of the following design elements (see “Requirements model elements” on page 8 for more information on these elements):

- An **actor** outside the system.
- A **thing** within the system (also called a domain object).
- A **concept** that needs to be recorded, but might not be a thing.
- Another related **use case**. There are two kinds of relationships that use cases can have with one another:
  - A use case can **use** another use case, represented by a simple “associated with” relationship.
  - A use case can **extend** another use case, represented by an “extends” relationship.

Now, let’s take a look at the text of the *Adding a book* use case to see what other design elements it refers to.

# Identifying actors

The action of the *Adding a book* use case is initiated by a librarian, a human user. This is a person who actually enters the information for a new book and initiates the cataloging operation. Because the librarian exists outside the system and provides the stimulus setting the use case in motion, the librarian is an actor. We can now create an Actor element in the *Library Catalog* model to represent the librarian.

## Adding an Actor element

To create a new actor, follow these steps:

1. Select **Actors** in the middle pane of the Relationships Browser.
2. Select **New** from the **Relationships** menu, or from the pop-up menu of the relationship.
3. When prompted for the name of the actor, type *Librarian*.
4. Select **OK**. The new actor appears in the list in the rightmost pane.

# Adding a hypertext link

Since we have created an Actor element to represent the librarian, we can now indicate the relationship between this actor and the use case in which it participates. One way to do this is with a hypertext link in the use case description.

To insert a hypertext link for the *Librarian* actor, follow these steps:

1. Select **Use Cases** in the middle pane of the Relationships Browser.
2. Select *Adding a new book to the catalog* from the list of use cases.
3. In the text pane, select the word *librarian* in the use case description. (You can do this with the keyboard, by swiping the mouse, or by double-clicking on the word.)
4. Select **Insert Link→Reference** from the pop-up menu of the text pane. A window appears showing the different kinds of objects you can link to.
5. In this case, the object we want to link to is an actor, so select **Actor** from the list. A list of available actors appears; at this point, we have only created one actor, *Librarian*, so no other choices appear.
6. Select *Librarian* and then select **OK**.
7. A window now appears verifying the label for this actor you want to appear in the text of the use case (the label of a link can be different from the name of the destination element). If you wanted to change the wording of the reference to the librarian in the use case description, you could change it here; for now, select **OK** to accept the default.

An actor icon appears beside the word *librarian* to indicate the existence of a link to an actor in the model. You can follow this hypertext link by double-clicking on the linked word. This opens a browser on the *Librarian* actor.

## Identifying things and responsibilities

The object model of an OO system usually has a strong correspondence to the actual real-world entities of the domain: for each domain entity (whether physical or intangible), there is generally one or more corresponding implementation objects. This “natural” model helps to make the system understandable, and it also means that changes to the problem domain map well to changes in the software.

To choose the correct objects for the system, you need to identify the required behavior of the objects. With UML Designer, you do this by creating **things** (domain objects) and assigning **responsibilities** to them.

In the *Adding a new book* use case, there are several terms that refer to entities that exist within the system (unlike the librarian, who is outside the system):

- library
- catalog entry
- catalog

These are candidates to become objects in the implemented system. We can now create Thing elements in the *Library Catalog* model to represent each of these.

### Adding (and linking to) a new Thing element

In the actor example above, we manually created a new Actor element for *Librarian* and then separately created a hypertext link to it. This time, let’s let the system do some of the work for us; we can simultaneously create a new element and a hypertext link to it. To do this, follow these steps:

1. Select **Use Cases** in the middle pane of the Relationships Browser.
2. Select *Adding a new book to the catalog* from the list of use cases.
3. In the use case text, select the word *library*.
4. Select **Insert Link→Reference** from the pop-up menu in the text pane. A window appears listing the different kinds of objects you can link to.
5. We want to link to a thing, so select **Thing** from the list. You can accept the default name *Library,* which is derived from the selected text in the use case.
6. Select **New**, indicating that we want to create a new element.
7. Select **OK** to confirm the text label for the link.
8. Repeat the same process for *catalog entry* and *catalog*.

For each hypertext link, a Thing icon ◦ appears along the linked text in the use case description indicating the link to a Thing element. If you select **Things** in the middle pane of the Relationships Browser, you should see *Catalog, Catalog entry,* and *Library* listed.

### Identifying responsibilities

The next step is to identify the responsibilities of the domain objects. At this stage, it can also be helpful to use “computer-free” techniques such as CRC cards to identify responsibilities, entering the results as new responsibility elements. (CRC cards—“Class-Responsibility-Collaboration” cards—are a technique developed by Ward Cunningham and Kent Beck using index cards to represent candidate objects in a system and elucidate their collaborations.)

To find responsibilities, go through the text of a use case and identify what the things and actors must do. You can then assign each responsibility to the appropriate thing or actor. In addition, each responsibility can be linked to **collaborating participants**, the other things or actors that participate in the responsibility.

## Adding responsibility elements

Consider *Catalog*, which we have identified as a thing in the library catalog example. Based on our use cases, *Catalog* (which we have identified as a thing) might have the following responsibilities:

- Add catalog entry
- Assign accession number to entry
- Find entry by search criteria
- Remove catalog entry

To add these responsibilities of *Catalog* to the model, follow these steps:

1. Select **Things** in the middle pane of the Relationships Browser.
2. Select *Catalog* from the list of things.
3. Select **New→Responsibility** from the pop-up menu of *Catalog*. A window appears prompting you for the details of the new responsibility.
4. Type *Add catalog entry* and select **OK**.
5. For **Idiom**, accept the default (**Action**). This responsibility describes something *Catalog* does.

   Don't worry about participants for now; we will deal with those shortly.
6. Repeat the same procedure for the other responsibilities listed above.

Similarly, you can create responsibilities for actors by following the same steps. Select the actor *Librarian* and add the following responsibilities:

- Input new catalog entry
- Classify book and assign Dewey number
- Query catalog using search criteria
- Remove existing catalog entry

**Note:** Because Actor elements are not transformed into protocols, actor responsibilities do not have idioms.

### Using another idiom

All of the responsibilities we have added for *Catalog* have been simple "action" responsibilities. However, consider another Thing, *Catalog Entry*; it does not have to do anything except keep track of its own attributes (author, title, and accession number). We can define these as responsibilities using the **Value** idiom to give us a head start on implementation later.

To add these responsibilities, follow these steps:

1. In the Relationships Browser, select *Catalog Entry* from the list of things.
2. Select **New→Responsibility** from the pop-up menu. A window appears prompting you for the details of the new responsibility.
3. For **Idiom**, select **Value**. This responsibility describes something *Catalog Entry* keeps.
4. In the **Responsibility Name**field, type *Author*. (We phrase this responsibility as a noun because it describes a value rather than an action.)
5. For **Max Cardinality**, select **1**.

6. Optionally, we could specify a participant defining the type of the value being kept. At this point, though, we aren't really interested in how the author will be represented, so accept the default and select **OK**.
7. Repeat the same process for two other responsibilities, *Title* and *Accession Number*.

Because we used the **Value** idiom for these responsibilities, UML Designer can generate appropriate sets of messages when we transform *Catalog Entry* into a protocol. We'll see how this works in "Chapter 11. Protocols" on page 81.

## Linking to participants

In addition to indicating the responsibilities of a thing or actor, you can also indicate which other things or actors are collaborating participants in each responsibility. To do this, you create a link from a defined responsibility to a thing or actor that participates. (You can also specify participants along with idioms as you create a new responsibility.)

In our example, three of the responsibilities of *Catalog* also involve catalog entries:

- Add catalog entry
- Assign accession number to entry
- Remove catalog entry

Each of these is a responsibility of *Catalog*, but each also involves another thing, *Catalog entry*. Therefore, you can create a link from each of these responsibilities to the *Catalog entry* element to indicate that it participates. To do so, follow these steps:

1. Select **Things** in the middle pane of the Relationships Browser.
2. Double-click on *Catalog*. A browser opens on the *Catalog* element.
3. Select **Responsibilities** in the middle pane of the browser.
4. Select *Add catalog entry* from the list of responsibilities.
5. Select **Link→Participants** from the pop-up menu of the responsibility. A window appears prompting you for the link destination.
6. A participant can be either an actor or a thing. In this case, *Catalog entry* is a thing, so select **Thing** from the **Available Types** list.
7. Select *Catalog entry*.
8. Select >> to add *Catalog entry* to the list of linked participants.
9. Select **OK**.
10. Repeat the same process with *Assign accession number to entry* and *Remove catalog entry*, adding *Catalog entry* as a participant in each responsibility.
11. Close the browser.

# Identifying concepts

In addition to actors and things, there might be other important ideas described in the use case that need to be documented. Going through the use case description, we can identify terms that need additional explanation. As you do this, it's important to decide on standard terminology, in order to be sure everyone understands the requirements the same way. Careful definition of terms can help to overcome the imprecision and ambiguity inherent in natural language.

As a general rule, each term should refer to only one concept, and each concept should be described by only one term. In addition, any domain-specific terminology, or domain-specific usage of common terms, should be fully defined. We can use concept elements to capture this information.

### Adding a Concept element

In the *Adding a new book* use case, several phrases appear to be important concepts that should be documented. For example, *Dewey number* is domain-specific terminology that should be defined for the benefit of non-librarians. A description can also help us understand how we need to represent it in the system.

The Dewey number is an attribute of a catalog entry, so it's probably not a thing in its own right; therefore, we'll use a Concept element to define it. To do this, follow these steps:

1. Select **Use Cases** in the middle pane of the Relationships Browser.
2. Select *Adding a new book* in the list of use cases.
3. Select the phrase *Dewey number* in the hypertext pane.
4. Create a hypertext reference link from the phrase to a new Concept element. (Use the same process you used to create the links above, but select **Concept** as the kind of object to link to.)
5. In the text pane for the new *Dewey number* concept element, type a description of a Dewey number:

   *A Dewey number is a numerical classification identifying a book's subject matter.*

Another phrase that looks like a good candidate for a Concept element is *serial number*. Like the Dewey number, the serial number is an attribute of a catalog entry, so it is probably not a thing; however, it's an important concept that probably should be recorded. But suppose that, in trying to define *serial number*, we find that our use case does not reflect correct terminology?

---

## Revising description text

As you find out more about the problem domain, you will probably find that some parts of your use cases or requirements are vague or inaccurate or do not reflect the correct terminology. UML Designer accommodates this situation; you can rename model elements and revise their contents while still preserving any links between them. Furthermore, because all models are stored in the Smalltalk repository, you can always return to a previous version.

For example, we might find out that librarians assign a new book an *accession number* rather than a *serial number.* We now need to update our *Adding a new book* use case, replacing the phrase "serial number" with "accession number". The new use case description should read:

*A new book arrives at the library, and a librarian creates a new catalog entry with the book's author, title, and Dewey number. The catalog entry is then assigned an accession number and registered in the catalog. The catalog uses the book's Dewey number to print a bookplate and spine label for the book.*

Edit the use case text to use the correct terminology and then save the new text. After you save it, select **Text Editions** from the pop-up menu in the hypertext pane. A browser opens showing all of the saved editions of the text of this use case; if you change your mind, you can always reload an older edition.

# Chapter 10. Use case diagrams

Once you have developed use cases and identified actors, you can create use case diagrams to graphically depict your actors and use cases and the relationships between them. The UML Designer Use Case Diagrammer helps you build UML-compliant use case diagrams based upon the underlying model elements you have defined.

## Creating a use case diagram

To create a new use case diagram for the library catalog system, follow these steps:

1. Select **Use Case Diagrams** in the middle pane of the Relationships Browser.
2. Select **New** from the pop-up menu of the relationship.
3. When prompted for the name of the diagram, type *Library catalog* and then select **OK**. The new diagram appears in the list in the rightmost pane.
4. Double-click on *Library catalog* in the list of use case diagrams to open the Use Case Diagrammer.

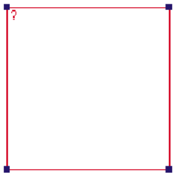
## Adding a system figure

The first step in building a use case diagram is to add a figure representing the boundaries of the library catalog system. This boundary separates the use cases (which are essentially part of the system) from the actors (which are outside the system).

To add a new figure to the diagram, you first create the figure and then attach it to an underlying model element. A system figure must be attached to a group element, which we have not created; in this case, therefore, we will be adding a node figure and creating an underlying model element simultaneously.

To add a system figure, follow these steps:

1. Select ☐ **System** from the tool bar.
2. Use mouse button 1 to place the icon in an empty area of the drawing surface. When first added to the drawing surface, the icon is labeled with a question mark. This indicates that it is not yet associated with an underlying model element. All elements in UML Designer diagrams must be associated with underlying models.



3. Click mouse button 2 on the icon to display its pop-up menu.
4. Select **Attach** from the pop-up menu. A window appears showing the available groups to attach to the system figure.

   A **group** is an organizational element you can use to collect related model elements together. In a use case diagram, a group is the underlying model element for the system figure, which is essentially a grouping of related use cases.

5. Because no groups yet exist in the library catalog model, type *Library Catalog* in the **Specify New Item** field and then select **New**.

The system figure is now attached to the new group, and the label *Library Catalog* appears in the diagram.

Library Catalog

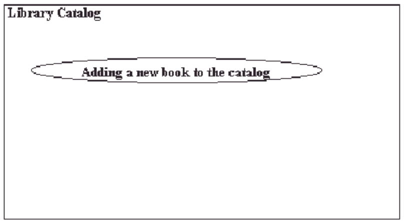
## Adding a use case figure

The next step is to add a use case figure to the diagram. Because we have already created several use cases, this time we only need to add a figure, which we can then attach to an existing model element.

1. Select 0 **Use Case** from the tool bar.
2. Click mouse button 1 to place the figure within the system figure.

   Because it is not yet associated with an actual use case, the use case figure is labeled with a question mark.
3. Select **Attach** from the pop-up menu of the use case figure. A window appears listing the available use cases in the model.
4. Select *Adding a new book to the catalog* from the list.
5. Select **OK**.

The use case figure is now attached to the underlying use case and is labeled appropriately.



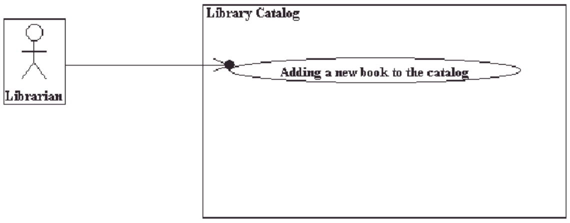
### Adding an actor figure

The next step is to show the link between the use case and an actor. For the use case figure, we manually added the figure and then attached it to the underlying element. This time, we'll have UML Designer do some of the work for us:

1. Select the *Adding a new book* use case figure.
2. Select **Hide/Show Relationships** from the pop-up menu of the use case figure. A window appears listing links for existing relationships involving the use case, followed by any identified candidate links (candidate links are indicated by a preceding **+**).

   In this case, UML Designer has identified a candidate link between the *Adding a book* use case and the *Librarian* actor. This link is suggested because of the reference to *Librarian* (and the accompanying hypertext link) in the description of the use case.

3. Select *+(self >- (Associated With) -> Librarian <Actor>)* from the **Hidden Relationships** list.
4. Select **>>** to add the link to the **Shown Relationships** list. This creates an explicit “associated with” link and indicates that you want the relationship to be represented in the diagram.
5. Select **OK**.
6. A window appears prompting you to confirm the direction of the association. Select **Yes** to confirm the default (from the actor to the use case).
7. An Actor figure attached to *Librarian* appears on the drawing surface. You can now move this figure in order to rearrange the diagram any way you like.



## Other ways to create relationships

There are two other ways you can create an explicit link between a use case and an actor, or between two use cases:

- You can create an explicit “associated with” relationship in the Relationships Browser. These links are independent of any hypertext links that might exist between the elements and must be created as a separate step.

  When you create an explicit relationship in the Relationships Browser, it is listed as an existing relationship in the Use Case Diagrammer. However, it will still not appear in the diagram unless you make it visible using **Hide/Show Relationships** on the pop-up menu.

- You can add figures for the elements to the diagram, and then use **Association** or **Extension** (depending upon the kind of link) to connect them. Creating a visible association in a use case diagram automatically creates an association between the underlying model elements.

---

# Deleting figures

Because a diagram and its underlying model exist independently, it is possible to delete a figure from the diagram without removing the attached element from the model.

For example, we might decide we do not want the diagram to show the link between the Librarian actor and the *Adding a new book* use case. To remove the link figure without removing the actual link between the elements, follow these steps:

1. Select the link figure connecting the Librarian figure to the *Adding a new book* figure.
2. Select **Delete Figure** from the link figure’s pop-up menu.

To confirm that the link still exists (even though it is no longer shown), select **Hide/Show Relationships** from the Librarian figure's pop-up menu. You should still see the link listed as an existing relationship. (To restore the figure to the diagram, use the **>>** button to add it back to the list of shown relationships.)

# Chapter 11. Protocols

By now we have developed a fairly good idea of the requirements of our system, at least good enough for a first pass. We understand what use cases it must support, we've identified the actors and things, and we know what their responsibilities are. The next step is to begin analysis of the system, in order to start figuring out how we will satisfy these requirements.

At this point, your first instinct might be to begin writing code, implementing things as objects and responsibilities as methods. A better approach, however, is to more closely examine the responsibilities of the things in the system in order to understand how they work together. We can do this through the use of protocols. (In fact, you can often do both: go ahead and sketch out some classes, but then come back to analysis and use protocols to make sure the design is watertight.)

Rigorous use of protocols guarantees substitutability: any class conforming to a given protocol (or to one refining that protocol) can be substituted for any other class conforming to that same protocol. If a class conforms to the expected protocol, type mismatches cannot happen. In Smalltalk terms, this means never getting a *doesNotUnderstand* message.

## Protocols and things

A reasonable initial assumption in modeling is that there will be one protocol (and therefore one type) for each defined thing. In this simple mapping, the responsibilities of the thing become message specifications in the protocol, defined with more detail and more precision. This one-to-one mapping does not always hold true, but it's a good place to start.

Protocols provide an intermediate step in the complex transformation of requirements into implementation code. Protocols themselves are not code, but are abstract interface specifications that can be transformed into an OO implementation in a fairly straightforward manner.

## Generating a protocol

Once you have defined a thing and its responsibilities, you can use them to automatically generate a protocol. When generating a protocol, UML Designer assumes the default mapping of one message specification for each responsibility, and it uses the names of the responsibilities to generate a default message name. You can change this afterward if the generated message name is not satisfactory. You can also make other manual changes, like adding message specifications, splitting a protocol into two, or merging two protocols into one.

We've defined several responsibilities for the thing *Catalog*, so we can now generate a protocol that **implements** the thing. To generate a protocol for *Catalog,* follow these steps:

1. Select **Things** in the middle pane of the Relationships Browser.
2. Select *Catalog* from the list of things.
3. Select **Transforms→Generate Protocol** from the pop-up menu of *Catalog*.
4. In the window that appears, select *Catalog* from the **Protocol Names** list.

5. You can specify several options that affect the generated protocol:
   - If you select **Delete Existing Messages**, the messages in an existing protocol will be deleted before the new messages are generated. (This option is available only if you are regenerating an existing protocol.)
   - If you select **Overwrite Existing Messages**, the messages in an existing protocol will be overwritten, but you can still reload them if you want to get them back. (This option is available only if you are regenerating an existing protocol.)
   - If you select **Generate Associations**, UML Designer will create associations between the generated messages and the responsibilities they implement. (This option is selected by default.)
   - If you select **Generate Messages**, UML Designer will generate messages in the protocol. Otherwise, the generated protocol is empty. (This option is selected by default.)

   For our example, accept the default options and select **OK** to generate the protocol.

## Message specifications

To take a look at the protocol we just generated, select **Protocols** in the middle pane of the Relationships Browser and then double-click on <*Catalog*>. A browser opens from which we can explore the <*Catalog*> protocol.

In the new browser, select **Messages** in the middle pane. In the rightmost pane, you should see a list of the message specifications that were automatically generated for the protocol, based upon the responsibilities we defined earlier. UML Designer uses a straightforward transformation to convert responsibility names into message names that could eventually become Smalltalk or Java method names. If the generated message names are not satisfactory, you can change them.

For the <*Catalog*> protocol, the following message specifications were automatically generated from the responsibilities of the *Catalog* thing:

| **Responsibility** | **Generated message specification** |
|---|---|
| **Add catalog entry** | *#addCatalogEntry:* |
| **Assign accession number to entry** | *#assignAccessionNumberToEntry:* |
| **Find entry by search criteria** | *#findEntryBySearchCriteria* |
| **Remove catalog entry** | *#removeCatalogEntry:* |

Note that some of the generated message specifications end in colons, indicating that they take parameters. This is because the corresponding responsibilities each has a collaborating participant defined. To see more information about parameters, double-click on *#addCatalogEntry:*. A browser opens on the *#addCatalogEntry:* message specification, showing additional details of the message specification.

Select **Parameters** in the middle pane of the browser. In the rightmost pane, you should see a single parameter listed: *aCatalogEntry.* This parameter was automatically generated because the *Add catalog entry* responsibility had a link to *Catalog entry* as a collaborating participant. Because of this relationship, UML Designer assumes that an object representing the participating *Catalog entry* thing will have to be passed to the method implementing the *Add catalog entry* responsibility.

The type of a parameter must be defined by a protocol, and at this stage we have not defined a protocol for *Catalog entry*. Therefore, the parameters of the generated message specifications all have the default type <*Object*>, even though the generated parameter names (such as *aCatalogEntry*) might suggest a different type. Later, after you have generated protocols for all of the things in your design, you can change the parameters of the message specifications to the correct types.

## Generating using idioms

For another Thing element, *Catalog Entry*, we defined several responsibilities using the **Value** idiom. Let's see how these responsibilities become protocol messages:

1. Generate a protocol for *Catalog Entry* using the same procedure you used for *Catalog* (select **Transform→Generate Protocol** and accept the default options).
2. After you generate the protocol, select **Protocols** from the list of relationships and then select the new protocol, <*CatalogEntry*>.
3. Double-click on <*CatalogEntry*> to open a browser on the protocol.
4. In the middle pane of the new browser, select **Messages**. You should see a list of messages defined in the protocol:
   - *accessionNumber*
   - *accessionNumber:*
   - *author*
   - *author:*
   - *title*
   - *title:*

   These messages are the getters and setters for the values kept by the catalog entry. UML Designer automatically generated these messages based on the three responsibilities we specified for the *CatalogEntry* Thing.
5. In the middle pane of the browser, select **Attributes**. You should see a list of attributes defined in the protocol:
   - *accessionNumber*
   - *author*
   - *title*

   These attributes represent the values kept by the catalog entry. UML Designer also generated these attributes based on the three responsibilities we specified for the *CatalogEntry* Thing.

## Changing parameter and attribute types

When we defined the responsibilities of *Catalog Entry*, we didn't specify any participants. We can now make changes to the generated messages to specify the appropriate types for the messages and attributes.

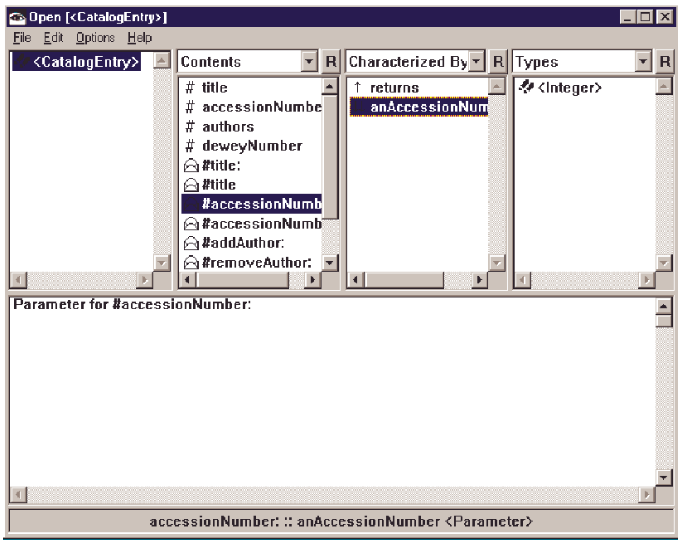
This time, let's use the Path Browser, which will allow us to see everything we need in one browser. The Path Browser shows the successive navigation of not just one relationship, but of up to four relationships, starting from the selected source element. With the Path Browser, therefore, we can browse a protocol, its message specifications, the parameters of a selected message, and the type of a selected parameter all in the same browser. (See "The Path Browser" on page 30 for more information about the Path Browser.)

## Opening the Path Browser

To open the Path Browser on the <*CatalogEntry*> protocol, follow these steps:

1. In the Relationships Browser, select **Protocols** in the middle pane.
2. Select <*CatalogEntry*> in the list of protocols.
3. Select **Open With→Path Browser** from the pop-up menu of <*CatalogEntry*>.

The leftmost pane of the Path Browser shows the starting point of the navigation (the <*CatalogEntry*> protocol). Each successive pane shows a navigation from the previous pane, as indicated by the selection in the drop-down list. The default navigation for a protocol, **Contents**, appears in the second pane.



Parameter for #accessionNumber:

accessionNumber: :: anAccessionNumber <Parameter>

We can now use the Path Browser to browse the attributes and message specifications of the <*CatalogEntry*> protocol, and make the changes we need to make.

## Changing attribute types

To change the types of the attributes of <*CatalogEntry*>, follow these steps:

1. Select *#accessionNumber* from the **Contents** pane of the Path Browser.

   The next pane shows the default navigation, the types (protocols) currently linked to the attribute. Because we did not specify any participants, the type of *#accessionNumber* is defaulted to <*Object*>.
2. Select **Link→Types** from the pop-up menu of *#accessionNumber*.
3.

   Use the **<<** and **>>** buttons to add <*Integer*> to the **Linked** list and remove <*Object*>. Select **OK** when you are finished.
4. Repeat the same process for *#author* and *#title*, specifying <*String*> as the type for each.

## Changing message parameter types

In addition to the attributes, the messages of <*CatalogEntry*> also need to be changed to have parameters and return values of the correct types. To change the parameter types, follow these steps:

1. Select *accessionNumber:* in the **Contents** pane of the Path Browser.

The default navigation for a message specification, **Characterized by**, appears in the third pane. This navigation shows all of the parameters and return values of the selected message specification. (If you wanted to see only the input parameters, you could select the **Parameters** navigation in the third pane).

2. Select *anAccessionNumber* in the **Characterized by** pane.

   The default navigation for a parameter, **Types**, appears in the fourth pane. This navigation shows the types (protocols) currently linked to this parameter. Currently, the type is defaulted to <*Object*>.
3. Select **Link→Types** from the pop-up menu of *anAccessionNumber*. A window appears prompting you to select which object types should be linked to the selected parameter.
4. Use the **<<** and **>>** buttons to add <*Integer*> to the **Linked** list and remove <*Object*>. Select **OK** when you are finished.
5. Use the same procedure to change the parameter types for the other setter methods, *author:* and *title:*. Both should take parameters of type <*String*>.

### Changing return value types

Changing a return value type is similar to changing a parameter type. To change the return value types for the messages of <*CatalogEntry*>, follow these steps:

1. Select *accessionNumber:* in the **Contents** pane of the Path Browser.
2. Select *returns* in the **Characterized by** pane. The **Type** pane indicates that the return type is currently unspecified.
3. Select **Link→Types** from the pop-up menu of *returns*. A window appears prompting you to select which object type should be linked to the selected parameter.
4. Use the **>>** to add <*Integer*> to the **Linked** list. You do not need to remove anything from the list; a return value has only one type by definition, so the previous value is automatically removed. Select **OK** when you are finished.
5. Use the same procedure to change the parameter types for the other methods. *accessionNumber* should return <*Integer*>, while the getter and setter methods for author and title should all return <*String*>.

## Defining a message manually

Suppose we now decide we want to add getter and setter messages to <*CatalogEntry*> for storing the Dewey number. We could go back and define this as a responsibility of the *Catalog Entry* Thing and regenerate the protocol, but let’s try directly adding the responsibility to the protocol.

To add the new messages, follow these steps:

1. Select **Protocols** in the Relationships Browser.
2. Select <*CatalogEntry*> from the list of protocols.
3. Select **New→Message Specification** from the pop-up menu of <*CatalogEntry*>.
4. When prompted for the message specification, type the following (be sure to include spaces on either side of the colon):

   ```
   deweyNumber : <String>
   ```

   This is the formal UML syntax declaring a message specification. It indicates a unary message with no parameters that returns an object of type <*String*>. This is the getter message for the *deweyNumber* attribute.

5. Use **New→Message Specification** again to add the setter message for the *deweyNumber* attribute:

   ```
   deweyNumber: (number: <String>): <String>
   ```

   This is the UML syntax declaring a unary message taking one parameter of type <*String*> and returning a <*String*>.

# Chapter 12. Designing classes and building class diagrams

Our analysis so far has given us use cases, actors, things, responsibilities, and protocols. Now we can use the model we have built to begin **design**. Until this point, we have not made any assumptions about any particular implementation technology (other than a general object-oriented approach), but it’s now time to start mapping the model we have built to actual Smalltalk or Java classes to implement it.

This design phase includes designing classes — in which you decide what your classes will be and which protocols they conform to — and establishing associations and links between them. Although you can create these relationships nonvisually by using the Relationships Browser, it is probably easier to do so from within the Class Diagrammer.

## Opening the Class Diagrammer

To begin building a class diagram for the library catalog example, follow these steps:

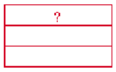
1. In the Relationships Browser, select **Class Diagrams** in the middle pane.
2. Select **New** from the pop-up menu of the relationship.
3. When prompted for the name of the new class diagram, type *Library Catalog*.
4. Select **OK**. The new Class Diagram element appears in the rightmost pane.
5. Double-click on *Library Catalog* in the rightmost pane. The Class Diagrammer opens on the new diagram.

## Adding a class design figure

To add a class design to the diagram, follow these steps:

1. Select **Class** from the tool bar.
2. Place the new figure on the drawing surface.

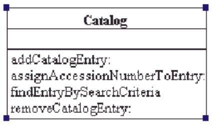
   When first added to the drawing surface, the new figure is labeled with a question mark. This is because it is not yet associated with any underlying model element (in this case, a class design).

3. Click mouse button 2 on the figure to display its pop-up menu.
4. Select **Attach** from the pop-up menu.
5. A window appears prompting you for the class design you want to link to. The one we want doesn’t exist yet, so instead of selecting from the list of available class designs, type *Catalog* in the **Specify New Item** field at the bottom of the window.
6. Select **New**.
7. The next window confirms the name of the new class design, and also confirms that we want to create a real Smalltalk class in addition to the class design. From here, you can specify the superclass and the Smalltalk application for the

new class; if you specify a class that already exists, UML Designer attaches the class design to the existing class. Accept the defaults and select **OK** to create the class design.

The new class design is created and attached to the figure, which is now labeled **Catalog**.



When first created, the new class design figure is empty; it does not show any methods or attributes. If the underlying Catalog class (which we just created) had any methods or instance variables, they would appear in the class design figure. (If you want to see the class definition, double-click on the Catalog class figure, or select **Open With→Class Browser** from its pop-up menu, to open a class browser.) In our case, however, we want to use a protocol to indicate the interface of the Catalog class.

## Establishing protocol conformance

The *Catalog* class is the implementation of the catalog, which we identified as a thing earlier in our analysis. The *Catalog* class, therefore, will implement the responsibilities of the catalog, which means that it will conform to the <*Catalog*> protocol. (For the purpose of this discussion, the *class* and the *class design* are effectively the same once we have linked them together.)

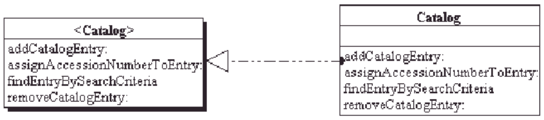
The <*Catalog*> protocol defines the interface of the catalog object in the system. The protocol includes message specifications that define the messages the catalog object must respond to, what types of parameters those messages require, and what types of values they return. To conform to the <*Catalog*> protocol, the *Catalog* class must implement an actual Smalltalk method for each of the messages specified in the protocol. UML Designer can automatically generate the appropriate stub implementations for these methods when protocol conformance is established.

### Adding a protocol figure

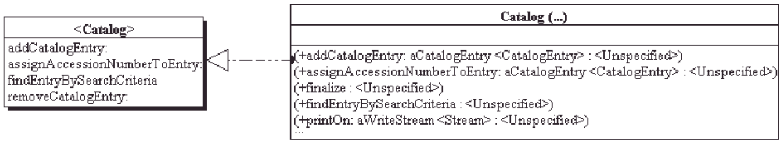
To establish a conformance relationship between the *Catalog* class and the <*Catalog*> protocol, follow these steps:

1. In the Class Diagrammer, select **Protocol** on the tool bar.
2. Add the Protocol figure to an empty area of the drawing surface.
3. Select **Attach** from the pop-up menu of the Protocol figure.
4. When prompted for the protocol to link to, select <*Catalog*>. Then select **OK**.
5. Select **Conformance** on the tool bar.
6. Click mouse button 1 on the Catalog class design figure to indicate that it is the source of the relationship.
7. Click mouse button 1 on the <Catalog> protocol figure to indicate that it is the destination of the relationship.

When you complete the connection, you should see methods appear in the Catalog class design figure to show that it conforms to the <*Catalog*> protocol.


<Catalog>
addCatalogEntry:
assignAccessionNumberToEntry:
findEntryBySearchCriteria
removeCatalogEntry:
Catalog
addCatalogEntry:
assignAccessionNumberToEntry:
findEntryBySearchCriteria
removeCatalogEntry:

Now select **Display Options→Method Name Only** from the pop-up menu of the Catalog class figure. This turns off the **Method Name Only** toggle so you can see the complete method signatures.


<Catalog>
addCatalogEntry:
assignAccessionNumberToEntry:
findEntryBySearchCriteria
removeCatalogEntry:
Catalog (...)
(+addCatalogEntry: aCatalogEntry <CatalogEntry> : <Unspecified>)
(+assignAccessionNumberToEntry: aCatalogEntry <CatalogEntry> : <Unspecified>)
(+finalize : <Unspecified>)
(+findEntryBySearchCriteria : <Unspecified>)
(+printOn: aWriteStream <Stream> : <Unspecified>)
...

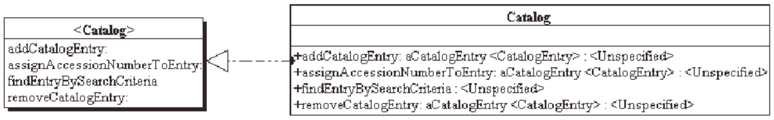
The method signatures include specifications of the types of the parameters and the return values of the methods. A plus sign (+) before a method signature indicates that the method is specified (it is taken from the protocol the class conforms to). Parentheses around the method name indicate that the method is **unimplemented**, meaning that no real method implementation exists in the underlying Smalltalk class.

## Generating stub method implementations

At this point, although we have specified that the class design conforms to a protocol, the actual *Catalog* class does not yet have real methods corresponding to the ones in the class design. To have UML Designer generate stub implementations, follow these steps:

1. Select **Transforms→Generate Real Class** from the pop-up menu of the Catalog class figure.
2. In the window that appears, select **Catalog** as the name of the class to generate.
3. Select **Replace Existing** to indicate that we want to regenerate a class that already exists in the image.
4. Select **OK**.

After the operation completes, the notation of the method signatures in the class figure should change to indicate that the methods are now implemented.


<Catalog>
addCatalogEntry:
assignAccessionNumberToEntry:
findEntryBySearchCriteria
removeCatalogEntry:
Catalog
+addCatalogEntry: aCatalogEntry <CatalogEntry> : <Unspecified>
+assignAccessionNumberToEntry: aCatalogEntry <CatalogEntry> : <Unspecified>
+findEntryBySearchCriteria : <Unspecified>
+removeCatalogEntry: aCatalogEntry <CatalogEntry> : <Unspecified>

To see the actual implementations, double-click on the Catalog class figure (or select **Open With→Class Browser** from its pop-up menu) to open a class browser on the *Catalog* class. You should see a new method category, *generated*, which contains stub implementations of the methods specified in the <*Catalog*> protocol.

## Adding more elements

We've now diagrammed the *Catalog* class and its conformance to the <*Catalog*> protocol. In order to completely diagram our design as it stands, we need to add the other classes and protocols we've come up with so we can show their relationships to one another.

To add the other classes and protocols, follow these steps:

1. Select **Class** on the tool bar to add a second class figure to the diagram.
2. Select **Attach** from the pop-up menu of the new figure.
3. Attach the new figure to a new class design called *Library*. Specify that you want to generate a real Smalltalk class.
4. Use the same procedure to add another class figure attached to a class design (and Smalltalk class) called *CatalogEntry*.
5. Now select **Protocol** on the tool bar to add a second protocol figure to the diagram.
6. Select **Attach** from the pop-up menu of the new figure.
7. Attach the new figure to the <*CatalogEntry*> protocol.

You should now have five design elements represented on the diagram:

- *Catalog* class
- <*Catalog*> protocol
- *CatalogEntry* class
- <*CatalogEntry*> protocol
- *Library* class

## Creating associations

To complete our first pass at a class diagram, we must now connect the design elements to one another to show the associations with one another. As you make connections between design elements, you can move the elements around on the drawing surface in order to keep the diagram neat and readable. The visual arrangement of the figures in the diagram has no effect on the underlying model.

### A closer look at associations

Before we add associations to our class diagram, we need to take a close look at associations and their attributes.

The associations we've created up until this point (such as the connection between the *Catalog* class and the <*Catalog*> protocol) are more correctly called links. In UML terms, a **link** is an instance of an association, meaning that a semantic relationship exists between the linked elements. A link represents an instance of one of the types of relationships predefined by the UML and UML Designer metamodels. Protocol conformance is an example of such a relationship; the link between the *Catalog* class and the <*Catalog*> protocol is a specific instance of a conformance relationship.

On the other hand, an **association** defines a *type* of relationship, but not a specific instance of such a relationship. In other words, an association defines a possible connection between objects in the system defined by *your* model, rather than being part of the UML or UML Designer metamodel. Another way of looking at it is that an association describes a possible set of links. An association has properties, such as multiplicity, that place constraints on how it can be instantiated.

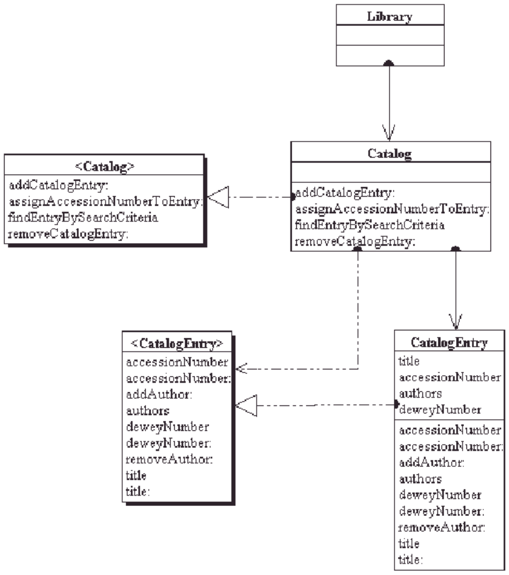
## Adding associations to the diagram

To show the associations between the elements in the library catalog system, follow these steps:

1. Use **Conformance** to connect the *CatalogEntry* class figure to the <*CatalogEntry*> protocol figure. This indicates that the *CatalogEntry* class conforms to the <*CatalogEntry*> protocol.
2. Use **Association** to connect the *Catalog* class figure (the source) to the *CatalogEntry* class figure (the destination). This indicates that an association exists between the two elements (a *Catalog* must keep track of *CatalogEntry* objects).

   When first created, an association is labeled with the default generated name, which is based on the name of the destination element. We will change this label later to more correctly describe the association.
3. Use **Association** to connect the *Library* class figure (the source) to the *Catalog* class figure (the destination). Again, this indicates that an association exists (in this case, a *Library* uses one or more *Catalog* objects to keep track of its books).
4. Finally, use **Dependency** to connect the *Catalog* class figure (the source) to the <*CatalogEntry*> protocol figure (the destination). This indicates that the *Catalog* class figure depends upon the interface specified by the <*CatalogEntry*> protocol. This is because some of the methods of *Catalog* require parameters of type <*CatalogEntry*>.

With all of the relationships added, the relationships in the diagram should look something like this (you may have arranged the figures differently):

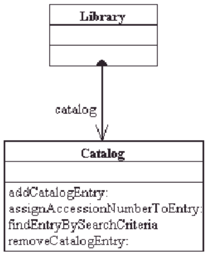


## Showing multiplicity

For our system, we can assume that a library has only one catalog, so the default association name *catalog* is fine. However, we need to explicitly show the multiplicity of the association:

1. Select the association between *Library* and *Catalog*.
2. Select **Multiplicity→Edit From** from the pop-up menu.
3. In the window that appears, specify
4. Now select **Multiplicity→1-to-1** from the pop-up menu.

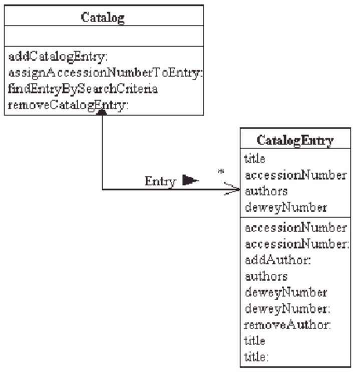
The association between *Library* and *Catalog* should now look something like this:



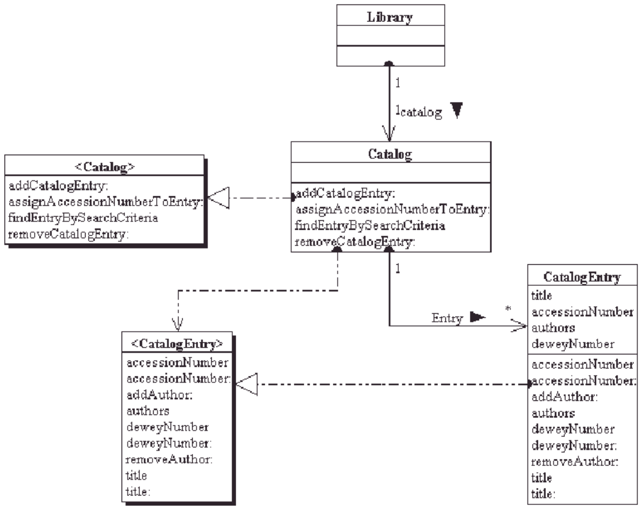
We also need to show multiplicity on the association between *Catalog* and *CatalogEntry*. In this case, each catalog must work with multiple entries, though each entry will be associated with only one catalog. To label this association, follow these steps:

1. Select the association between *Catalog* and *CatalogEntry*.
2. Select **Name→Show To** from the pop-up menu. The default association name, *catalog entry*, appears.
3. Because each catalog is associated with multiple entries, we need to change the association name to be plural. Hold down the Ctrl key and click mouse button 1 on the label to directly edit it so it reads *entries*. When you are finished editing the label, press Enter or click elsewhere on the diagram.
4. Select the association again.
5. Select **Multiplicity→1-to-n** from the pop-up menu.

The association between *Catalog* and *CatalogEntry* should now look something like this:



The complete diagram, with labels, should now look something like this (again, your diagram might be arranged differently):



This diagram shows the relationships between the objects we have defined in initial iteration: *Library*, *Catalog*, and *CatalogEntry*. It also shows protocol conformance and the dependency relationship between *Catalog* and the <*CatalogEntry*> protocol. This tells us that any class that conforms to the <*CatalogEntry*> protocol could be substituted for *Catalog* in this system.

# Chapter 13. Modeling existing Smalltalk classes

In some situations, your model might need to describe one or more Smalltalk classes that have already been implemented, “reverse engineering” analysis and design artifacts.

With UML Designer, you can accomplish this by creating a corresponding class design in your model for each existing Smalltalk class. Any Smalltalk class can be attached to a class design, whether or not the class was generated by UML Designer. You can also create protocols based on the methods of existing classes.

The simplest way to attach a class design to an existing class is to use the class name as the class design name. When you create a class design with the same name as an existing class, UML Designer by default suggests the existing class when prompting which class the class design should be attached to. By accepting the defaults, you can quickly create a class design and link it to an existing class.

However, you can also create a class design with a different name. When prompted for the real class to link to, you can select any class that exists in the image.

## Attaching a class design to an existing class

For example, suppose we have a new requirement for the library catalog system: we now want to track circulation, keeping records of which patrons have checked out which books. This means we will almost certainly need a class to represent borrowers. We could choose to build a new *Borrower* class for this purpose; but suppose we then find out that we already have a *LibraryPatron* class that was implemented for a system that prints library cards. The *LibraryPatron* class has the following attributes, each with a getter and setter method:

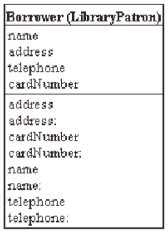
- name
- address
- telephone
- card number

This seems to be everything we need, so now we just need to incorporate the *LibraryPatron* class into our *Library Catalog* model. To do so, follow these steps:

1. Make sure a *LibraryPatron* class exists in the image. (You can create the class yourself using the standard Smalltalk browsers; just add instance variables and stub getters and setters for the attributes above.)
2. In the Class Diagrammer, add a new ▤ class figure to the drawing surface.
3. Select **Attach** from the pop-up menu of the new figure.
4. When prompted for the class design to link to, type *Borrower* in the **Specify New Item** field.
5. Select **New**.
6. In the next window, make sure **Link To Real Class** is selected.
7. Specify *LibraryPatron* in the **Class Name** field. (The name of the underlying Smalltalk class does not have to be the same as the name of the attached class design.)

8. Select **OK**.

The new **Borrower** class figure should now be automatically populated with the attributes and methods from the existing *LibraryPatron* class. Since the name of the class design is different from that of the underlying class, the label of the class figure gives both names.



## Retrieving a protocol

We now have a *Borrower* class design, which shows the methods defined by the *LibraryPatron* class. We can now perform some more reverse engineering and retrieve a protocol from the class design. This can be valuable when modeling an existing system; it makes it possible to formally type the parameters and return values of a class and identify which common interfaces it supports. This can help you find methods that do not accept or return the correct types. For a given class, only some of its methods make up its public API, and these might be grouped into one or more protocols to which the class conforms. You can designate one protocol as its **main protocol—**the one defining the methods making up the primary function of the class.

You can retrieve protocols from any class, even one that is not attached to a class design. When you retrieve a protocol from a class, you can select which method categories are included. In the retrieved protocol, UML Designer will create a message specification for each method in the selected categories.

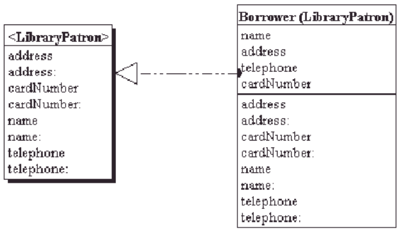
To retrieve a protocol from the methods of the *LibraryPatron* class, follow these steps:

1. In the Class Diagrammer, select the Borrower class figure.
2. Select **Transforms→Retrieve Protocol** from the pop-up menu.
3. In the next window, select the category or categories of the methods you want to retrieve into the protocol. Select **And Categories** if you want to retrieve only the methods that are in *all* the selected categories; select **Or Categories** if you want to retrieve all of the methods that are in *any* of the selected categories.

   In the *Borrower* class, the methods are not categorized, so select *Not categorized*.
4. For now, you can accept the defaults in the other fields in the window. These fields enable you to control the details of how the methods are retrieved into the protocol:
   - In the **Methods** field, you can specify that you want to replace any messages in an existing protocol, if there is one. You can also control whether private, public, or all methods are retrieved. (Typically, a protocol would include only public methods.)
   - In the **Protocols** field, you can specify a name for the protocol that is different from that of the class design.
   - In the **Options** field, you can specify whether you want to include methods of the ancestors of the selected class.

5. Select **OK** to retrieve the protocol.

When UML Designer retrieves a protocol from an existing class design, it automatically creates a “conforms” link between the class design and the retrieved protocol. However, this link does not automatically appear in a class diagram. If you go back to the Class Diagrammer, you can use **Hide/Show Relationships** to make this conformance link visible.



## Retrieving multiple classes at once

The procedure described above is useful for bringing a single existing class into your model. Sometimes, though, you’ll need to create class designs for a whole group of existing classes. This is particularly true in situations where you’re modeling an entire system that has already been implemented.

Rather than bringing each class into the model one at a time, you can retrieve them all at once from the Relationships Browser or the Class Diagrammer. To do this from the Relationships Browser, follow these steps:

1. In the leftmost pane of the Relationships Browser, select the model you want to contain the new class designs.
2. Select **Retrieve Classes** from the pop-up menu.
3. In the **Classes** list, select all of the classes for which you want to create class designs in the model. Use the **Filter** field to limit the list to classes with a specified prefix. Select **All Classes** if you want to see all classes in the image; select **Visible Classes** to see only the classes visible from the current model.

   The visible classes are those in the current model’s Smalltalk application or its prerequisites. If the classes you want to select are not visible, you can use the VisualAge Application Manager to make the application containing the classes you want a prerequisite of the model application.
4. If you want to automatically create corresponding protocols as well as class designs, select **Retrieve Protocols**. (If you are retrieving only a single class, you can also specify a name for the protocol; if you are retrieving multiple classes, UML Designer uses the default, which is the same as the class name.)
5. If you are retrieving protocols, use the **Categories** list to select the method categories you want to include in the retrieved protocols. You must select at least one category to include any message specifications in the generated protocols.
6. Use the **Options** and **Methods** fields to control the details of how methods are retrieved. (See the previous section for more information about these fields.)
7. Select **OK**. Class designs (and protocols, if selected) are now generated and included in the current model.

# Chapter 14. Sequence diagrams

The third type of UML diagram you can build with UML Designer is the sequence diagram.

Like a class diagram, a sequence diagram is another view of the model; each figure in a sequence diagram is attached to an underlying model element. Because a sequence diagram depicts message sends and returns between objects, your class designs must have some defined behavior before you can build a sequence diagram.

A sequence diagram uses instances (rather than class designs) because it shows an actual series of message sends between instantiated objects at run time. However, you can also create and use instances in class diagrams or in the Relationships Browser.

## Creating a sequence diagram

For the library catalog system, we might want to create a sequence diagram that shows adding a new book to the catalog (one of our use cases). The sequence diagram will be a new model element added to the *Library Catalog* model.

To start working on a new sequence diagram, follow these steps:

1. In the Relationships Browser, select **Sequence Diagrams** in the middle pane.
2. Select **New** from the pop-up menu of the relationship.
3. When prompted, specify *Adding a book* as the name of the sequence diagram. The new diagram element appears in the list in the rightmost pane.
4. Double-click on the new diagram. The Sequence Diagrammer opens.

## Working with the Sequence Diagrammer

### Adding objects to a sequence diagram

Three objects are involved in the “Adding a book” scenario:

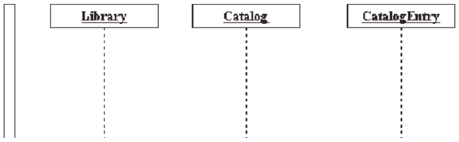
- A *Library* object, representing the system context; this object creates the new catalog entry and initiates the cataloguing operation.
- A *Catalog* object, which reads the necessary information from the catalog entry and assigns it an accession number.
- A *CatalogEntry* object, which is created and cataloged.

To add these objects to the sequence diagram, follow these steps:

1. Select **Object** from the tool bar.
2. Use mouse button 1 to place the new figure near the left edge of the drawing surface (adjacent to the system activation).

   An object figure appears by default as an icon at the top of the drawing surface with a dashed **lifeline** extending downward. The vertical placement of the mouse pointer does not matter when you add a new object to a sequence diagram; only its horizontal position is important.
3. Select **Attach** from the pop-up menu of the new object figure.

4. When prompted, select *Library* as the class design to link to. UML Designer automatically creates a new instance element for the class design you select.
5. Select **OK**. The object figure is now labeled **Library**.
6. Repeat the same procedure for the other two object figures, attaching them to *Catalog* and *CatalogEntry* respectively.



## Adding method calls to the diagram

To show the sequence of events involved in adding a book to the catalog, you must now add figures representing the method calls between the objects in the diagram. Each figure shows a single method call between a sender and a receiver, as well as the return after the method has completed. In addition, the sequence diagram also shows activation on each object's lifeline, so you can see which objects are active (that is, which objects are currently executing in response to method calls) at any given time.

In general terms, the sequence of events involved in adding a book to the catalog goes something like this:

1. A librarian (an actor outside the system) uses the library system interface to create a new catalog entry. The new catalog entry records the information describing the new book.
2. The librarian uses the library system interface to add the new catalog entry to the catalog.
3. The catalog queries the catalog entry for its descriptive information for indexing purposes.
4. The catalog allocates an accession number and assigns it to the new entry.

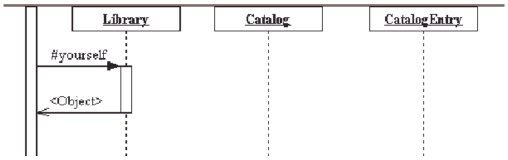
### Adding the initial method call

The first method call on a sequence diagram must originate from the system activation at the left edge (only an active object can send a message to another object). This initial method call represents a call from outside the diagram; in our diagram, it represents input from an outside actor initiating the cataloging operation. We have not yet defined an actual method for *Library* that would accept external input, so for now we can use the standard *yourself* method as a placeholder. This method call does not itself have any significance; it is used simply to activate the *Library* object.

To add this method call to the diagram, follow these steps:

1. Select **Method call** from the tool bar.
2. Click mouse button 1 on the system activation to indicate the source of the message (in this case, an object outside the diagram).
3. Click mouse button 1 on the *Library* object's lifeline to indicate the receiver of the message.

4. When prompted to choose a method, select *Object* in the **Classes** list, *Object* in the **Protocols** list, and #*yourself* from the **Messages** list. This indicates that the *yourself* method (inherited from *Object* and defined in the <*Object*> protocol) is the target of the call.
5. Select **OK**. The method call to *yourself*, and its return, appear as a method call figure. In addition, the *Library* object's lifeline now shows activation between the message send and return.



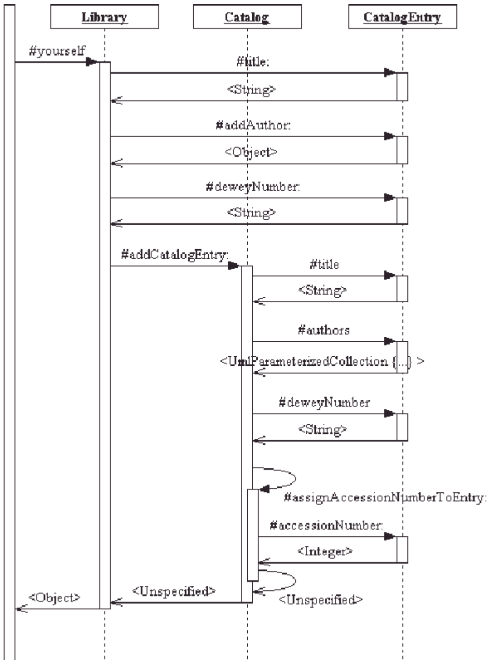
6. Select the activation for the *yourself* method call on the *Library* object's lifeline.
7. Use mouse button 1 to drag the bottom selection handle downward, lengthening the activation. (The activation of the *Library* object lasts for the duration of the cataloging operation.)

   As you add more method calls, you can continue to lengthen the activation as needed. You can also move a method call and its associated activation up or down by dragging the center of the activation.

## Adding the remaining method calls

To build the rest of the sequence diagram, we need to add a series of instance method calls. For each method call, select **Method call**, click on the source object's lifeline (within an activation), and then click on the target object's lifeline. Remember that you can resize or move activations in order to make room for method calls, and you can also move the object figures to the left or right in order to make the diagram readable.

Use the following illustration as a guide to show you how to construct the diagram.



The sequence diagram shows the following series of method calls:

1. After instantiating a new *CatalogEntry* object, the *LibraryObject* uses setter methods to set the *title*, *author*, and *deweyNumber* attributes of the new catalog entry.
2. *Library* then sends the message *addCatalogEntry:* to *Catalog,* which tells it to take the new entry, index it, and assign it an accession number.
3. *Catalog*, now active, uses getter methods to retrieve the *title*, *author*, and *deweyNumber* attributes of the catalog entry. (At this point there would be some additional processing to index the new entry, but that behavior is not currently part of our model.)
4. *Catalog* sends the message *assignAccessionNumberToEntry:* to itself. This method allocates a new accession number that can then be assigned to a catalog entry. Note the stacked activations on the *Catalog* lifeline, indicating nested method execution.
5. Finally, *Catalog* sends *accessionNumber:* to the *CatalogEntry*, assigning it the newly allocated accession number.

# Chapter 15. Publishing models

The UML Designer publishing features can generate formatted output based on the contents of your model (or a subset of your model), in either HTML or RTF format.

Publishing is a two-step process:

- The first step is to build a Publication element as part of the model. The Publication element in turn contains a set of Topic elements, each of which is linked to a model element.
- The second step is to generate HTML or RTF output based on the Publication element. The output produced can then be browsed or printed.

A Publication element defines the structure and content of a published document based on your model. It consists of a series of Topic elements, arranged in a hierarchical fashion; the arrangement of Topic elements corresponds to the organizational structure of the document that will be produced.

Each Topic element contains a text element; this text is what will appear in the final output. The text can consist of literal text or hypertext links (which will be resolved when you generate the final output).

There are two basic ways to publish your model:

- Automatically generating a publication element and output in one step
- Manually building a publication element and generating output

## Publishing automatically

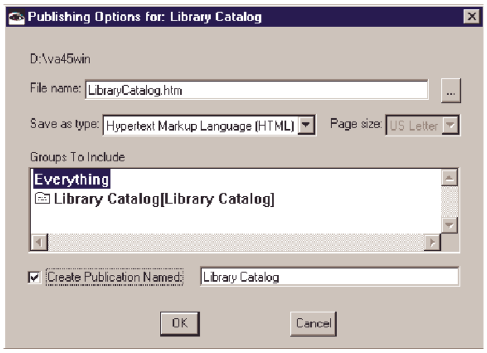
The fastest way to publish your model is to automatically generate the Publication element and output document in a single step. This is generally the easiest way to create published output, and you can publish any model this way.

To publish automatically, follow these steps:

1. In the Relationships Browser, select the element you want to publish. This can either be the model element or an element within the model.
2. If you want to include the element and any elements it links to, select **Publish→With Children** from the pop-up menu. For example, you might use this option to publish an entire model, or to publish a protocol along with all of its messages, parameters, and return values.

   If you want to include only the selected element, select **Publish→Single**. The resulting publication will include only the text of the selected element.

A dialog now appears prompting you for the publishing options you want to use.



Publishing Options for: Library Catalog

D:\va45win

File name: LibraryCatalog.htm

Save as type: Hypertext Markup Language (HTML)

Page size: US Letter

Groups To Include

Everything

Library Catalog[Library Catalog]

☑ Create Publication Named: Library Catalog

3. In the **File name** field, specify the name of the output file. Select the **...** button to open a file dialog from which you can select a location for the output file.
4. In the **Save as type** field, select the format of the published output:
   - Select **Hypertext Markup Language (HTML)** if you want to create online output viewable with a Web browser.
   - Select **Rich Text Format (RTF)** if you want to create a hardcopy document. RTF is compatible with most word-processing software such as Microsoft Word.
5. In the **Page size** field, select the page size you want to use for an RTF document. (This field is disabled if you have selected another output format.)
6. In the **Groups To Include** field, select which groups you want to include in the published document. To include all elements regardless of group membership, select **Everything** (this is the default).
7. Select the **Create Publication Named** check box if you want to save the Publication element that is automatically created. This is not necessary if you are only interested in the final output. There are several reasons you might want to save the Publication element:
   - To speed up processing the next time you publish the model. By saving the Publication element, you can avoid having to generate it again later. (However, remember that if you use the same Publication element later, it won't reflect any elements or links you have added or removed.)
   - To manually modify the Publication element. You can add, remove, or rearrange topics and the text they contain, and then regenerate the formatted output. (For more information about modifying a Publication element, see "Publishing manually" on page 105.)
8. Select **OK** to start publishing.

After the publishing operating finishes, you can open the output document using the appropriate tool. If you selected RTF output, there will be a single file with the name you specified. If you selected HTML output, there will be one or more HTML files; the top-level file has the name you specified, while the other files use the same name with a numeral appended.

## Including diagrams in a publication

If your RTF or HTML publication includes any diagrams, the generated document will contain references to these diagrams as Graphic Interchange Format (GIF) files. However, the publication process does not generate the GIF files automatically. If the GIF files are not present when you view or print the document, the diagrams will not appear.

To create a GIF file of a diagram, follow these steps:

1. Open the diagram using the appropriate UML Designer diagrammer.
2. Adjust the zooming and centering of the diagram so it appears in the diagrammer exactly as you want it to appear in the published document.
3. Select **Save As GIF** from the **File** menu.
4. Accept the default generated file name for the GIF file (the published document will use this name to refer to the diagram). Save the GIF file to the same directory that contains your published document.

If a diagram changes, you must repeat this process to update the GIF file. Because the graphics are stored in separate files, it is not necessary to repeat the publication process if only the diagrams have changed.

## Publishing manually

If you prefer to customize your document, you can manually build the Publication element from which the formatted output is generated. You can do this either by creating a Publication element and adding topics to it one by one, or by automatically generating a Publication element and then modifying it. (See "Publishing automatically" on page 103 for more information about automatically generating a Publication element.)

### Creating a Publication element

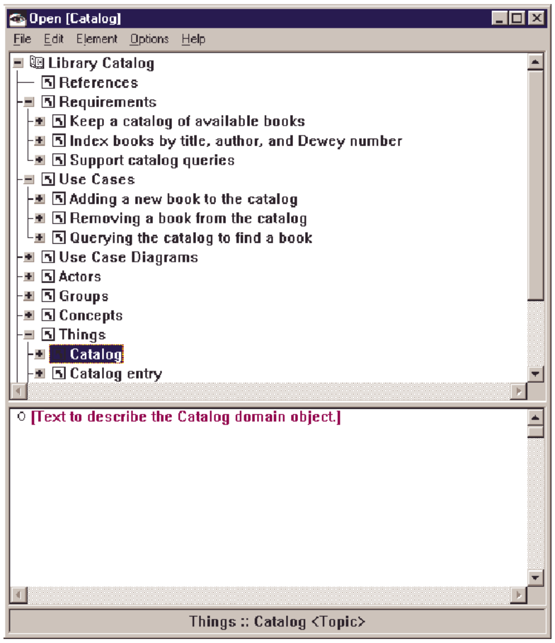
To create a new, empty publication, follow these steps:

1. In the Relationships Browser, select **Publications** in the list of relationships. (If **Publications** doesn't appear in the list, select the **All Relationships** filter.)
2. Select **New** from the pop-up menu.
3. When prompted, specify a name for the new Publication element.

The new Publication element appears in the list of elements.

### Editing a Publication element

To edit the contents of a Publication element, double-click on the element in the Relationships Browser. (You can do this with any Publication element, whether created manually or automatically.) A Focus browser opens showing you the contents of the publication.


Open [Catalog]
File Edit Element Options Help
Library Catalog
References
Requirements
Keep a catalog of available books
Index books by title, author, and Dewey number
Support catalog queries
Use Cases
Adding a new book to the catalog
Removing a book from the catalog
Querying the catalog to find a book
Use Case Diagrams
Actors
Groups
Concepts
Things
Catalog
Catalog entry
Things :: Catalog <Topic>

[Text to describe the Catalog domain object.]

The top pane of the Focus browser shows the hierarchical structure of the publication. Each entry in the top pane is a Topic element, and their arrangement reflects the organization of the finished document. (If the publication is a new one you created manually, it contains only a single top-level topic representing the publication as a whole.)

If a topic has subtopics, a plus sign (+) appears to the left of the topic in the list. To see the subtopics, click on the + to expand the list. Click on the minus sign (–) to collapse the list.

The bottom pane shows the hypertext contents of the selected topic. This is the text that will appear in the formatted output; it can also contain links to other model elements, which will be resolved in the final document.

## Adding and removing topics

To add a topic to the publication, follow these steps:

1. Select the topic you want to add a subtopic to. (For a top-level document division, select the topic representing the publication as a whole.)
2. Select **New→Topic** from the pop-up menu.
3. When prompted, specify the name you want to give the new topic.
4. Select **OK** or press Enter.

The new topic appears in the browser (you might need to click on + to see the subtopics).

To remove a topic, select the topic you want to remove and then select **Delete** from the pop-up menu.

## Reordering topics

There are several ways you can change the order and placement of topics in a publication.

- Select **Ordering→Promote** to move a topic up a level in the document structure (putting it at the same level as its current parent).

- Select **Ordering→Demote** to move a topic down a level in the document structure (making it a subtopic).
- Select **Ordering→Up** or **Ordering→Down** to change a topic's place in the document by one position, without changing its level.
- Select **Ordering→Reorder To** to move a topic elsewhere in the publication. A window appears prompting you for the location you want to move the topic to.

### Editing text

To add text to a topic, select the Topic element in the Focus browser and type the text in the hypertext pane. Any text you enter will be included in the published document.

In addition, you can include links to model elements, which will cause them to be included in the publication. There are two kinds of links:

- A **reference link** causes a reference to an element to appear in the published topic. In an HTML document, the reference is a hypertext link to a separate topic containing the element text. (A reference link causes an additional topic containing the element text to be included automatically when published.)

  In an RTF document, a reference link causes the name of the element to appear in the topic text, and a separate topic containing the element text is automatically included.

  Reference links are useful in situations where multiple topics refer to the same element. This can avoid unnecessary duplication of text, since all of the links will point to the same topic.
- An **inline link** causes the element's text to be included directly in the topic text.

## Generating output

Once you have built your publication, you can use it to generate formatted output just as you would with any other model element.

To generate output from a publication, select the Publication element and then select **Publish** from the pop-up menu. For more information, see "Publishing automatically" on page 103.

# Part 4. Appendixes

# Index