# Constellation™

A Real-Time Software Framework

# UML Guide

*Constellation Version 1.0*

Real-Time Innovations

**Technical Support**

Real-Time Innovations, Inc.
155A Moffett Park Drive
Sunnyvale, CA 94089

| | |
|---|---|
| Phone: | 408-734-4200 |
| Fax: | 408-734-5009 |
| E-mail: | support@rti.com |
| Web Site: | http://www.rti.com |

**Note:** Please send any omissions, corrections, or other items of documentation errata to errata@rti.com.

# Available Documentation

*Constellation* documentation includes:

- ❑ The Overview, Capabilities and Benefits, ConstellationOverview.pdf. This document provides an overview of *Constellation*'s main features and discusses the benefits of using *Constellation*.
- ❑ The Getting Started Guide, GettingStarted.pdf. This document includes installation instructions, system requirements, supported architectures, and compatibility with other products.
- ❑ The Tutorial, ConstellationTutorial.pdf. This tutorial provides basic exercises to give you hands-on experience working in the *Constellation* environment to build, test, and run new components and applications.
- ❑ The User's Manual, Manual.pdf. The manual contains step-by-step instructions on how to use the tools, build components, run and debug applications.
- ❑ The Distributed Applications Guide, DistributedApplications.pdf. This document describes two approaches (NDDS® and CORBA®) for developing *Constellation* applications that need to communicate with other (*Constellation* and non-*Constellation*) applications.
- ❑ The UML Guide, ConstellationUMLGuide.pdf. This document provides an introduction to the Unified Modeling Language (UML) and describes how to create UML diagrams with *Constellation* and use them in your development process.

You can access these documents through the main online documentation file, **Constellation.html**, which also includes extensive online HTML documentation. This file is copied onto your system when you install *Constellation*. It is located in **<your installation path>/cs.8.0*x***, where ***x*** is a version-specific letter. You can open the file directly with any standard web browser such as Netscape® or Microsoft® Internet Explorer, or by starting *Constellation* and using the **Help** menu.

# Contents

# Chapter 1

# Introduction to UML in Constellation

The UML, or Unified Modeling Language, is a language used to model the structure and behavior of real-world systems, frequently software systems. *Constellation* provides support for a concrete component-based software development methodology. UML support in *Constellation* bridges the gap between an abstract model of your system in UML and concrete *Constellation* components. You can also easily synchronize one kind of model to the other.

Throughout this document, we will refer to a simple example that models a refrigerator. You will find this example in the **rti_tutorial** repository.

This document describes how to create UML diagrams with *Constellation* and use them in your development process. While some basic UML concepts are described here, this guide is not intended as a primer on UML. There are several good books available on UML, including Sinan Si Alhir's *UML in a Nutshell: A Desktop Quick Reference (Nutshell Handbook)* n.p.:O'Reilly & Associates; 1998.

## 1.1 Why Use UML Diagrams in Constellation Applications?

UML allows you to create a multi-level view of your system, using a language that can be used throughout the project—by architects and developers alike. With UML, you can isolate low-level implementation details to create a system-level understanding. UML's strength is its ability to provide a common visual language to convey system artifacts.

Its widespread use also makes it a known language; you can create a UML diagram of your system and expect most developers from your team and outside to understand the design.

UML can be used throughout a project's lifecycle, depending on your engineering process. You can use UML in the early phases to map high-level requirements, structure, and behavior. Using UML instead of going straight to implementation allows you to focus on the abstract nature of the system and its building blocks. This is particularly important for complex projects.

During project execution, implementation often needs to deviate from design for reasons such as inadequate design, incorrect assumptions, performance considerations, etc. Therefore it is common to modify UML designs to align them with the actual implementation. It is also useful to check that the requirements captured in a UML model are accounted for in the implementation.

Once a project is over, it is useful to update the UML model again to provide a complete set of final documentation. This can ease project maintenance and provide an accurate set of documentation for future projects that want to leverage off the project.

In summary, the benefits of using UML modeling include:

- ❏ Better high-level understanding.
- ❏ Better requirements tracking.
- ❏ Common way to express design intents.

The costs include the resources and effort to create and maintain the diagrams. This includes training for team members not familiar with UML.

## 1.2 Overview of UML Diagrams

Systems are described with *models*; models are depicted and manipulated with *diagrams*. Each UML diagram provides a view into an associated model of a system. The diagrams can be classified into the following categories:

- ❏ **Requirements diagrams** These diagrams are geared towards capturing the requirements of a project. These types of diagrams answer the question "How will the final product be used?"

These types of diagrams act as interfaces between the people creating the product's requirements and the designers and architects. They are typically created before the design or the implementation phase.

*Use Case Diagrams*

In *Constellation*, requirements are shown by creating *use case diagrams*, described in Chapter 4.

- **Behavior diagrams** These diagrams (also called interaction diagrams) show the interaction between various parts of the system. They typically model how one part responds to messages sent by other parts.

  It is common practice to create high-level interaction diagrams in the early phases of a project, typically derived from use case diagrams. During the implementation phase, new interaction diagrams are created to show the low-level interaction that models the expected or implemented dynamic behavior of the system.

  The parts of the model depicted in these diagrams imply certain structural relationships. For example, if two parts are sending messages to each other, they must have the right kind of association, navigability etc.

  *Sequence Diagrams*

  In *Constellation*, behavior diagrams are shown with *sequence diagrams*, described in Chapter 6.

- **Structure diagrams** These diagrams capture the structural relationship between various elements in a product's design. These elements may include classes, interfaces, and components. The purpose of these diagrams is to communicate the high-level design from the designer to the implementors.

  *Class Diagrams*

  In *Constellation*, these structural relationships are shown by creating *class diagrams*, described in Chapter 5.

- **Implementation diagrams** These diagrams capture the implementation details at a higher level than the final code. Use of these diagrams depends heavily on the underlying frameworks.

  *Implementation and State Machine Diagrams*

  All the component types supported by *Constellation*—ATCs, DFCs, STCs, COGs, FSMs, and Applications—fall in this category. Connector types—types, methods, and interfaces also fall into this category. (For descriptions of *Constellation* components and connectors, see the Chapter 2 in the Constellation User's Manual.)

Although each type of diagram has a clear place in the project life-cycle, each can be (and often, should be) used in other phases as well. For example, it is common for structural diagrams to be modified during the implementation phase to align them with the implementation.

## 1.2.1 Supported UML Diagrams

*Constellation* supports the most widely used diagrams from UML:

- ❏ Use Case Diagrams, described in Chapter 4.
- ❏ Class Diagrams, described in Chapter 5.
- ❏ Sequence Diagrams, described in Chapter 6.
- ❏ Component Diagrams, described in Section 1.2.2 and Section 2.3 in the Constellation User's Manual.
- ❏ State Machine Diagrams, described in Section 1.2.3 and Section 2.3 in the Constellation User's Manual.

## 1.2.2 Component Diagrams

*Constellation* supports two kinds of executable component diagrams: COGs and FSMs. These diagrams not only provide the design view of the system but also make it easy to create an executable application. This characteristic of executable-readiness sets these diagrams apart from other diagrams such as class and sequence diagrams. Once you have created a class or sequence diagram, you still need to create an implementation out of it.

Obviously, making a component diagram requires far more details than their higher-level counterparts. Therefore it is not uncommon to first create class and sequence diagrams to get a high-level view and then create executable component diagrams from those.

One way to look at this process is that component diagrams are the *implementation* of structure and behavior diagrams.

The COG diagrams supported in *Constellation* are closely related to UML component diagrams. A COG contains appropriate extensions to make it more accessible to the target application's domain and make efficient use of *Constellation*'s framework. In particular, COG diagrams support modelling data-flow behavior.

*Constellation* applications are specialized COG components that provide the definitions needed to resolve system-level resources such as clocks, habitats, etc.

For more information on COGs, see Chapter 7 in the Constellation User's Manual.

### 1.2.3 State Machine Diagrams

The finite state machine diagrams (FSMs) supported by *Constellation* are closely related to UML state machine diagrams. *Constellation* does, however, support *Constellation*-specific extensions that allow you to create reusable components and make the efficient use of *Constellation* framework.

For more information on FSMs, see Chapter 8 in the Constellation User's Manual.

## 1.3 Key Terms

### 1.3.1 Diagrams

Think of a *diagram* as a canvas on which you can draw certain types of model elements. Model elements are items such as classes, interfaces, COGs, state machines, and the relationships between them (see Section 1.3.2).

*Constellation* supports the following types of diagrams:

- ❏ Use Case Diagrams (Chapter 4)
- ❏ Class Diagrams (Chapter 5)
- ❏ Sequence Diagrams (Chapter 6)
- ❏ Implementation Model Elements (Section 1.4.2)

### 1.3.2 Model Elements

A *model element* represents the underlying information about an entity in a system. Different types of model elements are allowed in each type of diagram. For instance, a class diagram may contain model elements called classes, interfaces, packages, comments, and the relationships between those elements. Model elements called actors and classifier roles only appear in sequence diagrams. The model elements supported in each type of diagram are described throughout the remainder of this document. Model elements may be related to other model elements in any work unit.

Every model element has a name, and optionally, a *stereotype*. Stereotypes allow a meta-classification of model elements. For example, you can attach a "Cog" stereotype to certain classes to indicate that you expect the classes to be implemented as COGs. All elements with the same stereotype can then be described together. For example, you know that all classes with a "Cog" stereotype have one or more habitats.

Stereotypes are indicated with two pairs of angle brackets called guillemets, such as **<<*stereotype*>>**. The FridgeSimulation class in Figure 1.2 shows an example stereotype labeled **<<Cog>>**, which also shows *Constellation*'s icon for a COG component.

### 1.3.3 Depictions of Model Elements

A diagram contains visual depictions of one or more model elements. V*isual* information (size, color, etc.) can differ among depictions of the same model element. Conversely, *non-visual* information about a model element (name, presence of certain attributes, etc.) is shared among all diagrams because it is derived from the model element.

You can drag and drop copies of model elements into diagrams. Then you can change how a model element appears in the diagram, without affecting the actual model element.

A model element may be associated with multiple diagrams. It is also possible that a diagram may show only a part of the information contained in a model element.

For example, one visual rendering of a class element may show all its operations and attributes, but another may show only its attributes, and yet another may show neither attributes nor operations. Presenting information with such filtering is often useful when you only want to show the details that are relevant in a given context.

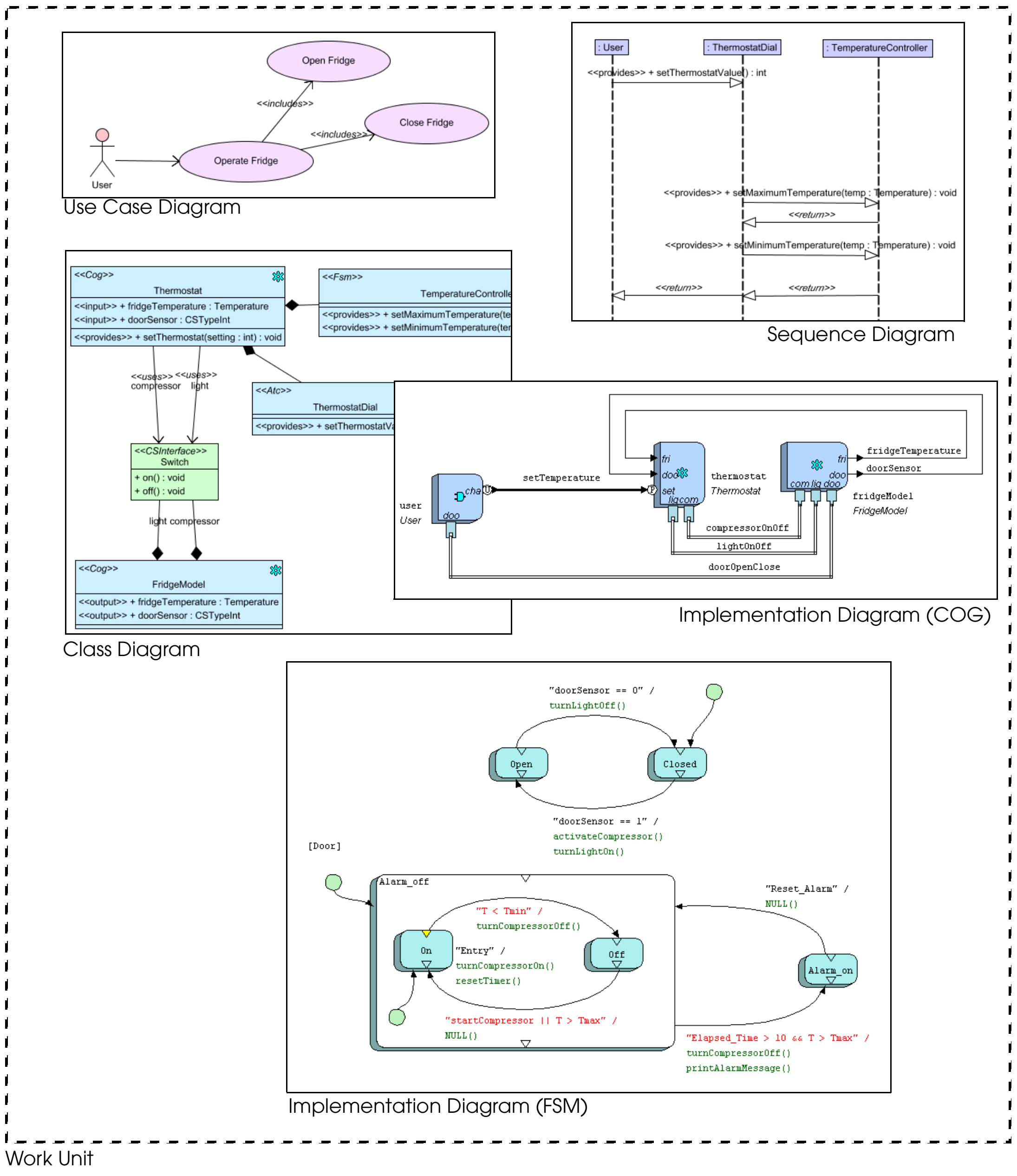
### 1.3.4 Work Units

*Constellation* organizes UML models and diagrams into groups called *work units*. A work unit consists of UML model elements and any number of UML diagrams. The level of granularity within each work unit is up to you.

A work unit may contain zero or more diagrams. You may have several diagrams of each type in a work unit (exception: a work unit may contain only one implementation diagram). Work units may not contain other work units. Diagrams show a subset of the existing model elements in the diagram's work unit and/or in other work units. A work unit and some sample diagrams are seen in Figure 1.1.

Unlike in other tools, elements in *Constellation* work units can refer to elements in other work units, increasing their potential for reuse. For example, imagine that you are modeling a robotic arm positioning application. One work unit might describe the language of the mechanism, joints, motors, and encoders. Then you might have another work unit

Figure 1.1 **Example Diagrams in a Work Unit**



that describes the particulars of the control algorithm, with trajectory planning and control elements that refer to the first work unit.

Suppose members of your team are editing different work units simultaneously and you have a work unit that refers to them. When the other work units are complete, you can pick up their changes simply by updating the referencing work unit.

For more information on work units, see Chapter 3.

### 1.3.5 Model

A *model* is a coherent collection of model elements used for a particular purpose. With *Constellation* this purpose is most likely the design and implementation of an application or a set of applications. Because of the highly reusable nature of *Constellation*'s component approach, the "model" of your application will likely span many work units and even many repositories.

## 1.4 Types of Model Elements: Design and Implementation

There are two kinds of model elements:

- ❏ Design Model Elements (Section 1.4.1)
- ❏ Implementation Model Elements (Section 1.4.2)

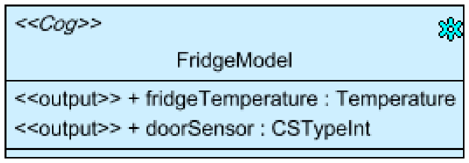
In a typical software engineering process, both kinds of elements go through several phases of modification. These changes are often necessitated by new requirements, the discovery of certain implementation issues, or during an effort to reduce a system's complexity. Chapter 2 discusses this iterative development process.

*Constellation* makes it easy to navigate between UML model elements, diagrams, traditional *Constellation* components, and other related items.

### 1.4.1 Design Model Elements

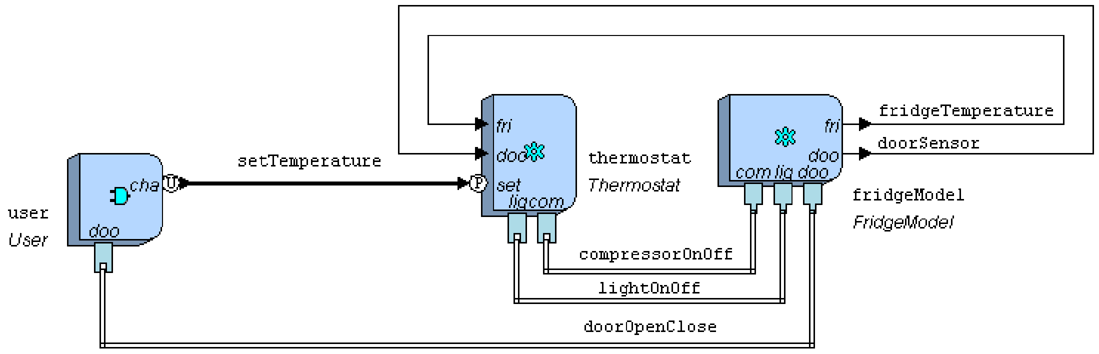
*Design model elements* include class, use case, and similar model elements and the relationships between them, such as associations, generalizations, and dependencies. These elements are typically created prior to entering the implementation phase. However, it is also possible to create these elements as a result of reverse-engineering the implementation.

Figure 1.2 **Design vs. Implementation Model Elements**



<<Cog>>
FridgeModel
<<output>> + fridgeTemperature : Temperature
<<output>> + doorSensor : CSTypeInt

FridgeSimulation Class—a design model element



FridgeSimulation COG—an implementation model element

Design model elements help you understand the system at an abstract level by hiding many implementation details. They can provide a high-level understanding of a system. But because they do present a “1,000 foot” view, they also show only a partial truth of the system: a lot of details are simply missing from the model. As a result, such elements need to go through a transformation into implementation elements before they can be used in a system.

## 1.4.2 Implementation Model Elements

*Implementation model elements* include COGs, FSMs, ATCs, DFCs, STCs, etc. Design model elements in a work unit are displayed in the **Current UML Model** and **Opened UML Models** tabs; the structure of a work unit's implementation model element, if any, is displayed in the **Definition** tab. Unlike design model elements, implementation model elements are directly usable in a system to provide specific services and behavior. These elements are created based on their design counterparts in the initial part of implementation. Once a skeleton is created, more details are added. It is common not to have any design counterpart for some of the details of an implementation.

# Chapter 2

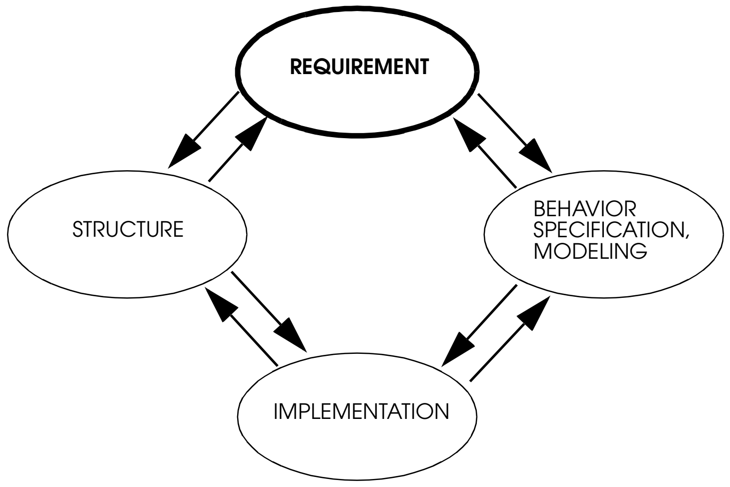
# Development Process

The process by which software is developed differs from organization to organization and potentially from project to project. Nevertheless, there are certain tasks, or stages of software evolution, that must be accounted for in any project of reasonable complexity. Some of these tasks include requirements analysis, structural design, behavioral design, and implementation. These stages may occur just once in a strict order, or they may—with increasing likelihood—iterate from one to another and back again.

*Constellation* is designed to facilitate your development process by recognizing each of these stages and providing specific tools each step of the way. It is also designed to be flexible: it enforces no particular style of development or order of operations, and if a particular stage of the process is not useful for your project, you may leave it out.

This chapter provides a general description of some of the stages through which your development process may progress and the tools that you will find useful at each stage. Chapter 3 through Chapter 6 describe individual features of *Constellation* in detail. Finally, Chapter 7, "Connecting Design and Implementation," closes the loop by showing you in detail how to leverage those features to realize the process described in this chapter.

## 2.1 Requirements Analysis



Requirements analysis is the task of determining what your finished program must *do*: the customer needs it must address, the high-level features it must have, and so on. It is usually one of the first things to do when beginning a new project, although you may revisit and refine your requirements for some time into the development cycle.

There are many ways to capture requirements, from documents produced in a word processor or text editor, to UML diagrams, to databases of hierarchically structured requirements maintained by a dedicated requirements management tool. Many projects will leverage a combination of approaches. *Constellation* allows you to mix and match tools and media however you require.

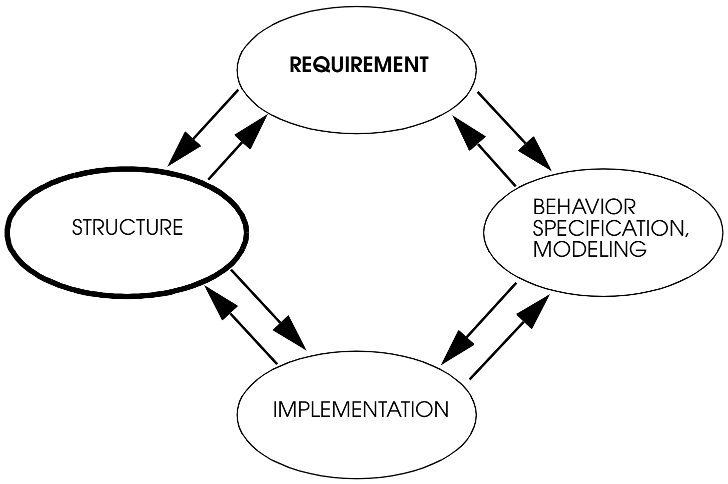
UML use case diagrams are one useful and increasingly popular mechanism for describing the scenarios in which a product will be used and the tasks it must support. In addition to its other diagramming tools, *Constellation* also includes a built-in editor for use case diagrams. Use case diagrams are described in Chapter 4.

For many large projects, collections of diagrams are not enough; the facilities of a powerful dedicated solution are required. One of the most popular requirements management systems is Telelogic DOORS®. It provides hierarchical distributed storage of large numbers of detailed requirements and extensive connections between them. *Constellation* offers tight integration with DOORS. Each reusable unit of software you produce (called a *work unit*) can be associated and synchronized with a DOORS requirements module. For more information on using DOORS with *Constellation*, see the online documentation (use the **Help** menu to select **Online Documentation**.)

In some cases, such a heavyweight requirements analysis may not be necessary or useful. It may be sufficient to describe the software in a word processing document, spreadsheet, or other external file. Or, you may use a custom tool developed in-house. In such cases, the connections between your development environment and your requirements-capture tool are necessarily looser. Nevertheless, it is desirable to maintain the software and the external requirements in parallel.

You can leverage *Constellation*'s work unit infrastructure and version control integration to enable just such situations. By adding your own documents to a work unit's *manifest*, you ensure that they will be stored with your design and implementation and that they will be checked into and out of version control simultaneously and automatically. For more information on work units, see Chapter 3.

## 2.2 Structural Modeling

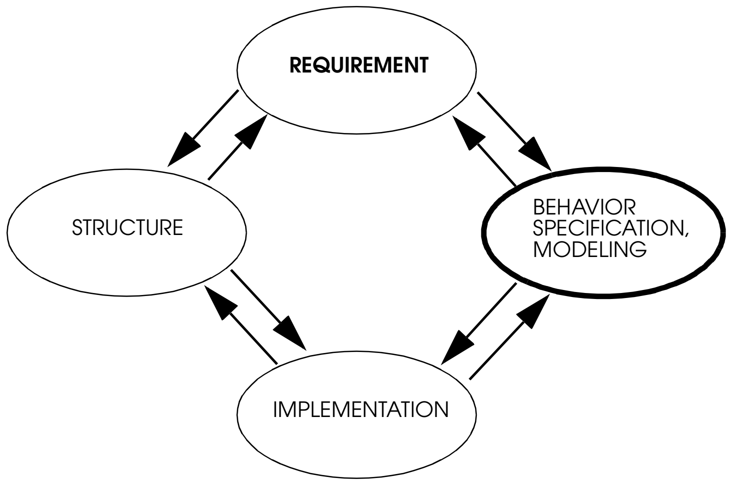


A program's structural design describes what its constituent parts are and how they connect to one another. It does not specify what the interactions among those parts are; that is the role of a behavioral design. Structural and behavioral design together comprise what is frequently called "the design phase" of a software process. Although it is true that some degree of requirements analysis will frequently precede a structural or behavioral design and that some degree of design will likely precede the bulk of the software implementation, the development process is often more fluid in practice than the phrase "design phase" might suggest. The above diagram attempts to capture that reality more accurately by separating structural and behavioral design and explicitly showing many of the interconnections between development tasks.

A structural design captured with a software modeling formalism such as UML is frequently known as a *structural model*. Structural models in UML are typically depicted with *class diagrams*, one of the several diagram types that *Constellation* supports. A class diagram appears as a directed graph in which the nodes represent types—classes and interfaces—and the lines show relationships among those types. Several kinds of relationships are supported, including aggregation, generalization, dependency, and so forth. For more information about structural model elements and class diagrams, see Chapter 5.

When many people start modeling the structure of their programs, they are unsure of the degree of detail that is appropriate to display in a class diagram. When trying to decide whether more or less detail is appropriate, always keep in mind the purpose of class diagrams: to define the identity and features of types and the relationships between them. Attempting to stretch the expressiveness of the medium beyond this purpose tends to result in diagrams that are prohibitively complex and difficult to understand. A structural model should capture enough detail to enable the reader to understand the problem and its proposed solution; it should not try to specify every implementation detail.

## 2.3 Behavior Modeling



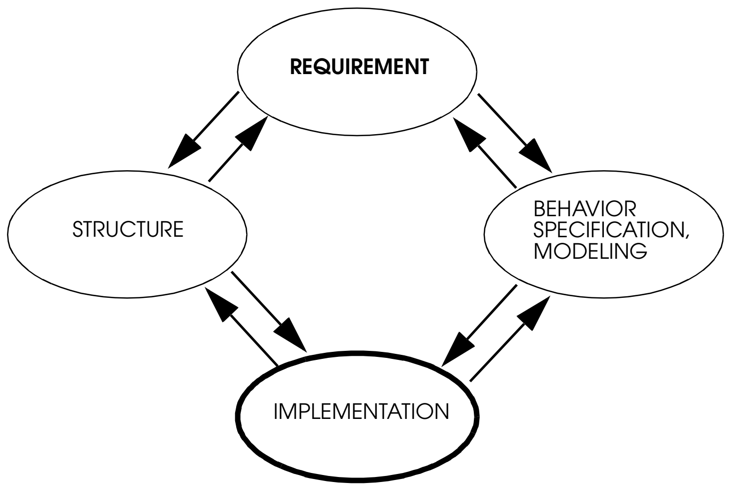
A behavioral design (or, when captured by a modeling formalism such as UML, a *behavioral model*) specifies the interactions among participants in your system. Unlike structural models, which capture the *static relationships* between types, behavioral models capture the *dynamic communication* between instances.

Behavioral models are displayed graphically in different ways, depending on the semantics and constraints of the interaction being modeled. Any interaction may be represented by a sequence diagram. In such a diagram, each role in the interaction (for example, the drive shaft, wheels, and cruise control in an automotive application) appears as a vertical column. A line drawn between two columns represents a message being passed; messages occurring later in time appear below messages occurring earlier in time. See Chapter 6 for more information on sequence diagrams.

Some interactions have more rigorous semantics, however. A state machine, for instance, is one type of behavioral model that specifies not only an order of operations, but explicitly shows which operations may occur between which participants and under what circumstances. State machines are represented by state machine diagrams, which are described in the *Constellation User's Manual*.

*Constellation* supports both sequence and state machine diagrams. And because of the additional expressiveness available in state machine diagrams, *Constellation* state machines are in fact executable implementation; they can govern the interactions among other components and can form the foundation of a deployable application.

## 2.4 Implementation



At some point, you will be ready to begin building your executable application. Typically, you will have completed some degree of requirements analysis and structural and behavioral design prior to that point. However, new issues are likely to come up during implementation that necessitate reconsideration of those requirements and models. *Constellation* does not restrict you to a linear process but facilitates iterative motion among your development stages and tasks. This chapter describes the concepts involved in implementing your software based on your previous work; specifics and mechanics are given in Chapter 7.

*Constellation* offers tools for executing both behavioral and structural models. As noted in Section 2.3, *Constellation* state machines are fully executable and support hierarchical nesting of software components. State machines are described in Chapter 8 in the Constellation User's Manual.

Executable structural models in *Constellation* take the form of component diagrams. Class models are insufficient for this task because they leave crucial information unspecified. For instance, although they can show that classes A and B are associated and that B has an Execute() method, they cannot specify that A invokes Execute() on B, nor the order in which calls are made in with respect to other method calls that might take place. *Constellation* component diagrams (also called Composite Object Group, or COG, diagrams) offer a richer set of semantics specifically for indicating provision and utilization of methods, data flow, and other important information. COGs are described in Chapter 7 in the Constellation User's Manual.

The additional semantics in component diagrams make them somewhat unsuited for the early stages of requirements analysis and software design. You may want to document that a role needs to be fulfilled and what relationships that role's type has to other hypothetical types without committing to a particular kind of implementation. How do you make the transition from class models to executable components? Once the transition has been made, how can you compare the final product with the initial design?

To solve this problem, *Constellation* offers a simple wizard interface to create a component from any class. You can synchronize the contents of the class and the component to the extent you choose. Afterwards, the connections between the class and the component will be remembered: you can immediately jump from any class to the component that implements it or from a component to the class that documents its structural design. For more information on using this tool, refer to Chapter 7.

If your initial design and current implementation start to grow apart, you may wish to resynchronize them with one another to indicate that the implementation changes have been accepted into the component's specification and to provide skeleton implementation for new features from the structural design. The same process described in Chapter 7 that created the component in the first place will also guide you in merging your recent changes.

# Chapter 3

# Work Units and Diagrams

## 3.1 What are Work Units?

*Constellation* bundles definitions, their support files, and related UML content together in groups of files called *work units.* A *work unit* contains *diagrams*, which contain depictions of *model elements*.

❑ A *diagram* is a visual depiction of a group of model elements. Each type of diagram helps you visualize a different aspect of your system. Some types of diagrams, such as COGs, encapsulate software implementation. A work unit may contain at most a single implementation diagram, or it may contain no implementation at all.

❑ A *model element* describes a single item in your system. Different kinds of model elements are used in each type of diagram. For instance, use case diagrams may contain actors and use cases; class diagrams may contain classes, interfaces, and packages; sequence diagrams may contain actors and classifier roles; implementation diagrams may contain either a COG or an FSM.

See Section 1.3 for more information on diagrams and model elements.

## 3.2 Software Reuse

Work units are the fundamental units of software reuse in *Constellation*. To facilitate reuse, model elements and diagrams in one work unit can contain references to model elements in another work unit. It is possible to refer to a model element in a work unit that is read-only; using a work unit does not require modifying it.

Examples of dependencies between work units include:

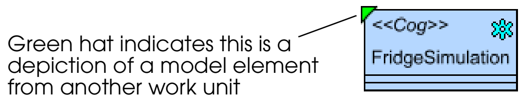
- ❑ A model element in one work unit can have a base class or derived class in another work unit.
- ❑ A diagram can include a depiction of a model element from another work unit.
- ❑ A COG or other implementation diagram can call methods and access data provided by an implementation located elsewhere.

Each *Constellation* window displays the contents of one work unit at a time; this work unit is known as the *current work unit*. By default, new model elements are created in the current work unit. A model element always exists in exactly one work unit at a time; that work unit is said to be its *home* (base) work unit. Once a model element is created in a work unit, it cannot be moved or copied to another work unit.

A model element or diagram in a work unit can refer to a model element from another work unit. In this case, the model element is said to be *foreign* with respect to the current (referring) work unit.

When looking at a diagram, depictions of *foreign* model elements have a small triangular green hat[1], as seen in Figure 3.1. Place the cursor on the hat to see a ToolTip indicating the work unit in which the depicted model element exists.

Figure 3.1 Foreign Model Element

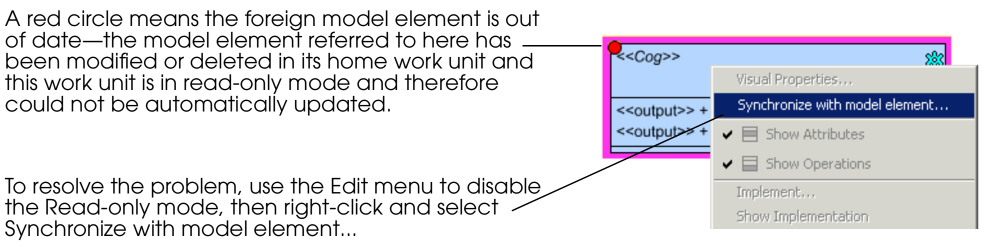

Green hat indicates this is a depiction of a model element from another work unit
<<Cog>>
FridgeSimulation

If a foreign representation of a model element is inconsistent with the current state of the model element itself, the green triangle becomes a red circle, as seen in Figure 3.2. For example, if you delete a model element from its home work unit, any depictions of the model element in diagrams of editable work units will also be removed. But if the

1. The implementation diagrams of FSMs and COGs do not use the foreign work-unit indicator. All contained items within implementation diagrams are foreign.

deleted element appears in a work unit that is in read-only mode or is not fully loaded, the element is marked with a red circle to show that it is out of sync.

Figure 3.2 **Foreign Model Element with an Error**


A red circle means the foreign model element is out of date—the model element referred to here has been modified or deleted in its home work unit and this work unit is in read-only mode and therefore could not be automatically updated.
To resolve the problem, use the Edit menu to disable the Read-only mode, then right-click and select Synchronize with model element...
<<Cog>>
<<output>> +
<<output>> +
Visual Properties...
Synchronize with model element...
Show Attributes
Show Operations
Implement...
Show Implementation

## 3.3 Physical Structure

Think of a work unit as a document made up of several physical files. Although a work unit contains several files, it is designed to be worked with as a cohesive entity—you will open it, save it, and check it into and out of configuration management systems atomically.

Each work unit, including the files that belong to it, exists in its own directory within a *Constellation* repository. If the work unit contains implementation, it must exist in a sub-folder appropriate for its type of implementation (COG, ATC, etc.). Work units without implementation may exist anywhere in a repository.

For more information on *Constellation* repositories and their structure, see Chapter 5 in the Constellation User's Manual.

What files are created for a work unit?

- ❑ Each work unit has a manifest file (**.mf**) that lists the files that comprise it.
- ❑ Each work unit has an XMI file (**.xmi**), used to store the UML model elements.
- ❑ Each class, use case, and sequence diagram is stored in an SVG-format vector graphics file (**.svg**).
- ❑ Implementation diagrams are stored in formats appropriate for their particular type: ATCs are stored in ***.atc** files, COGs are stored in ***.cog** files, and so on. Primitive components additionally have associated C++ source code files (***.cxx**).

Dependencies between work units are managed by three index files, which exist on a per-repository basis:

- ❏ A *model element index* (**model_index.xml**) maps each model element's Universally Unique Identifier (UUID)—a long string of numbers and other characters—to the UUID of the model element's home work unit.
- ❏ A *work unit index* (**work_unit_index.xml**) maps work unit UUIDs to the repository-relative paths of the work units in that repository.
- ❏ Finally, an *implementation index* (**REPOSITORY_INDEX**) maps the names of implemented work units to the repository-relative paths of implementation diagram files. (Since the names of implementation diagrams serve as programmatic identifiers, they must be unique within a repository.) For more information, see Section 5.2.1 in the Constellation User's Manual.

## 3.4 Work Unit Mechanics

### 3.4.1 Creating Work Units

A new, empty work unit can be created anywhere in a repository; a new work unit that will contain an implementation diagram must be created in the repository sub-folder appropriate for its implementation type (**/cogs**, **/atcs**, etc.).

For example, to create a work unit that includes a COG implementation diagram, select the Repository tab, right-click the appropriate **cogs** folder and select **New Work Unit...** from the pop-up menu. You will see a new folder; inside it will be all of the files initially belonging to the new work unit. Then you can add class, use case, and sequence diagrams to it by using the buttons in Figure 3.5.

To create a work unit that will not contain an implementation diagram, select the Repository tab, right-click any folder and select **New Work Unit (No Component)...** from the pop-up menu.

### 3.4.2 Opening Existing Work Units

To open an existing work unit, select the **Repository** tab and double-click the work unit's **.mf** file. Work units with implementation may also be opened by double-clicking the implementation file (***.cog**, ***.atc**, etc.).

If you open a work unit that contains references to other work units, those other work units will be loaded to the extent necessary to satisfy the references.

Open work units are presented in two parallel ways: in the browser pane on the left and in the workspace on the right.

When one or more work units are open, additional tabs appear at the bottom of the Repository Explorer: the **Current UML Model** tab displays all model elements contained in the current work unit. The **Opened UML Models** tab is similar but displays model elements from all work units currently loaded. For work units with implementation, a **Definition** tab will also be displayed on the left. The Definition Explorer is described in Section 4.5 in the Constellation User's Manual.

As seen in Figure 3.3, the diagrams in a work unit are organized by the role they play in the work unit. There are buttons on the top that display the diagrams according to these four categories:

- **Implementation** implementation diagrams (COGs, FSMs, ATCs, DFCs, STCs, and applications)
- **Requirements** use case diagrams
- **Structural** class diagrams
- **Behavior** sequence diagrams

If a work unit uses more than one type of diagram, buttons for each type will appear at the top of the workspace. Click the buttons to switch views. If no diagrams exist in a work unit (such as for a new, empty work unit), the workspace will be blank and the four buttons will not appear.

When the **Implementation** button is selected (as seen in Figure 3.3), you will see three tabs at the top of the workspace: **Diagram View** (for those implementation types editable in a diagrammatic view), **Form View**, and **Documentation**. These tabs are described in Chapter 4 in the Constellation User's Manual.

When the **Requirements**, **Structure**, or **Behavior** button is selected (as seen in Figure 3.4), you will see one tab for each diagram of the corresponding type in the work unit. For example, Figure 3.4 shows two class diagrams called Overview and Temperature Control; these only appear when the **Structure** button is selected.

Figure 3.3 **Displaying Different Types of Diagrams in a Work Unit**

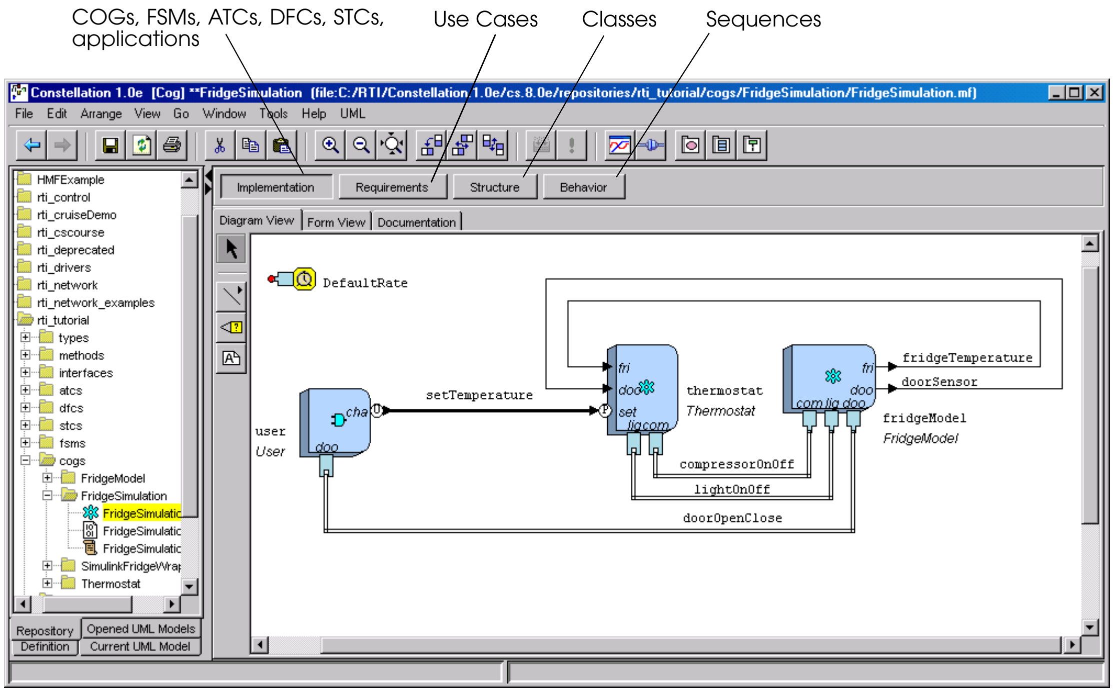
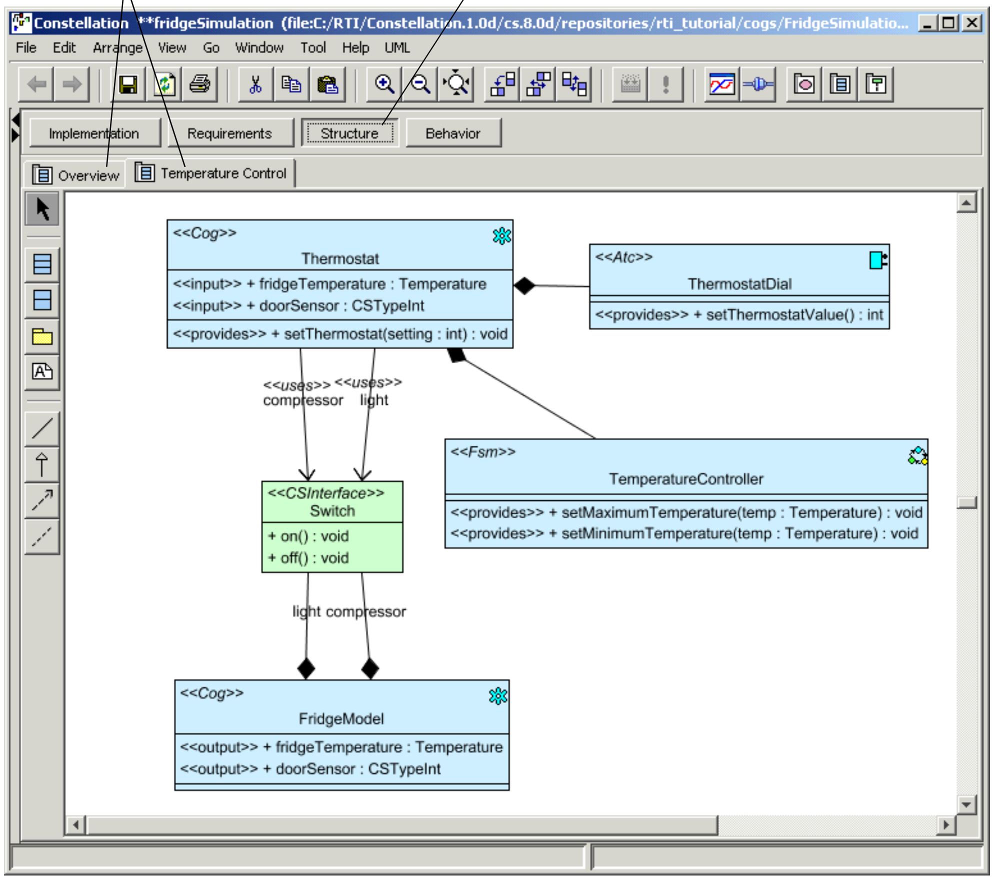


Figure 3.4 **Class Diagram Workspace**

Tabs for each Class Diagram appear when Structure is selected


Constellation **fridgeSimulation (file:C:/RTI/Constellation.1.0d/cs.8.0d/repositories/rti_tutorial/cogs/FridgeSimulatio...
File Edit Arrange View Go Window Tool Help UML
Implementation
Requirements
Structure
Behavior
Overview
Temperature Control
<<Cog>>
Thermostat
<<input>> + fridgeTemperature : Temperature
<<input>> + doorSensor : CSTypeInt
<<provides>> + setThermostat(setting : int) : void
<<Atc>>
ThermostatDial
<<provides>> + setThermostatValue() : int
<<uses>> compressor
<<uses>> light
<<CSInterface>>
Switch
+ on() : void
+ off() : void
<<Fsm>>
TemperatureController
<<provides>> + setMaximumTemperature(temp : Temperature) : void
<<provides>> + setMinimumTemperature(temp : Temperature) : void
light compressor
<<Cog>>
FridgeModel
<<output>> + fridgeTemperature : Temperature
<<output>> + doorSensor : CSTypeInt

3. Work Units

### 3.4.3 Saving Work Units

To save an open work unit, click the **Save** button in the toolbar or choose **Save** from the **File** menu. You will also be prompted to save any open work unit with changes when you quit *Constellation*.

### 3.4.4 Destroying Work Units

To destroy a work unit, right-click its folder in the Repository Explorer and choose **Destroy** from the resulting pop-up menu. Once you confirm your selection, the work unit folder and all files inside will be permanently deleted.

## 3.5 Working with Class, Use Case, and Sequence Diagrams

*Constellation* diagrams can be in one of two modes: Read-only or Edit. The background color of the workspace indicates whether a diagram can be edited (white) or is read-only (yellow). The **Read-only** toggle command is under the **Edit** menu.

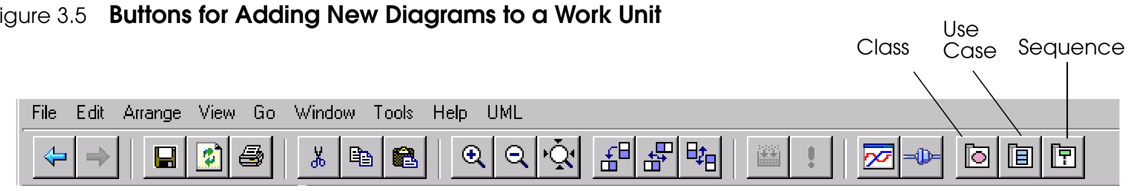
The basic steps for adding, renaming, and deleting diagrams is the same, regardless of the diagram type (class, use case, or sequence).

Exception: you cannot add an implementation diagram to an existing work unit; the implementation must be created when the work unit is created. Similarly, you cannot delete an implementation diagram from a work unit.

**To create and rename a class, use case, or sequence diagram:**

1. Select the appropriate button from the toolbar (see Figure 3.5).

Figure 3.5 **Buttons for Adding New Diagrams to a Work Unit**

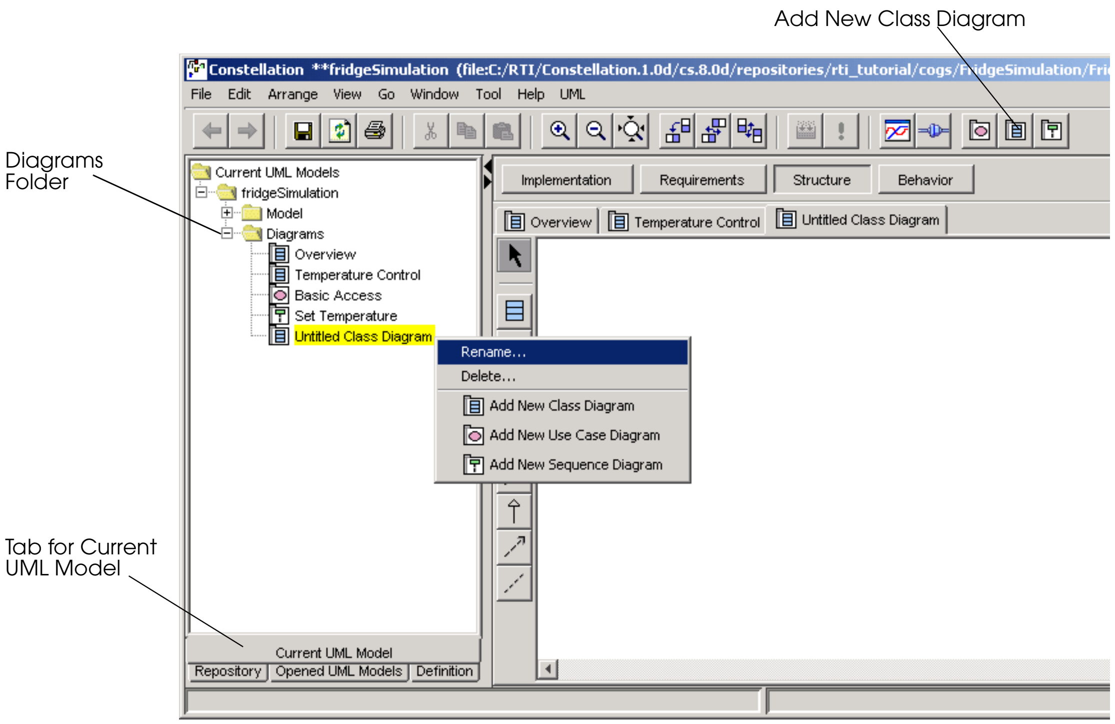


   An empty workspace will be displayed under a new tab labeled **Untitled <type> Diagram** (where <**type**> is Class, Use Case, or Sequence), as seen in Figure 3.6.

2. Select the **Current UML Model** tab.

3. If the **Model** folder at the top of the tree is open (expanded), double-click the folder to close (collapse) it. This will make it easier to see the entries in the **Diagrams** folder in the next step.
4. Expand the **Diagrams** folder to locate the **Untitled <type> Diagram**.
5. Right-click **Untitled <type> Diagram** and select **Rename...** from the pop-up menu, as seen in Figure 3.6.
6. Enter the new name and click **OK**.

Figure 3.6 **Adding and Renaming a Class Diagram**



**To open an existing class, use case, or sequence diagram:**

❑ Recall from Section 3.4.2 that diagrams are organized according to their role in the work unit. Select one of the four buttons on the top to see its related diagrams, as seen in Figure 3.4. For instance, to open a work unit's class diagrams, select the **Structure** button; to open sequence diagrams, select **Behavior**.

**To delete a class, use case, or sequence diagram:**

1. Select the **Current UML Model** tab.
2. If the **Model** folder at the top of the tree is open (expanded), double-click the folder to close (collapse) it. This will make it easier to see the entries in the **Diagrams** folder in the next step.
3. Expand the **Diagrams** folder to locate the diagram..
4. Right-click the diagram and select **Delete...** from the pop-up menu.

## 3.6 Working with Model Elements

### 3.6.1 Creating Model Elements

To create a model element, use the vertical drawing toolbar located to the left of the workspace. Figure 3.7 shows the location of this toolbar and other important tabs and buttons (in this case, for class diagrams).

The buttons on the drawing toolbar depend on the type of diagram selected and are described in their respective chapters:

- ❏ Use Case Diagrams: Section 4.2
- ❏ Class Diagrams: Section 5.2
- ❏ Sequence Diagrams: Section 6.2
- ❏ Implementation Diagrams: Chapter 4 in the Constellation User's Manual

The mechanics of using the drawing buttons are the same, regardless of the diagram type. To create a single model element and insert it in the current diagram, click the button for the desired type, then click again inside the workspace.

To create more than one model element of the same type, double-click the button (making it "sticky"), then click multiple times in the workspace. For instance, to create three interface model elements, double-click the Interface button, then click in the workspace three times. To "un-stick" a button, click the Select tool . The bottom of Figure 3.7 shows how these button looks when sticky.

**Note:** It is possible to create a model element in a work unit without displaying it in any diagram. Do this by right-clicking on a **Model** node in either the **Current UML Model** or **Opened UML Models** tabs of the repository browser and choose one of the **Add** commands from the resulting pop-up menu. In practice, you will usually immediately insert a depiction of a model element into a diagram.

### 3.6.2 Reusing Model Elements

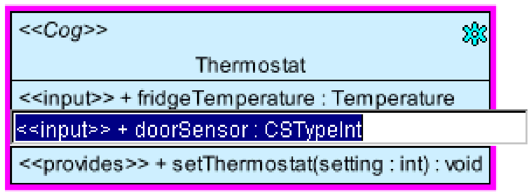
To insert a depiction of an existing model element in a diagram, select the model element from the **Current UML Model** or **Opened UML Models** tabs, then drag and drop the model element into the workspace. Or, you can right-click the model element and choose **Insert into Current Diagram** from the pop-up menu.

### 3.6.3 Changing a Model Element's Properties

You can change a model element's properties by using its Properties window (such as the one in Figure 3.8). To access a particular model element's properties, either double-click the model element in the workspace, or right-click it and select **Model Element Properties...** from the pop-up menu.

Additionally, most individual model element properties—including the model element name, the names of class attributes and operations, stereotypes, and so on—may be edited in place by double-clicking the appropriate text. A small text field will appear, as seen in Figure 3.9; press <**Enter**> to commit your changes or <**Escape**> to revert them.

Figure 3.9 **Editing a Specific Model Element Property**



<<Cog>>
Thermostat
<<input>> + fridgeTemperature : Temperature
<<input>> + doorSensor : CSTypeInt
<<provides>> + setThermostat(setting : int) : void

### 3.6.4 Changing How a Model Element Appears in a Diagram

You can change how any depiction of a model element appears in a diagram. Right-click the element and select **Visual Properties...** from the pop-up menu. Figure 3.10 shows the resulting dialog.

For information on the icon that may appear in the upper right corner, see Section 3.6.5.

Figure 3.7 **Working with Diagrams**

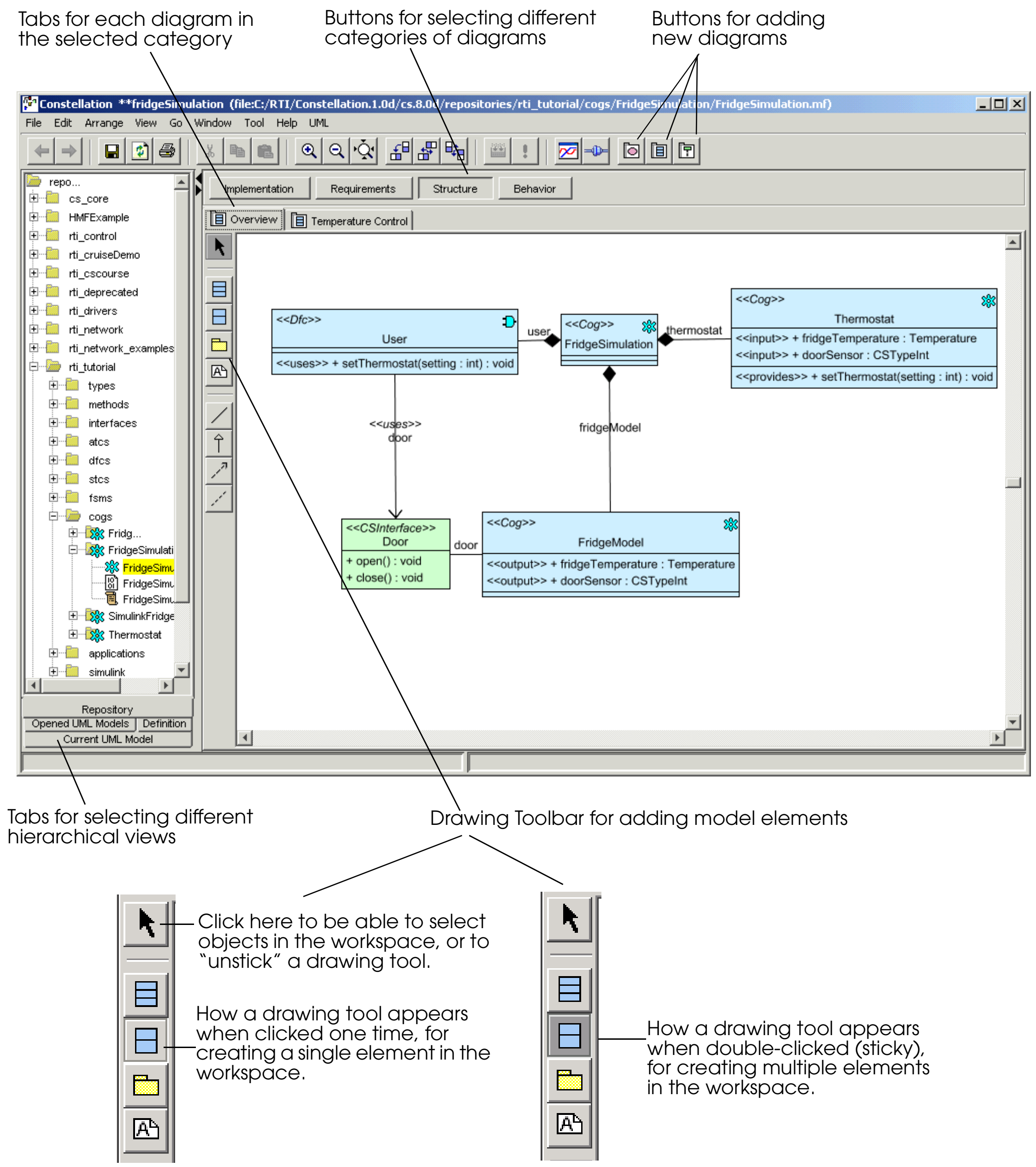
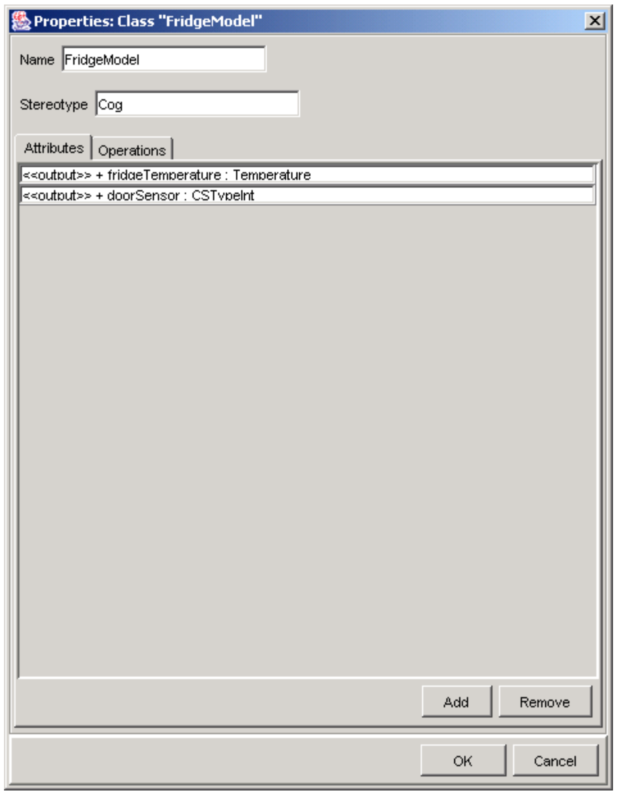

Tabs for each diagram in the selected category
Buttons for selecting different categories of diagrams
Buttons for adding new diagrams
Constellation **fridgeSimulation (file:C:/RTI/Constellation.1.0d/cs.8.0d/repositories/rti_tutorial/cogs/FridgeSimulation/FridgeSimulation.mf)
File Edit Arrange View Go Window Tool Help UML
Implementation
Requirements
Structure
Behavior
Overview
Temperature Control
repo...
cs_core
HMFExample
rti_control
rti_cruiseDemo
rti_cscourse
rti_deprecated
rti_drivers
rti_network
rti_network_examples
rti_tutorial
types
methods
interfaces
atcs
dfcs
stcs
fsms
cogs
Fridg...
FridgeSimulati
FridgeSimu
FridgeSimu
FridgeSimu
SimulinkFridge
Thermostat
applications
simulink
Repository
Opened UML Models
Definition
Current UML Model
<<Dfc>>
User
<<uses>> + setThermostat(setting : int) : void
user
<<Cog>>
FridgeSimulation
thermostat
<<Cog>>
Thermostat
<<input>> + fridgeTemperature : Temperature
<<input>> + doorSensor : CSTypeInt
<<provides>> + setThermostat(setting : int) : void
<<uses>>
door
fridgeModel
<<CSInterface>>
Door
+ open() : void
+ close() : void
door
<<Cog>>
FridgeModel
<<output>> + fridgeTemperature : Temperature
<<output>> + doorSensor : CSTypeInt
Tabs for selecting different hierarchical views
Drawing Toolbar for adding model elements
Click here to be able to select objects in the workspace, or to "unstick" a drawing tool.
How a drawing tool appears when clicked one time, for creating a single element in the workspace.
How a drawing tool appears when double-clicked (sticky), for creating multiple elements in the workspace.

Figure 3.8 **Editing Model Element Properties**

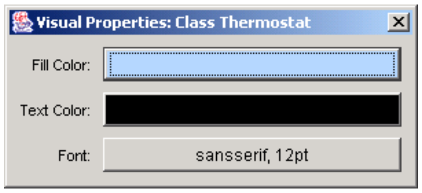


Name FridgeModel

Stereotype Cog

<<output>> + fridgeTemperature : Temperature
<<output>> + doorSensor : CSTypeInt

Figure 3.10 **Editing Visual Properties**



Fill Color:

Text Color:

Font: sansserif, 12pt

*Click one of the three buttons to change the fill color, text color, or font. Note that this only changes how the depiction of the model element appears in the current diagram.*

### 3.6.5 Adding Custom Stereotype Icons

If you use a stereotype that matches one of *Constellation*'s component types—ATC, DFC, STC, COG, FSM, or APP—a matching icon appears in the upper right corner, such as the COG icon seen in Figure 3.9.

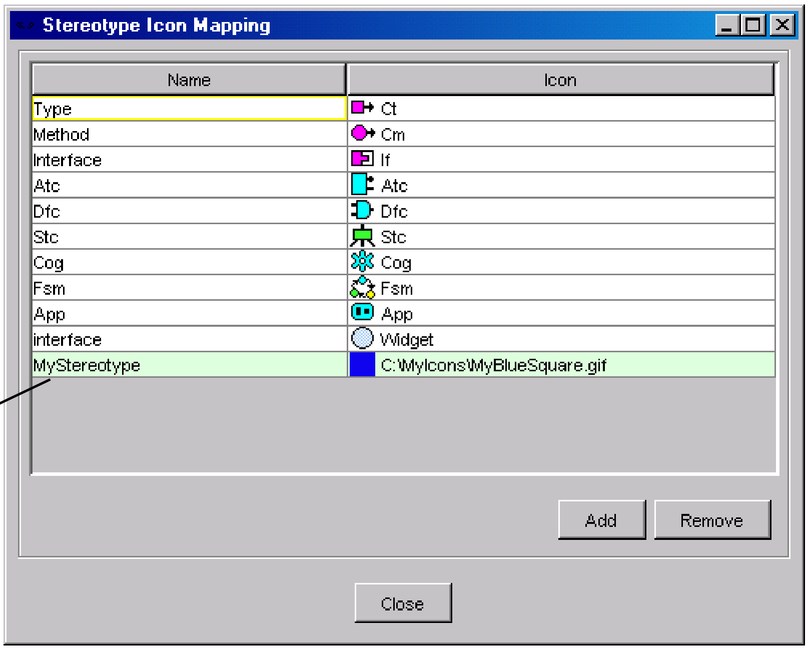
You can also create a mapping between your own custom stereotype and any icon (in .gif format) that you choose. To do so, select the **Stereotype icons...** command under the **UML** menu. (The **UML** menu only appears on the toolbar when you have a UML diagram open.) Figure 3.11 shows the dialog for adding and removing your own stereotype icon mappings, as well as how the icon is used in a model element.

The icons used for predefined *Constellation* components cannot be changed or removed; they appear in white in the table to indicate they are read-only.

To add a mapping, click **Add**. Double-click the Name cell to enter the stereotype name. (Note: stereotypes are case-sensitive.) Then double-click the Icon cell and click the browse button ("...") to locate a .gif file to use as the icon for the stereotype. User entries in the table have a green background to indicate they can be changed, as seen in Figure 3.11.

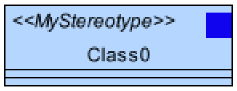
To edit a user-created mapping, double-click the stereotype name or icon.

Figure 3.11 **Mapping Stereotype Icons**



| Name | Icon |
|---|---|
| Type | Ct |
| Method | Cm |
| Interface | If |
| Atc | Atc |
| Dfc | Dfc |
| Stc | Stc |
| Cog | Cog |
| Fsm | Fsm |
| App | App |
| interface | Widget |
| MyStereotype | C:\MyIcons\MyBlueSquare.gif |

Only green rows can be changed.



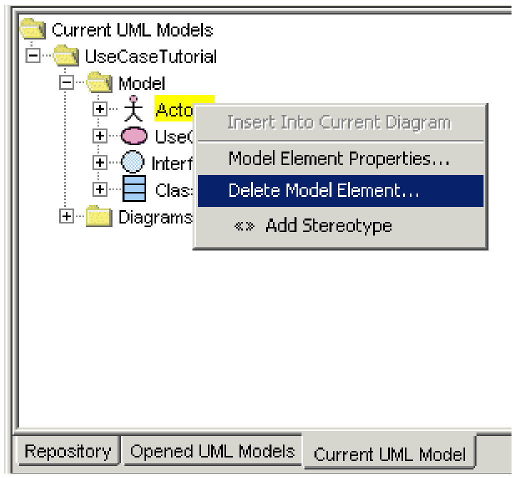
New icon appears when you use the mapped stereotype.

### 3.6.6 Deleting Model Elements

To delete a model element, select the model element (from the **Current UML Model** or **Opened UML Models** tabs or from a diagram's workspace), then right-click and select **Delete Model Element...** as seen in Figure 3.12.

**Note:** using the <**Delete**> key is *not* the same as **Delete Model Element...**. The <**Delete**> key is like an eraser, it takes the depiction of the model element out of the workspace, but does not remove it from the list of defined model elements.

Figure 3.12 **Deleting a Model Element**



## 3.7 Viewing Exported Diagrams

*Constellation* saves UML diagrams in SVG format, which is XML-based. There is no standard format for UML diagrams recognized by the OMG, but SVG is a strong candidate. Free browser plug-ins are available (from Adobe®), so *Constellation* UML diagrams are easily viewable in any web browser. Diagram files are named ***<model_name>_<diagram_name>.svg***.

Chapter 4

# Use Case Diagrams

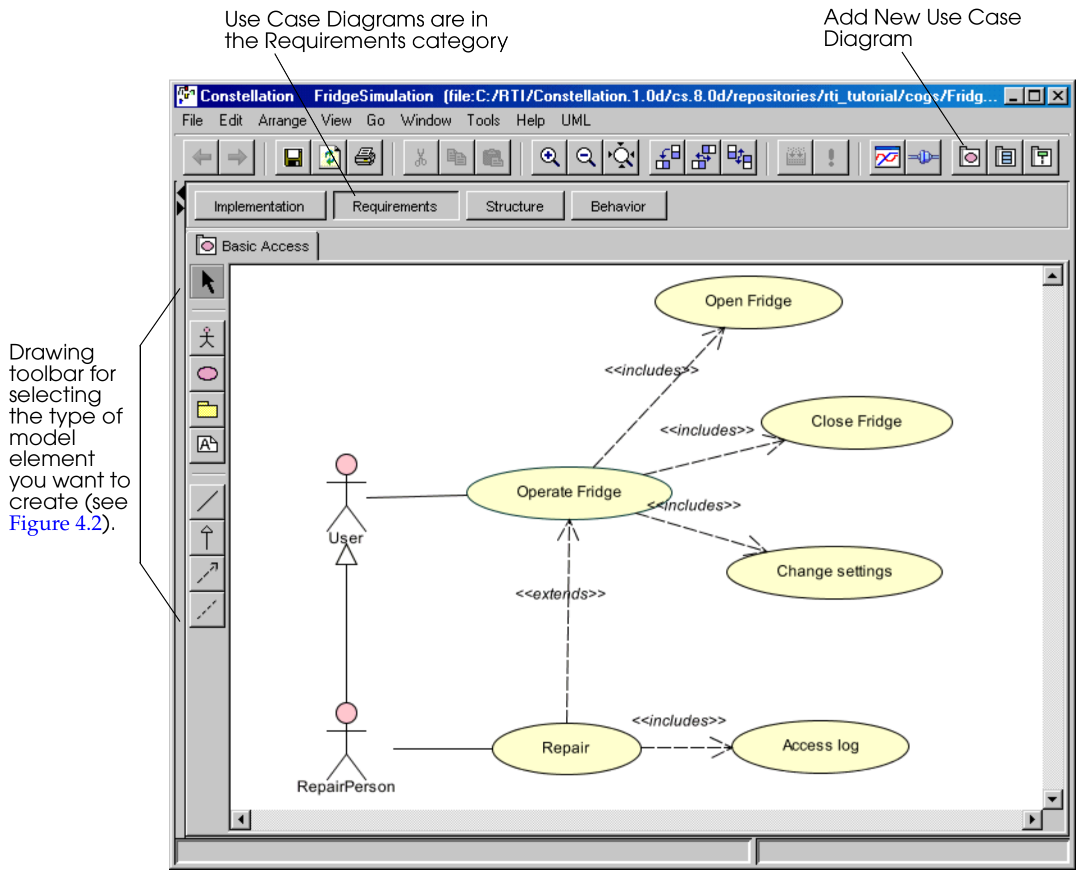
## 4.1 Concepts

A use case diagram captures the scenarios in which your system will be used and the individuals involved in those scenarios. It is generally used during the requirements phase of a project to model, at a high level, what the system will have to accomplish when it is complete.

This chapter describes the model elements that can be used in use case diagrams and how to add, delete and modify them in *Constellation*. Figure 4.1 shows a sample use case diagram. This chapter also includes a tutorial to show you how to create this diagram.

The ordering of the "steps" in a model is important in the finished system, but it is not captured by a use case diagram. Use case diagrams specify only which actions the system should support; they do not specify how they should be supported, nor do they show in what order. Order of operations is captured by sequence diagrams (see Chapter 6).

Figure 4.1 **Use Case Diagram Workspace**


Use Case Diagrams are in the Requirements category
Add New Use Case Diagram
Drawing toolbar for selecting the type of model element you want to create (see Figure 4.2).
Constellation FridgeSimulation (file:C:/RTI/Constellation.1.0d/cs.8.0d/repositories/rti_tutorial/cogs/Fridg...
File Edit Arrange View Go Window Tools Help UML
Implementation
Requirements
Structure
Behavior
Basic Access
Open Fridge
<<includes>>
Close Fridge
<<includes>>
Operate Fridge
<<includes>>
Change settings
User
<<extends>>
<<includes>>
Repair
Access log
RepairPerson

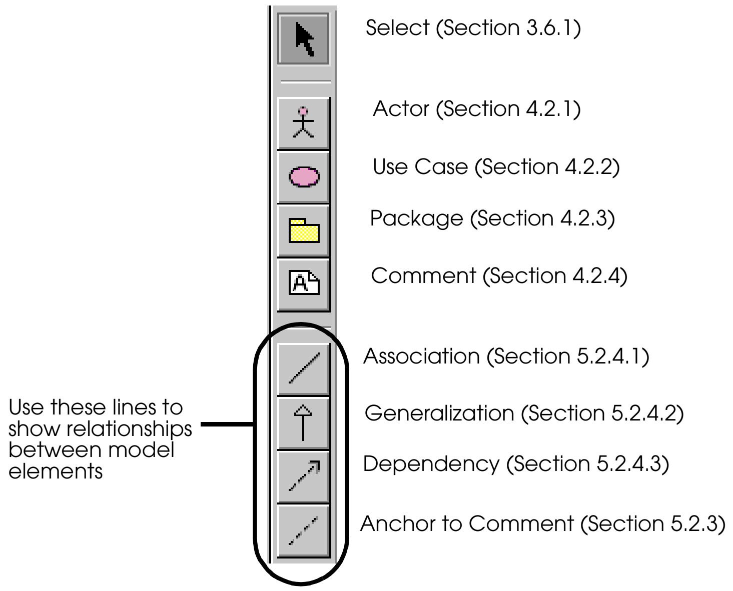
*This use case diagram is provided in the rti_tutorial repository's FridgeSimulation COG.*

## 4.2 Model Elements

Figure 4.2 shows the types of model elements you can include in use case diagrams.

For general information on how to work with diagrams and model elements, see Section 3.5 and Section 3.6, respectively.

Figure 4.2 **Drawing Toolbar for Use Case Diagrams**


Select (Section 3.6.1)
Actor (Section 4.2.1)
Use Case (Section 4.2.2)
Package (Section 4.2.3)
Comment (Section 4.2.4)
Association (Section 5.2.4.1)
Generalization (Section 5.2.4.2)
Dependency (Section 5.2.4.3)
Anchor to Comment (Section 5.2.3)
Use these lines to show relationships between model elements

*To insert one of these model elements, click the button to select, then click inside the workspace. To insert more than one, double-click the button, then click multiple times in the workspace.*

### 4.2.1 Actors

An *actor* is any entity (human or machine) that is outside the scope of your system but that interacts with it in a relevant way. It is conventionally depicted as a stick figure. In Figure 4.1, a user operating a refrigerator is modelled by an actor, and so is the repair person. If you were designing a car, the driver would likely be an actor. If you were designing only the car's transmission, the engine might be an actor in one or more use case diagrams.

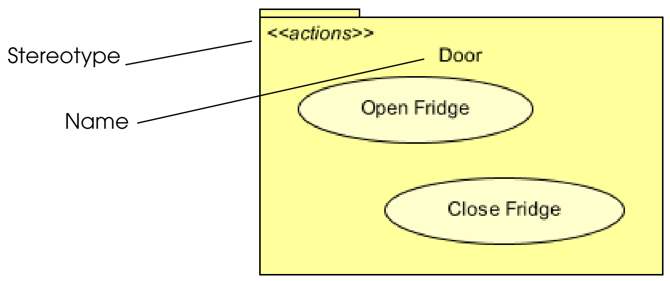
### 4.2.2 Use Cases

A *use case* model element describes a scenario in which actors are engaged. It is conventionally depicted as an oval with a name in the center. Because use case modelling is done at a very high level—when you are deciding *what* your system needs to do, not *how* it will do it—a use case typically does not correspond to any particular piece of executable code that you might produce later, nor does its name correspond to any programmatic identifier. It describes the context in which a system will be used, not that system's structure or complex behavior.

### 4.2.3 Packages

Packages provide a way to group related model elements together. A package can contain use cases, actors, relationships (the lines that connect uses cases and actors, see Section 4.2.5), and even other packages. You can use nested packages to show greater degrees of similarity between model elements. Like other model elements, a package can have a name and a stereotype, as seen in Figure 4.3.

Figure 4.3 **Package Diagram**



### 4.2.4 Comments

A *comment* model element allows you to attach simple documentation to any other model element in a diagram. Comments are typically used to describe certain characteristics that are not otherwise apparent.

Select the Comment tool from the drawing toolbar and click near the model element you want to comment. Double-click the text of the comment and type whatever documentation you desire. When you are finished typing, click outside the text box to commit your change (or press <**Escape**> to revert your change). Select the Anchor to

Comment tool and use it to draw a line between your new comment and the model element to which it refers, as seen in Figure 4.4.

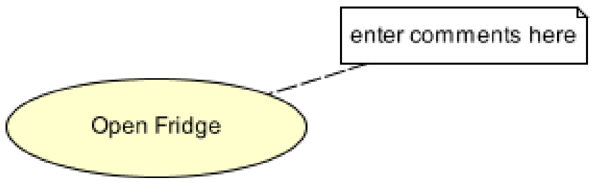


Figure 4.4 **Comment**

### 4.2.5 Relationships

Different types of lines are used to visually define relationships between other model elements. These relationship lines are also considered model elements. The types of relationships you can show between model elements in a use case diagram include:

- ❏ Generalizations (Section 4.2.5.2)
- ❏ Associations (Section 4.2.5.1)
- ❏ Dependencies (Section 4.2.5.3)

To draw a relationship between two model elements, select the desired button from the drawing toolbar, then in the workspace simply drag the line from one model element to the other. The direction of the line matters and is discussed in each of the following sections. For practice, follow along with the tutorial in Section 4.3.

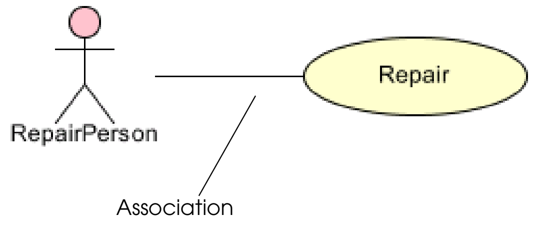
#### 4.2.5.1 Associations

An *association* shows a relationship in which two model elements know about or are affiliated with each other. In this type of relationship, the model elements have explicit knowledge of each other. In Figure 4.5, the RepairPerson actor is associated with the Repair use case. Contrast this with the *dependency* relationship (Section 4.2.5.3), in which one model element uses another. An association is a stronger type of relationship than a dependency.

#### 4.2.5.2 Generalizations

A *generalization* shows a relationship between two model elements in which one is a base model element and the other is a derived model element. In a use case diagram, the derived model element inherits behavior from the base. Another way to think of generalizations is that the derived element's behavior is a subset of the base element's behav-

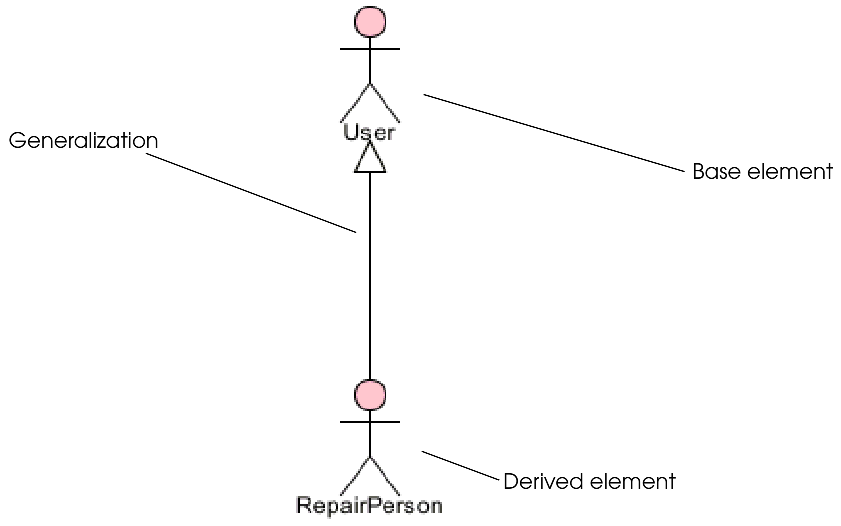
Figure 4.5 **Association**



ior. In our refrigerator example, the repair person inherits the behavior of a user—that is, a repair person does all the same things as a normal user, plus other actions.

The line for a generalization has a hollow-tipped arrow pointing from the derived element towards the base element. In Figure 4.6, a generalization is drawn from the RepairPerson actor towards the User actor.

Figure 4.6 **Generalization**

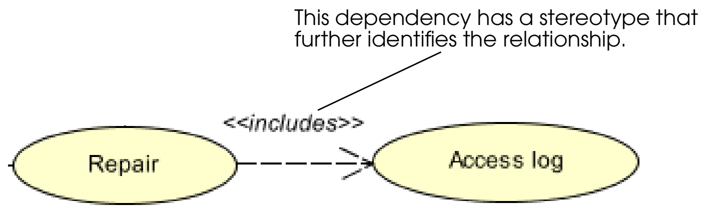


### 4.2.5.3 Dependencies

A *dependency* shows that one model element uses—is dependent on—another. This is a weaker type of relationship than an association (Section 4.2.5.1). For example, in Figure 4.7 we see that the repair process depends on accessing a log. In a dependency, model element A uses model element B, or relies on B's existence. However, A may not have explicit knowledge of B's inner workings.

A dependency appears as a dashed line with an arrowhead pointing towards the model element that is depended on, as seen in Figure 4.7, where the repair process depends on accessing a log. Figure 4.1 shows several more examples of dependencies.

Figure 4.7 **Dependency**


This dependency has a stereotype that further identifies the relationship.
<<includes>>
Repair
Access log

Dependencies often use stereotypes to show two specific types of dependency relationships known as *extends* and *includes*:

- Extends

  An *extends* relationship indicates that one use case (the *extending use case*) contains additional semantics that augment those of another use case (the *extended or base use case*). For example, in Figure 4.1 we see that the use case Repair *extends* the use case Operate Fridge—repairing a refrigerator is an extension of operating it.

  The extending use case is unlike a subtype because it does not include the semantics of the extended use case as part of itself; it provides only the additional information.

  To draw an *extends* relationship, use a *dependency* and add a stereotype called "***<<extends>>***", as seen in Figure 4.1. The dependency should be drawn from the extending use case to the extended use case.

- Includes

  An *includes* relationship indicates that one use case depends on another being carried out as a part of its own behavior. It is somewhat analogous to aggregation. It may denote a high-level task that contains a number of sub-tasks. For example, in Figure 4.1 we see that the use case Operate Fridge *includes* three other uses cases: Open Fridge, Close Fridge, and Change Settings—using a refrigerator includes opening it, closing it, and changing its settings.

  To draw an *includes* relationship, use a *dependency* and add a stereotype called "***<<includes>>***", as seen in Figure 4.7. The dependency should point to the included use case.

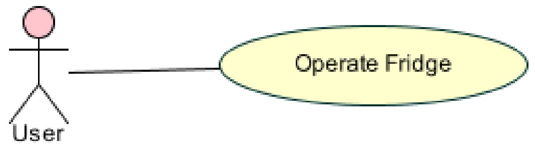
## 4.3 Tutorial

This section describes how to create the example use case diagram that is seen in Figure 4.1 and provided in the **rti_tutorial** repository (FridgeSimulation.cog). This will allow us to explore both the thought process and the mechanics of using the tools *Constellation* provides for creating use case diagrams.

### 4.3.1 Creating Model Elements

Let's start with a very basic use case—we have a User who needs to operate a refrigerator. The User will be represented by an Actor model element. The use case thus far is simply "Operate Fridge."

Figure 4.8 **A Simple Use Case Diagram**



**To create the diagram in Figure 4.8:**

1. *Start in one of your own repositories.* (For information on creating a repository, see the *Constellation User's Manual*.) The goal of this tutorial is to create a diagram that looks like the one in the **rti_tutorial** repository; do not write your changes in **rti_tutorial**.
2. Create a new use case diagram using the **Add New Use Case Diagram** button in the toolbar that stretches across the top of the application windows (seen in Figure 4.1).
3. Click the use case button in the vertical drawing toolbar along the left side of the diagram just created (seen in Figure 4.2).
4. Click the diagram workspace to create a new use case in the work unit and place a depiction of it in the diagram.
5. Double-click the use case's name to open an editing field; change the name to "Operate Fridge" and press <**Return**>.
6. Now choose the actor tool in the drawing toolbar and create a new actor in the diagram in the same way.

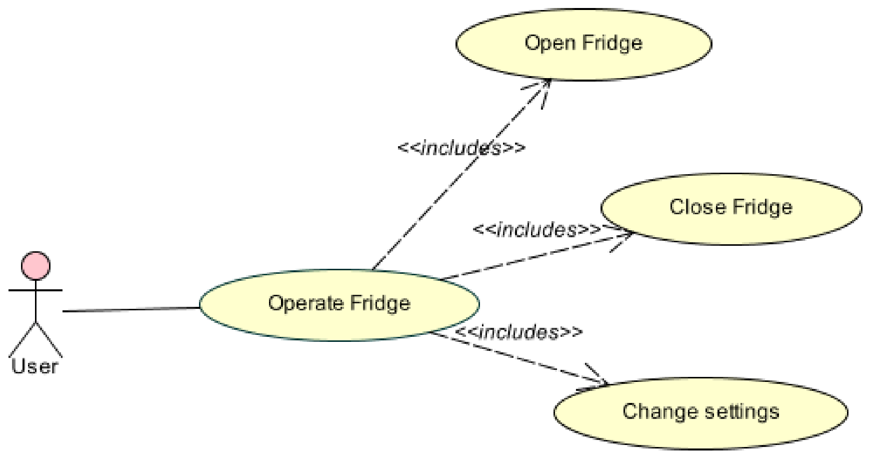
7. Change the actor's name to "User."
8. Show that the actor and use case are associated with each other by selecting the association tool and dragging from one to the other.

### 4.3.2 Subdividing a Use Case

Now that our basic use case has been created, we need to refine it to present a clearer idea of just what the user must do when operating the refrigerator. We will do this by breaking down the use case into several use cases of finer granularity with the *includes* relationship. An *included* use case is one that does not stand independently, it must be part of another, broader use case.

Part of operating a refrigerator *includes* opening and closing the door, as well as changing its settings. So we need to create three additional use cases and connect them to the original Operate Fridge use case using dependencies with <<*includes*>> stereotypes, as seen in Figure 4.9.

Figure 4.9 **Example with "Includes" Dependencies**



**To build upon** Figure 4.8 **to create the diagram in** Figure 4.9:

1. Double-click the use case button in the drawing toolbar to make it sticky.
2. Click the diagram workspace three times to create three new use cases in the work unit and place depictions of them in the diagram.
3. Click the Select button in the drawing toolbar (the dark arrow on the top) to "unstick" the use case tool.

4. Change the new use cases' names (Open Fridge, Close Fridge, Change Settings).
5. Double-click the dependency button.
6. Draw dependency lines *from* Operate Fridge *to* each of the three new use cases.
7. Click the Select button in the drawing toolbar.
8. Double-click each new dependency and add the stereotype <<*includes*>>.

### 4.3.3 Extending a Use Case

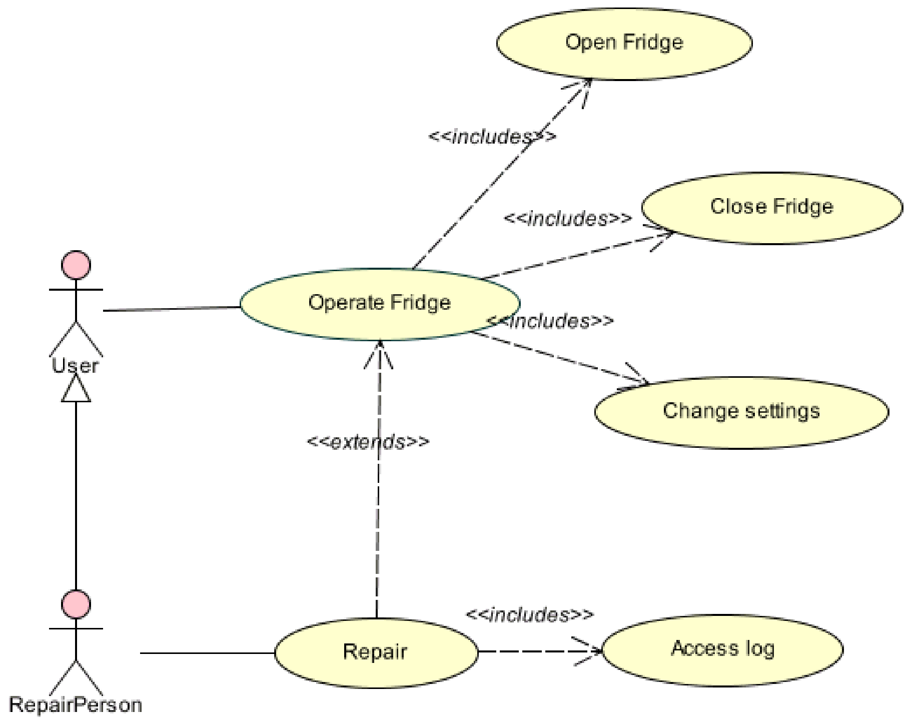
Now let's add another actor to the diagram to represent a repair person. A repair person inherits all the behavior of a typical user—we show this relationship with a *generalization* between them. Part of the repair person's activities are to perform the repair (which includes operating the refrigerator) and accessing a repair log. To show this, we need to add two more use case model elements, Repair and Access Log. Since the repair process uses the Operate Fridge use case plus other steps, the Repair use case *extends* (builds on the functionality of) the Operate Fridge use case. This is indicated by an <<*extends*>> stereotype on the dependency between the Repair use case and the Operate Fridge use case, as seen in Figure 4.10.

**To build upon** Figure 4.9 **to create the diagram in** Figure 4.10**:**

1. Add two new use cases: Repair and Access Log.
2. Show that the Repair use case extends the Operate Fridge use case by adding a dependency between them with a <<*extends*>> stereotype. The arrow should point towards Operate Fridge.
3. Add a new actor named "Repair Person."
4. Show the relationship between the Repair Person and the User by adding a generalization between them. The arrow should point towards User.
5. Add an association from the Repair Person to the Repair use case.
6. Add a dependency from Repair to Access Log with an <<*includes*>> stereotype.

Now you've completed your first use case diagram. The mechanics of creating class and sequence diagrams are very similar, as you will see in Chapter 5 and Chapter 6, respectively.

Figure 4.10 **Example with “Extends” Dependency and Generalization**



# Chapter 5

# Class Diagrams

## 5.1 Concepts

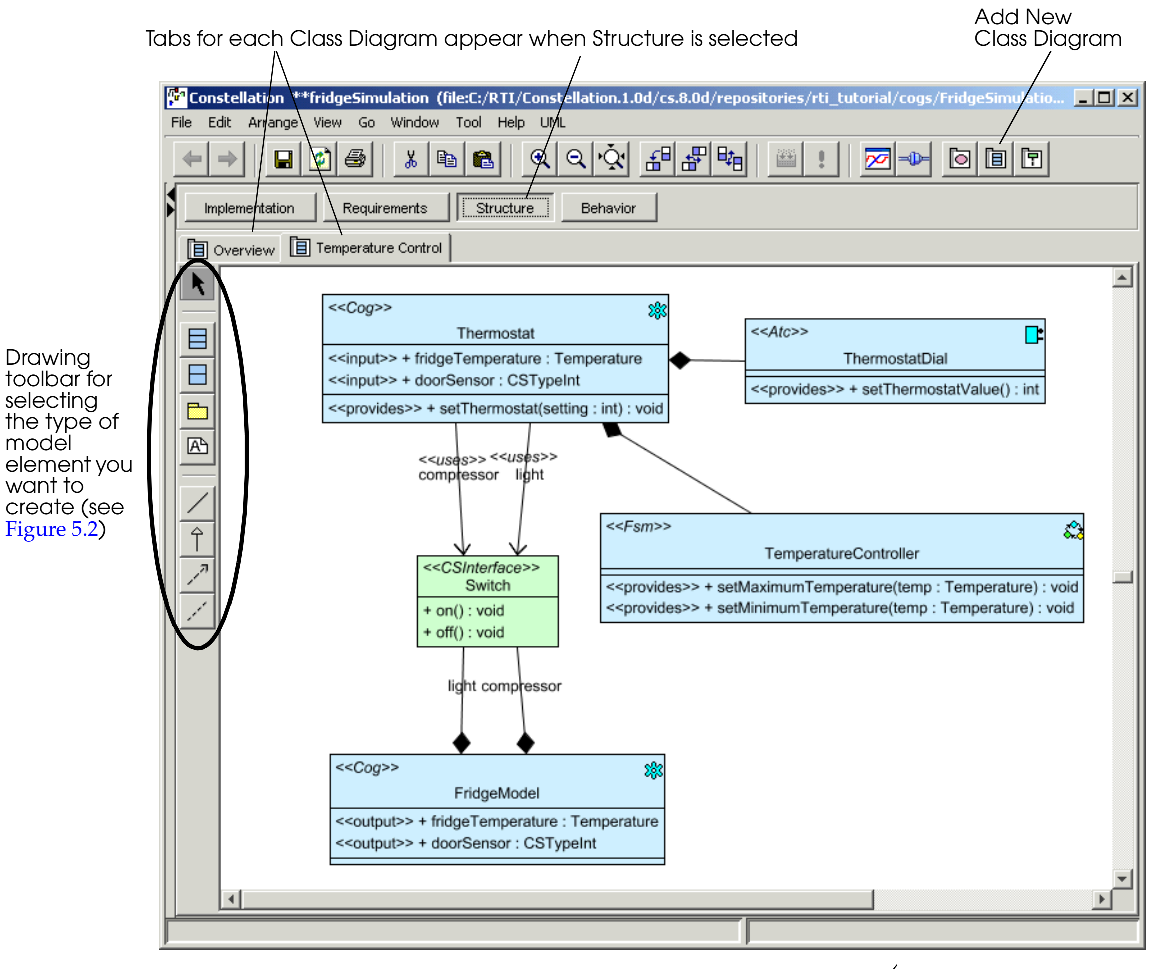
Class diagrams are perhaps the most commonly used type of UML diagram. They capture the structural relationships between various elements in a system without modeling too many of the implementation details. They are generally used starting with the architecture phase all the way through the implementation phase.

Class diagrams can also be used to reverse-engineer an implemented system, producing documentation of the system's high-level architecture.

By capturing only essential elements, a good class diagram facilitates understanding at the right level of granularity. It is common for a work unit to contain several class diagrams that share some elements. This kind of arrangement allows you to focus on different parts of the system—one class diagram at a time.

This chapter describes the model elements that can be used in class diagrams. Figure 5.1 shows a sample class diagram.

Figure 5.1 **Class Diagram Workspace**

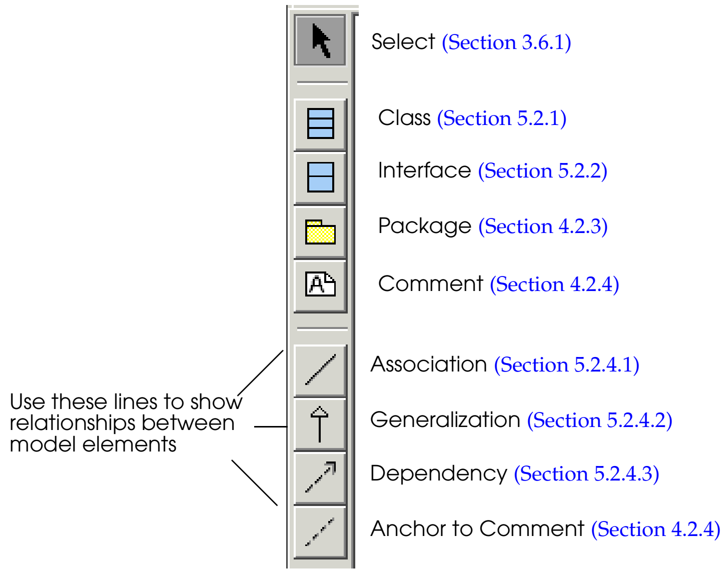

Tabs for each Class Diagram appear when Structure is selected
Add New Class Diagram
Drawing toolbar for selecting the type of model element you want to create (see Figure 5.2)
Constellation **fridgeSimulation (file:C:/RTI/Constellation.1.0d/cs.8.0d/repositories/rti_tutorial/cogs/FridgeSimulatio...
File Edit Arrange View Go Window Tool Help UML
Implementation
Requirements
Structure
Behavior
Overview
Temperature Control
<<Cog>>
Thermostat
<<input>> + fridgeTemperature : Temperature
<<input>> + doorSensor : CSTypeInt
<<provides>> + setThermostat(setting : int) : void
<<Atc>>
ThermostatDial
<<provides>> + setThermostatValue() : int
<<uses>> compressor
<<uses>> light
<<Fsm>>
TemperatureController
<<provides>> + setMaximumTemperature(temp : Temperature) : void
<<provides>> + setMinimumTemperature(temp : Temperature) : void
<<CSInterface>>
Switch
+ on() : void
+ off() : void
light
compressor
<<Cog>>
FridgeModel
<<output>> + fridgeTemperature : Temperature
<<output>> + doorSensor : CSTypeInt

*This class diagram is provided in the rti_tutorial repository's FridgeSimulation COG.*

## 5.2 Model Elements

Figure 5.2 shows the types of model elements you can include in class diagrams. Most of these model elements are also allowed in use case diagrams and therefore are described in Chapter 4. The only model elements unique to class diagrams are *classes* (Section 5.2.1) and *interfaces* (Section 5.2.2).

Figure 5.2 **Drawing Toolbar for Class Diagrams**


Select (Section 3.6.1)
Class (Section 5.2.1)
Interface (Section 5.2.2)
Package (Section 4.2.3)
Comment (Section 4.2.4)
Use these lines to show relationships between model elements
Association (Section 5.2.4.1)
Generalization (Section 5.2.4.2)
Dependency (Section 5.2.4.3)
Anchor to Comment (Section 4.2.4)

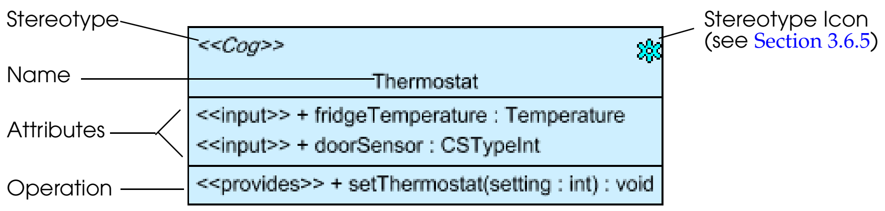
*To insert one of these model elements, click the button to select, then click inside the workspace. To insert more than one, double-click the button, then click multiple times in the workspace.*

For general information on how to work with diagrams and model elements, see Section 3.5 and Section 3.6, respectively.

### 5.2.1 Classes

A *class* model element denotes a structural element, potentially with some implementation attached. As seen in Figure 5.3, a class is shown as a rectangle that has up to three sections. The top section shows the class's name and stereotype. The middle section shows the attributes of the class. The bottom section shows the operations offered by that class.

Figure 5.3 **Class Model Element**



Stereotype
Name
Attributes
Operation
<<Cog>>
Thermostat
<<input>> + fridgeTemperature : Temperature
<<input>> + doorSensor : CSTypeInt
<<provides>> + setThermostat(setting : int) : void
Stereotype Icon
(see Section 3.6.5)

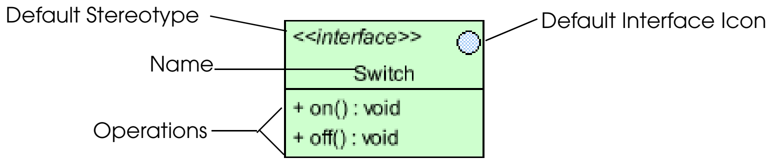
The last two sections are optional—you do not have to show them. In fact, hiding attributes and operations is usually a good idea when showing classes that are peripheral to the concept being explained.

If you give a class a stereotype that matches one of *Constellation*'s component types—ATC, DFC, STC, COG, FSM, or APP—a matching icon appears in the upper right corner, as seen in Figure 5.3. You can also use your own predefined stereotypes and their associated icons—see Section 3.6.5.

## 5.2.2 Interfaces

An interface model element is a common way of grouping operations used by multiple classes. In our Fridge example, both the Thermostat and the FridgeModel classes use the same set of operations from the Switch interface. As seen in Figure 5.4, interfaces are rendered much the same way as classes, except that the attribute compartment is not shown by default and it has a default stereotype and icon.

Figure 5.4 **Interface Diagram**

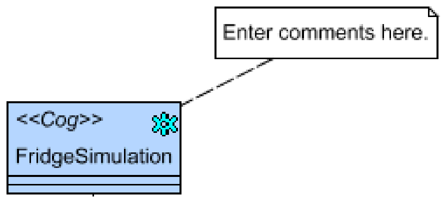


## 5.2.3 Comments

A *comment* model element allows you to attach simple documentation to any model element in a diagram. Comments are typically used to describe certain characteristics that are not otherwise apparent.

Select the Comment tool from the drawing toolbar and click near the model element you want to comment. Double-click the text of the comment and type whatever documentation you desire. When you are finished typing, click outside the text box to commit your change (or press <**Escape**> to revert your changes>). Select the Anchor to Comment tool and use it to draw a line between your new comment and the model element to which it refers, as seen in Figure 5.5.

Figure 5.5 **Comment**



## 5.2.4 Relationships

The types of relationships you can show between elements in a class diagram include:

- ❑ Generalizations (Section 5.2.4.2)
- ❑ Associations and Association Ends (Section 5.2.4.1)
- ❑ Dependencies (Section 5.2.4.3)

To draw these types of lines between two model elements, select the desired button from the drawing toolbar; then in the workspace, simply drag the line from one model element to the other.

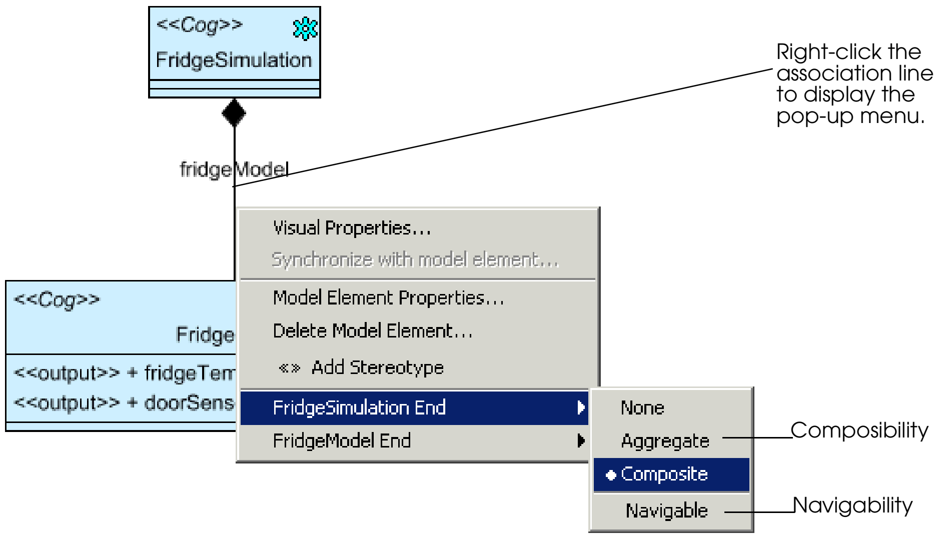
### 5.2.4.1 Associations and Association Ends

An *association* is another way to show a relationship between two model elements. Each association is drawn as a line with two endpoints called *association ends*. You indicate the nature of the relationship by selecting different properties for the association ends, as described in this section.

As is true with all types of UML model elements, an association can have a name and a stereotype, and so can each of the association ends. To enter names and stereotypes for an association and its end points, double-click the line to display the Properties window for the association (as described in Section 3.6.3).

Each association end also has a set of properties that you can modify by right-clicking and using the resulting pop-up menu. For examples, see Figure 5.6 and Figure 5.9.

Figure 5.6 **Association Ends: Specifying Composibility and Navigability**



**Properties for Association Ends**

❑ **Composibility** An *association end* can have three kinds of composibility: None, Aggregate, and Composite. These types are described below. Figure 5.6 shows how to make your selection. Each choice is indicated in the workspace with a different type of diamond tip, as seen in Figure 5.7.

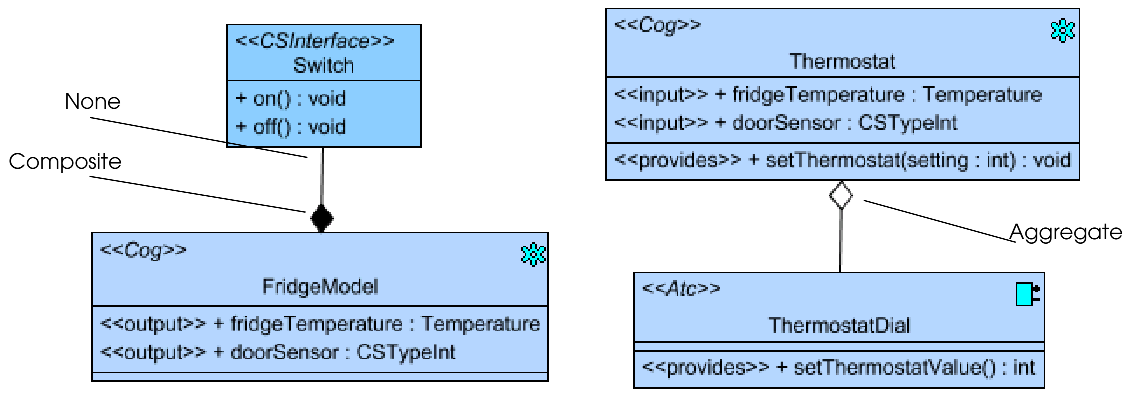
Dependencies, plain associations, aggregated associations, and composed associations form a continuum of increasingly strong claims about the elements they connect.

- **None** The default. This shows the weakest type of association. The model element connected to this type of association end knows about (makes references to) the element on the other end, but that is the extent of their relationship.

  For example, if a Compressor is associated with a Door in this manner, it implies that the Compressor knows about the Door, but it does not own the Door, and creation or destruction of one does not follow to the other—they are independent. This type of association end is shown by the lack of a diamond at the end. In Figure 5.7 we see that the Switch interface knows about the FridgeModel and the ThermostatDial knows about the ThermoStat, but no ownership or containment exists.

- **Aggregate** This type of end shows that one model element contains another model element. For example, a Thermostat has a ThermostatDial. This type of association end is shown by a hollow diamond on the aggregation end (the end that contains the included elements).
- **Composite** This type of end shows a strict ownership relationship between two model elements. It is a strong type of aggregation. In a composite, each model element belongs to a single collection. The elements that make of the collection have no reason to exist by themselves. It is common to refer to this type of relationship as a "*has a*" or containment relationship. For example, the relationship "FridgeModel *has a* Switch" will be shown as the FridgeModel class composing a Switch class. Composition implies that the composed object (Switch) will be created and destroyed along with the composing object (FridgeModel). This type of association end is shown by a solid diamond on the composing end (FridgeModel).

Figure 5.7 **Association Ends: Arrowheads used for Compositibility**

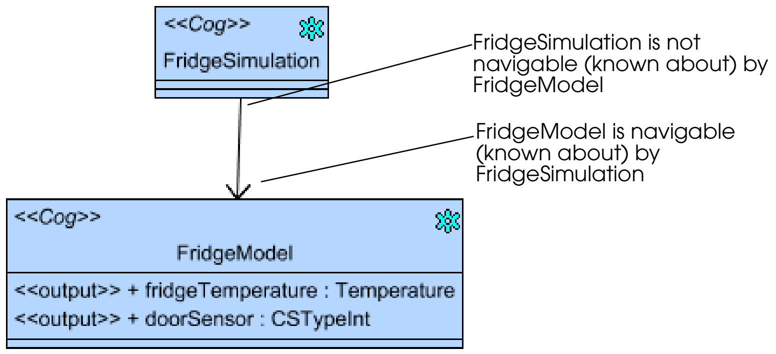


<<CSInterface>>
Switch
+ on() : void
+ off() : void
None
Composite
<<Cog>>
FridgeModel
<<output>> + fridgeTemperature : Temperature
<<output>> + doorSensor : CSTypeInt
<<Cog>>
Thermostat
<<input>> + fridgeTemperature : Temperature
<<input>> + doorSensor : CSTypeInt
<<provides>> + setThermostat(setting : int) : void
Aggregate
<<Atc>>
ThermostatDial
<<provides>> + setThermostatValue() : int

❑ **Navigability** This type of end simply shows that one model element knows about (can make references to) the model element on the other end. The default situation is that the model elements connected by an association have a mutual relationship—both ends know about each other and are free to make references to each other. This is rendered as a line with no arrowheads on the endpoints (the default). If you want to restrict the relationship so that it is one-way, turn *off* navigability for the end that should not be referenced. Figure 5.6 shows how to make your selection.

Setting an end to be *not* navigable adds an arrow to the opposite end—so that it appears as a one-way path between the elements. The end without an arrowhead is not navigable (cannot be referenced by the other end). Figure 5.8 illustrates the both types of end points.

Figure 5.8 **Association End: Arrowhead used for Navigability**



<<Cog>>
FridgeSimulation
FridgeSimulation is not navigable (known about) by FridgeModel
FridgeModel is navigable (known about) by FridgeSimulation
<<Cog>>
FridgeModel
<<output>> + fridgeTemperature : Temperature
<<output>> + doorSensor : CSTypeInt

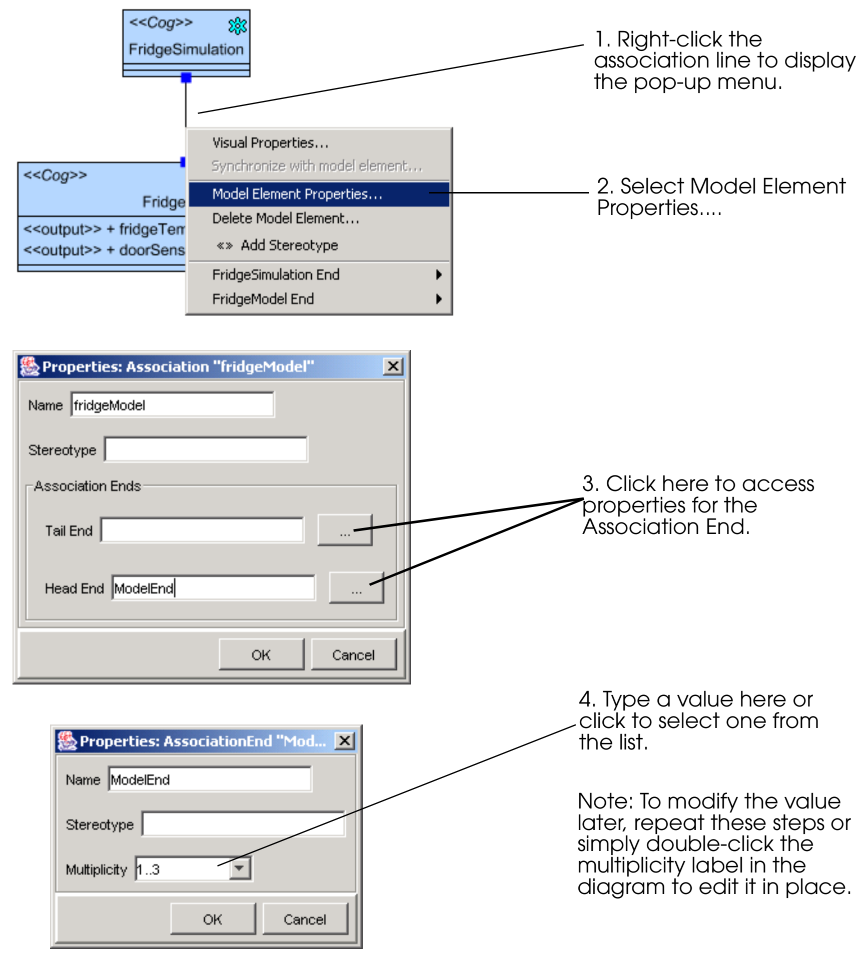
*In this example of navigability, the FridgeSimulation can refer to the FridgeModel, but the FridgeModel knows nothing about the FridgeSimulation.*

❑ **Multiplicity** Association ends can also show *multiplicity*, which indicates the number of objects of one class that are associated with an object in another class. For example, you can specify that a refrigerator has two doors, or that a thermostat is associated with *x* number of temperature sensors. You can choose not to specify multiplicity at all, signifying either that it is unimportant or it is undetermined. To specify a value, right-click the association and follow the steps in Figure 5.9. The various ways of specifying multiplicity are described below.

- **Any number (including zero)** Use an asterisk (“*”) to show that you can have any number of class or interface objects associated with that association end.
- **A specific number** Enter an exact number of class or interface objects associated with that association end. For example, you can specify that your refrigerator has two doors by entering “2” for the association end near the Doors class.

- **A range** Specify a valid range for the number of objects by entering two numbers separated with ".."—for example **2..4** means that you can have either 2, 3 or 4 associated objects—no more, no less. Specifying **4..*** means that you can have four or more associated objects. Note that **0..*** has the same meaning as a single asterisk ("*").
- **Multiple ranges** Specify a set of valid ranges by entering a series of comma-separated ranges. For example, **1..3, 6..7** specifies that you can have either 1, 2, 3, 6, or 7 associated objects.

Figure 5.9 **Association Ends: Specifying Multiplicity**



Name: fridgeModel
Stereotype:
Tail End:
Head End: ModelEnd

Name: ModelEnd
Stereotype:
Multiplicity: 1..3

1. Right-click the association line to display the pop-up menu.

2. Select Model Element Properties....

3. Click here to access properties for the Association End.

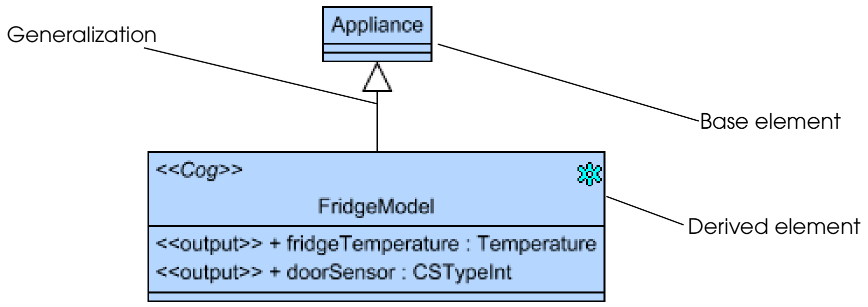
4. Type a value here or click to select one from the list.

Note: To modify the value later, repeat these steps or simply double-click the multiplicity label in the diagram to edit it in place.

#### 5.2.4.2 Generalizations

A *generalization* shows a relationship between a base element and a derived element. For example, you can show that FridgeModel is a derived class of Appliance by drawing a generalization between them. As seen in Figure 5.10, the line for a generalization has a hollow-tipped arrow pointing towards the base element. For more information on the concept of generalizations, see Section 4.2.5.2.

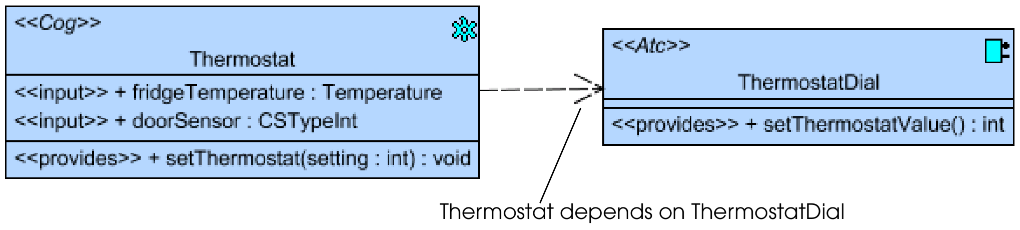
Figure 5.10 **Generalization**


Generalization
Appliance
Base element
<<Cog>>
FridgeModel
<<output>> + fridgeTemperature : Temperature
<<output>> + doorSensor : CSTypeInt
Derived element

#### 5.2.4.3 Dependencies

A *dependency* shows that one model element uses—is dependent on—another. For example, you could show that a Thermostat depends on a ThermostatDial. A dependency appears as a dashed line with an arrowhead pointing towards the class that is depended on (from Thermostat to ThermostatDial), as seen in Figure 5.11. Two commonly used stereotypes for dependencies are ***<<includes>>*** and ***<<extends>>***, see Section 4.2.5.3 for details.

Figure 5.11 **Dependency**


<<Cog>>
Thermostat
<<input>> + fridgeTemperature : Temperature
<<input>> + doorSensor : CSTypeInt
<<provides>> + setThermostat(setting : int) : void
<<Atc>>
ThermostatDial
<<provides>> + setThermostatValue() : int
Thermostat depends on ThermostatDial

# Chapter 6

# Sequence Diagrams

## 6.1 Concepts

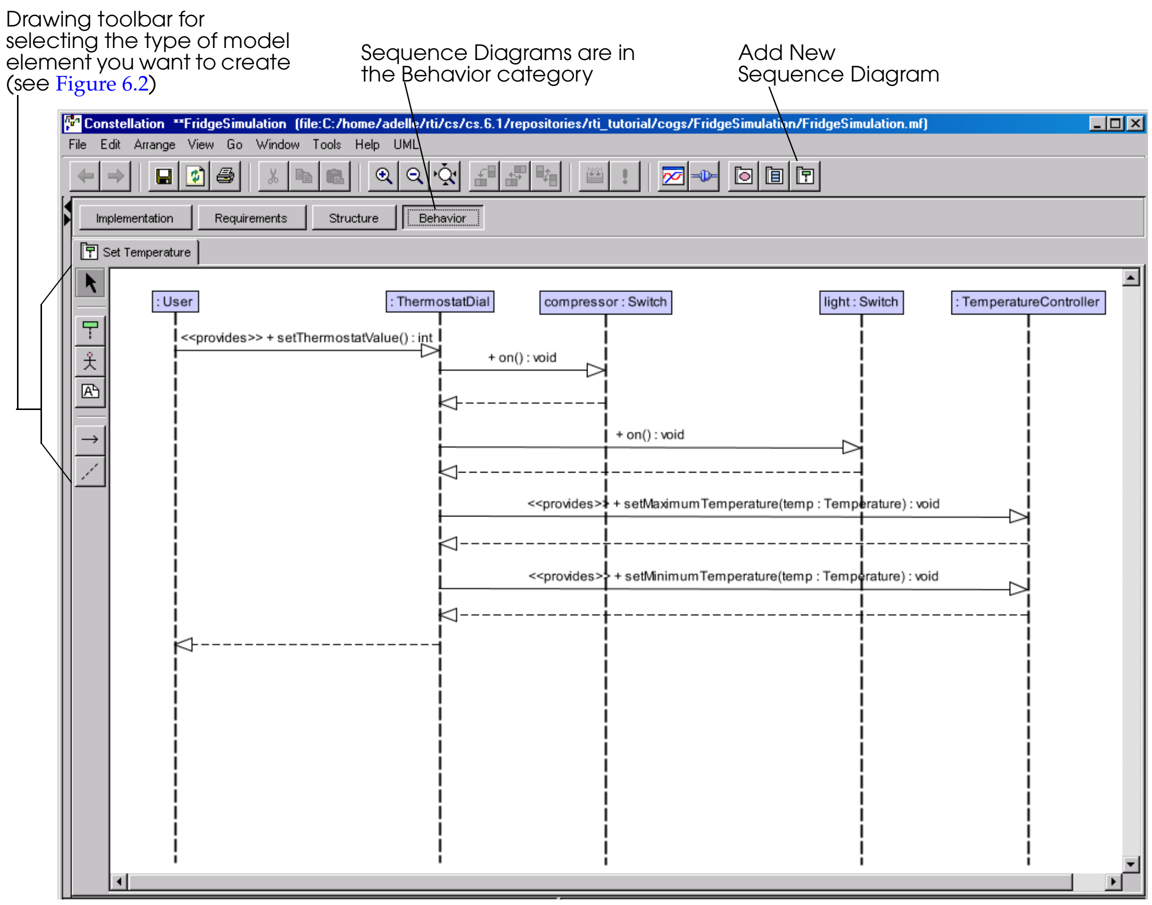
A sequence diagram models the interactions among objects and modules in your system by showing you a time line of the messages sent among them. They are frequently used in parallel with class diagrams, which model the structure of the system, to show the order of method calls among objects. However, they can also be used to model, at a very high level, the semantic interactions among roles played by objects whose types have yet to be specified. In this capacity, sequence diagrams may be used in parallel with use case diagrams.

For example, consider the refrigerator we have been modeling. Suppose that you are in the project's early design phase and have a use case diagram showing a user opening and closing the refrigerator door, as seen in Figure 4.1. Note, however, that the use case diagram specifies neither the steps required to perform these tasks nor the ordering of these steps. A high-level sequence diagram is well suited to capture just that information.

Now suppose that the structure of your refrigerator application has been modeled with class diagrams. The players and their APIs are well defined. Your class diagrams specify which messages class instances should respond to, but they do not specify which objects should be calling which methods when. Sequence diagrams can help you in this situation as well.

This chapter describes the model elements that can be used in sequence diagrams. Figure 6.1 shows a sample sequence diagram from the **rti_tutorial** repository (FridgeSimulation.cog).

Figure 6.1 **Sequence Diagram Workspace**


Drawing toolbar for selecting the type of model element you want to create (see Figure 6.2)
Sequence Diagrams are in the Behavior category
Add New Sequence Diagram
Constellation **FridgeSimulation (file:C:/home/adelle/rti/cs/cs.6.1/repositories/rti_tutorial/cogs/FridgeSimulation/FridgeSimulation.mf)
File Edit Arrange View Go Window Tools Help UML
Implementation
Requirements
Structure
Behavior
Set Temperature
: User
: ThermostatDial
compressor : Switch
light : Switch
: TemperatureController
<<provides>> + setThermostatValue() : int
+ on() : void
+ on() : void
<<provides>> + setMaximumTemperature(temp : Temperature) : void
<<provides>> + setMinimumTemperature(temp : Temperature) : void

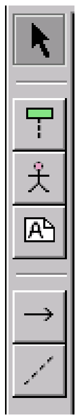
*This sequence diagram is provided in the rti_tutorial repository in the FridgeSimulation COG. The user turns the thermostat dial to select a temperature. Based on the temperature selected, the dial instructs the compressor and light switches to turn on, sets the maximum and minimum temperatures, and returns control to the user.*

## 6.2 Model Elements

Figure 6.2 shows the types of model elements you can include in sequence diagrams. Comments and their anchors are also allowed in class diagrams and therefore are described in Chapter 5. The two most important model element types in sequence diagrams are *classifier roles* (Section 6.2.1) and *messages* (Section 6.2.2).

For general information on how to work with diagrams and model elements, see Section 3.5 and Section 3.6, respectively.

Figure 6.2 **Drawing Toolbar for Sequence Diagrams**



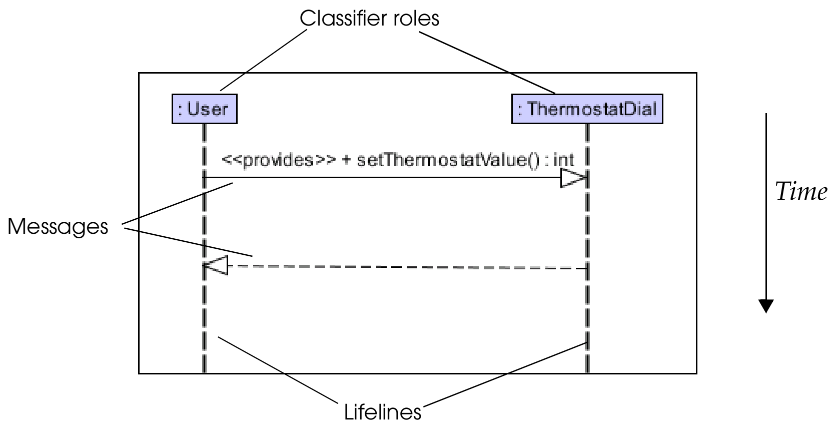
*To insert one of these model elements, click the button to select, then click inside the workspace. To insert more than one, double-click the button, then click multiple times in the workspace.*

### 6.2.1 Classifier Roles

A *classifier role* represents a semantic role played by an object in your system. Like an object, it may have a name, a base type, and so on. However, unlike a regular object in a language such as C++, a classifier role may belong to many types or none at all. A classifier role with no base types is useful when your sequence diagram models interactions at a very high level, before the names and identities of classes and interfaces in your implemented system are known.

Classifier roles appear as rectangles with vertical lines stretching downwards beneath them, as seen in Figure 6.3. These vertical lines are called *lifelines*. They represent the passage of time. Events occurring later appear beneath events that occurred earlier.

Figure 6.3 **Sequence Diagram Terms**



#### 6.2.1.1 Names and Base Types

The rectangle at the top of a classifier role's presentation is analogous to the top compartment of a class in a class diagram: it displays the name and stereotype of the classifier role. Unlike a class, a classifier role, like an object, may represent an instance of one or more base types.

The name of a classifier role itself and the name(s) of its base type(s) are typically both displayed and are separated by a colon, such as "**roleName : baseType**". The names of multiple base types are separated by commas.

For example:

- ❏ **"roleName :"** denotes a classifier role with the name "roleName" having no base type.
- ❏ **": className"** denotes a classifier role with no name, or whose individual name is unimportant, with a base type "className."
- ❏ **"role : class1, class2, class3"** denotes a classifier role named "role" of base types "class1," "class2," and "class3."

By displaying the colon even when the role name or the base type name is not present, it makes it easy to determine which of these names a string represents by looking at the relative position of the colon.

#### 6.2.1.2 Adding Classifier Roles

Because classifier roles in sequence diagrams often represent instances of classes that are also present in your model, you may want to create the class diagrams first. You can show the relevant classes and the relationships among them in those diagrams and then use the existing model elements to create classifier roles. The easiest way to do this is simply by dragging and dropping a model element from the model browser into a diagram:

- ❏ If the dropped model element is a use case, class, interface, or actor, a new classifier role will be created and displayed in the diagram. The new classifier role will have the dropped model element as its base type.
- ❏ If the dropped model element is itself a classifier role, that role will be displayed in the diagram.

If you think of classifier roles as analogous to objects, then a role's base classifier is the class to which that object belongs. When you drag and drop a class into a sequence diagram, you are in effect inserting an "instance" of that class.

The Actor button on the drawing toolbar (see Figure 6.2) not only creates a new classifier role but also a new actor to serve as a base classifier for that role. (When you use it, you will see both new model elements appear in the browser under the Model folder.) The new classifier role visually indicates that its base is an actor by displaying a very small stick figure to the right of its title box. The Actor tool is just a convenience feature; you could get the same effect by right-clicking the Model folder in the browser, selecting "Add New Actor," and then dragging and dropping that new actor into the sequence diagram.

### 6.2.2 Messages

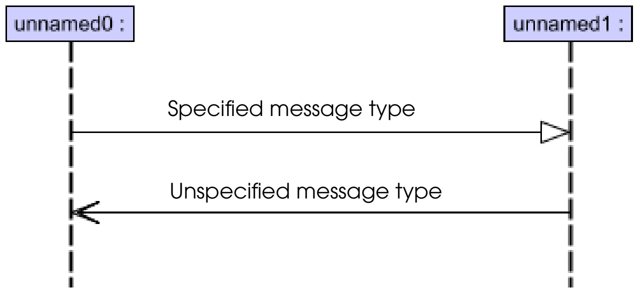
A *message* represents a directed communication between classifier roles. Like relationship model elements such as associations and generalizations, a message is drawn as a line.

Unlike relationship model elements, messages do not model physical structure; they model behavior. However, it is important to note that behavior implies structure: if an object of one type can send a message to an object of another type, the first object typically contains some kind of reference to the second.

In most cases, the time it takes to send a message may be considered instantaneous. For this reason, messages are usually drawn as horizontal lines: the sending and receiving ends appear at the same vertical level, and hence at the same moment in time. However, this need not be the case. A message may represent a method call using a Remote Procedure Call (RPC) technology such as CORBA®, in which case network latency will introduce a delay between a when a message is sent and when it is received. If the message will be received at a time substantially later than when it was sent, the message line is typically drawn slanting downwards. (Note that the tool prevents you from slanting the message line upwards, since that would indicate the message travelled backwards in time and was received before being sent.)

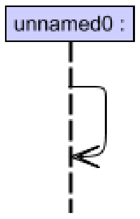
Messages are drawn as lines between the lifelines of two classifier roles. Each message line has an arrowhead that points toward the receiver of that message. If the message's type is set to **None** (the default), the arrowhead appears as two connecting lines (not a triangle). Once the type is specified, the arrowhead becomes a hollow triangle. Figure 6.4 shows both types of arrowheads. (Message types are described in Section 6.2.2.1.)

Figure 6.4 **Message Arrowheads in Sequence Diagrams**



Messages sent by a classifier role to itself double back on themselves to reconnect with the sending role, as seen in Figure 6.5. Arrowheads have the same meaning as discussed above.

Figure 6.5 **Message Sent by a Role to Itself**



### 6.2.2.1 Message Types

Messages may be of several types, corresponding to the various ways in which objects may communicate with one another. A method call takes place via a *call* message; the return of control at the end of a method call is represented by a *return* message, which appears as a dashed line. One classifier role instantiates another via a *create* message and destroys it via a *destroy* message. A classifier role may send messages to itself and may even destroy itself. Such self-destruction takes place via a *terminate* message.

A message’s type is set in its Properties window. As is true for all model elements, you can open a selected message’s Properties window by double-clicking, or by right-clicking and selecting **Model Element Properties...**. from the pop-up menu.

Setting the **Action** field in the Properties window to **Call method** indicates that the message's sender is calling a method provided by the message's receiver. The adjacent list box lets you select which method from a list of operations contained by the receiving classifier role’s base classifiers.

# Chapter 7

# Connecting Design and Implementation

In Chapter 2, "Development Process," you learned about the process-oriented concepts at work in *Constellation*. Chapter 3 through Chapter 6 discussed particular diagram types and related features. This chapter brings you full circle by showing you how to create and maintain connections between your software design and implementation.

## 7.1 Classes and Components

The class model elements of your system describe the various entities that exist in the system, the APIs they offer, and the relationships they have with one another. These features make such structural models ideal for design and documentation. However, to actually implement a class model element in software you need a more expressive language. For instance, not only do you need to specify that your class has a method called Execute(), you must also specify who will call it, when, and in what order. You must further specify which other classes your class depends on, and the specific methods it will call. The components (ATCs, DFCs, STCs, COGs, FSMs, and applications) presented in *Constellation*'s Diagram and Form Views provide these capabilities.

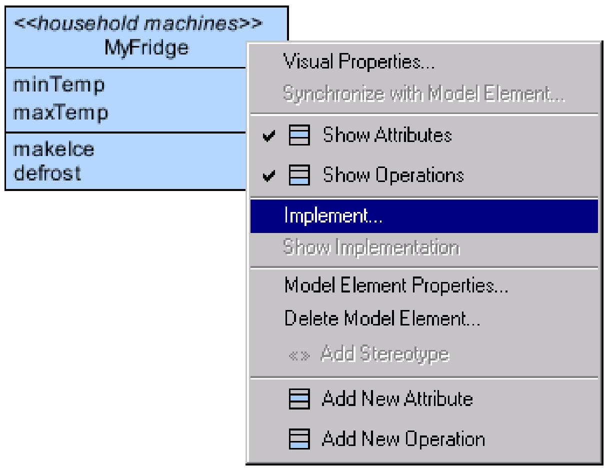
When you are ready to create a component that will implement the API for a class model element, it is not necessary to start from scratch. *Constellation* offers a wizard interface that can create or update a component based on a class model element. Conversely, the wizard can also take a preexisting component and create or update a class model element to match the component. The wizard also sets up hyperlinks between a component and its UML diagram.

## 7.2 Using the Class-to-Component Wizard

### 7.2.1 Creating a New Component

Your first use of the wizard will likely be when you are creating a component for the first time. If you have a class model element that specifies the APIs of the component you want to create, right-click a depiction of the class in any diagram and select **Implement...** as seen in Figure 7.1; this command will start the wizard.

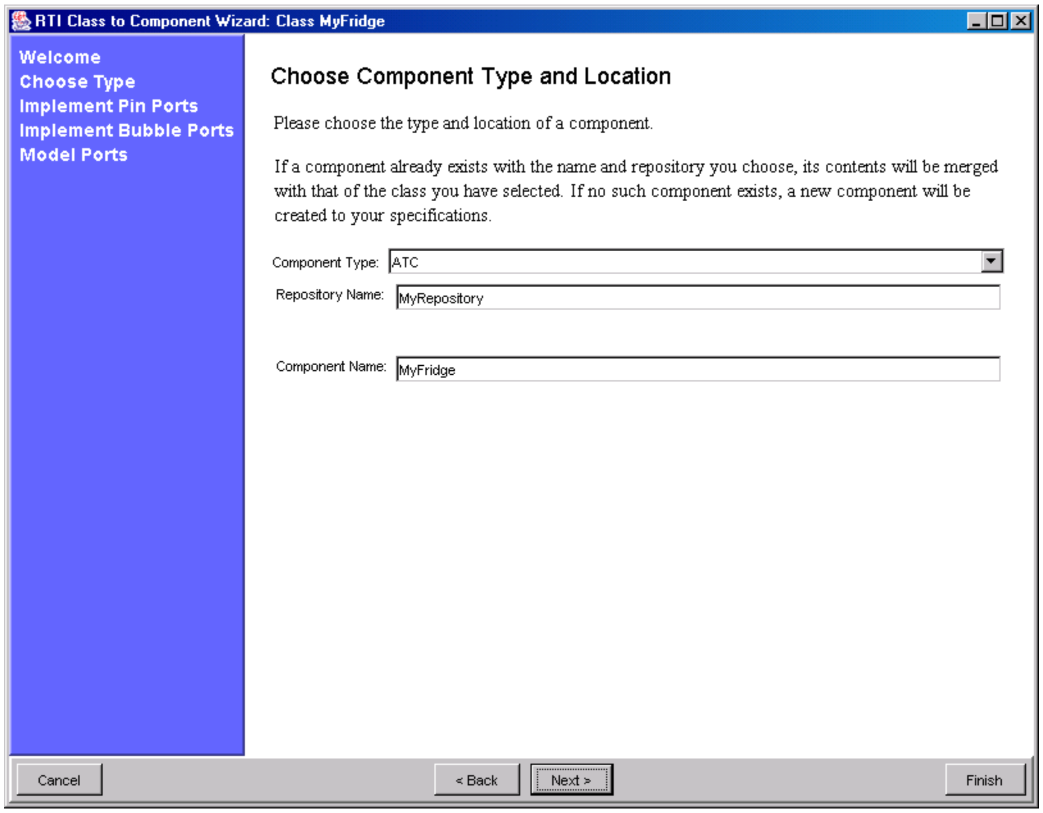
Figure 7.1 **Implementing a Class**



The wizard helps you create pin and bubble ports for each attribute and operation in the source class.

As seen in Figure 7.2, you will start by selecting a type (ATC, DFC, STC, COG, FSM, or application) and repository for your new component. If you need help deciding what type of component to use, see the *Constellation User's Manual*. Or, you can enter the name and repository of an existing component, in which case the Component Type field will match the existing component and become uneditable.

Figure 7.2 **Selecting a Component Type and Location**



Choose Component Type and Location

Please choose the type and location of a component.

If a component already exists with the name and repository you choose, its contents will be merged with that of the class you have selected. If no such component exists, a new component will be created to your specifications.

Component Type: ATC

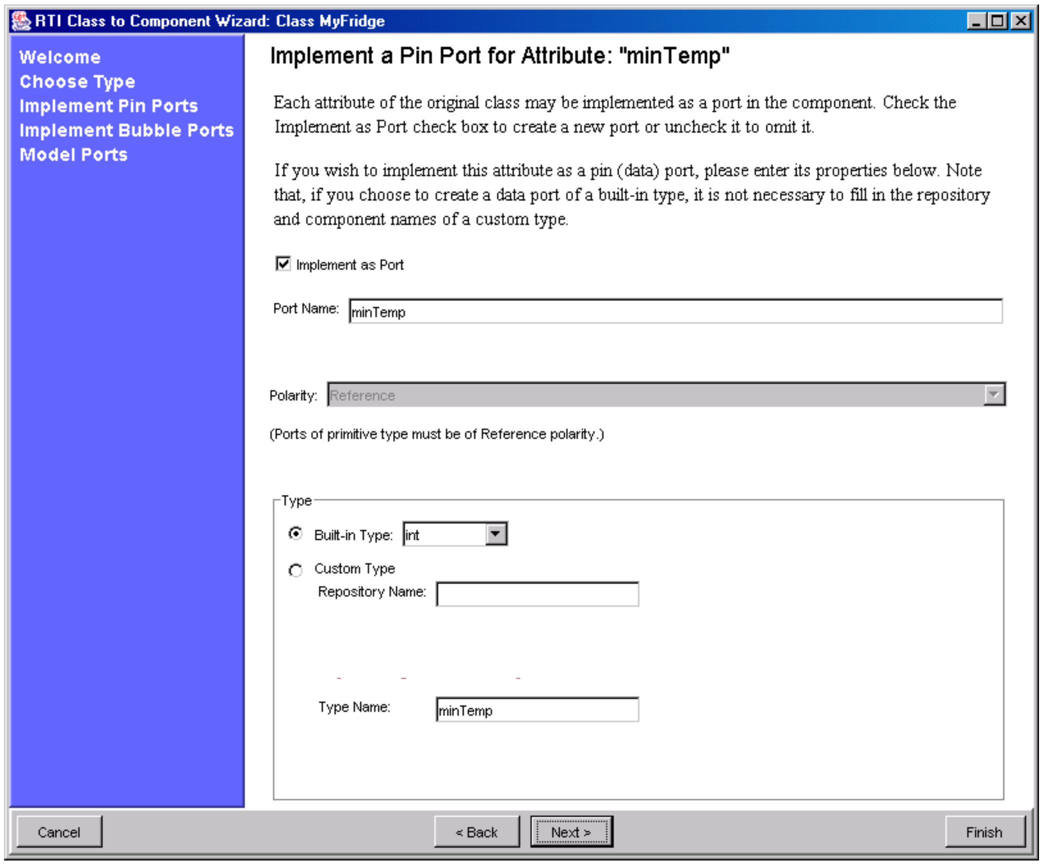
Repository Name: MyRepository

Component Name: MyFridge

If the wizard has enough information to supply default values for all of the remaining prompts, the Finish button will be enabled. You can click **Finish** to use the defaults or click **Next** to proceed through the prompts and supply information.

Next, the wizard will allow you to create pin (data) ports for each of the class's attributes, and bubbles for each of its operations. To skip an attribute or operation, clear the Implement check box and click Next. See Figures 7.3 and 7.4 for examples.

Figure 7.3 **Defining Pins for Attributes**


RTI Class to Component Wizard: Class MyFridge

Implement a Pin Port for Attribute: "minTemp"

Each attribute of the original class may be implemented as a port in the component. Check the Implement as Port check box to create a new port or uncheck it to omit it.

If you wish to implement this attribute as a pin (data) port, please enter its properties below. Note that, if you choose to create a data port of a built-in type, it is not necessary to fill in the repository and component names of a custom type.

[✓] Implement as Port

Port Name: minTemp

Polarity: Reference

(Ports of primitive type must be of Reference polarity.)

Type

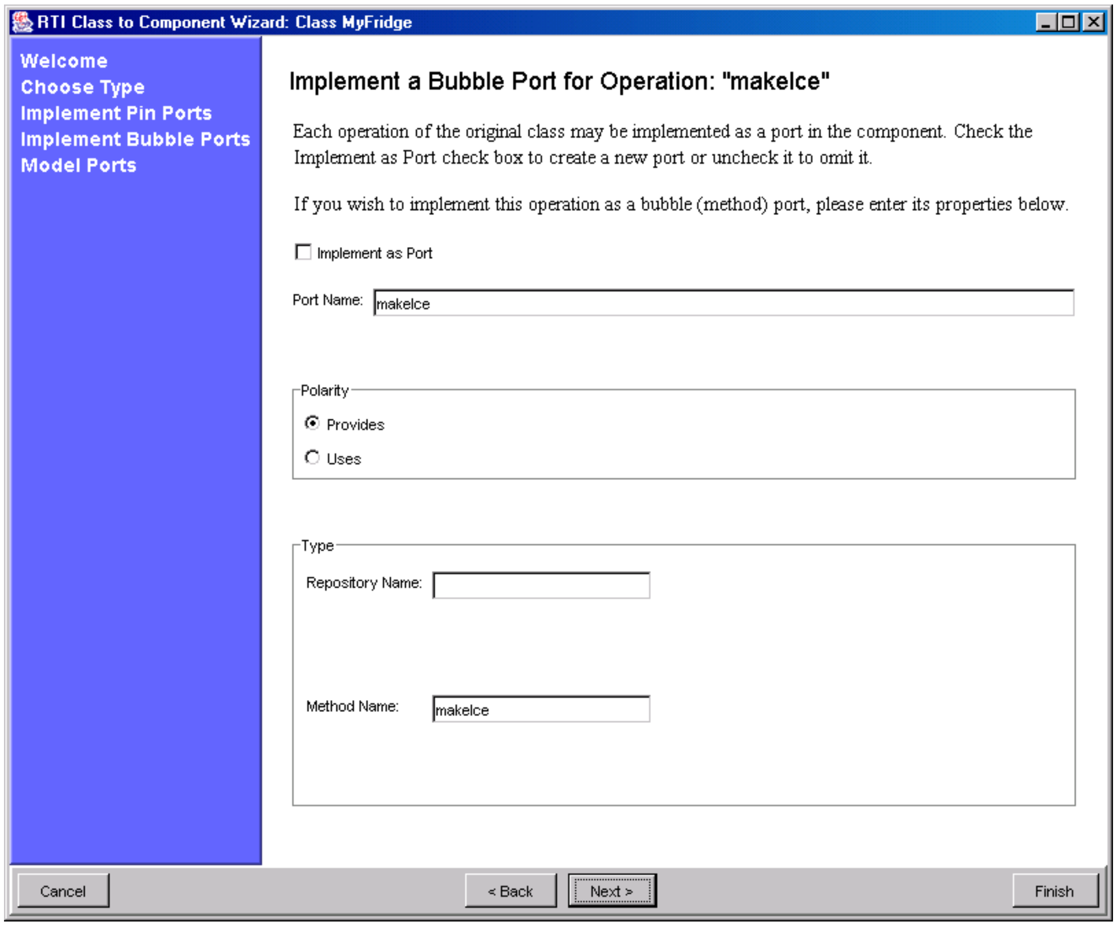
[● Built-in Type:] int

○ Custom Type

Repository Name:

Type Name: minTemp

Figure 7.4 **Defining Bubbles for Operations**



Implement a Bubble Port for Operation: "makeIce"

Each operation of the original class may be implemented as a port in the component. Check the Implement as Port check box to create a new port or uncheck it to omit it.

If you wish to implement this operation as a bubble (method) port, please enter its properties below.

☐ Implement as Port

Port Name: makeIce

Polarity

[✓ Provides]

Uses

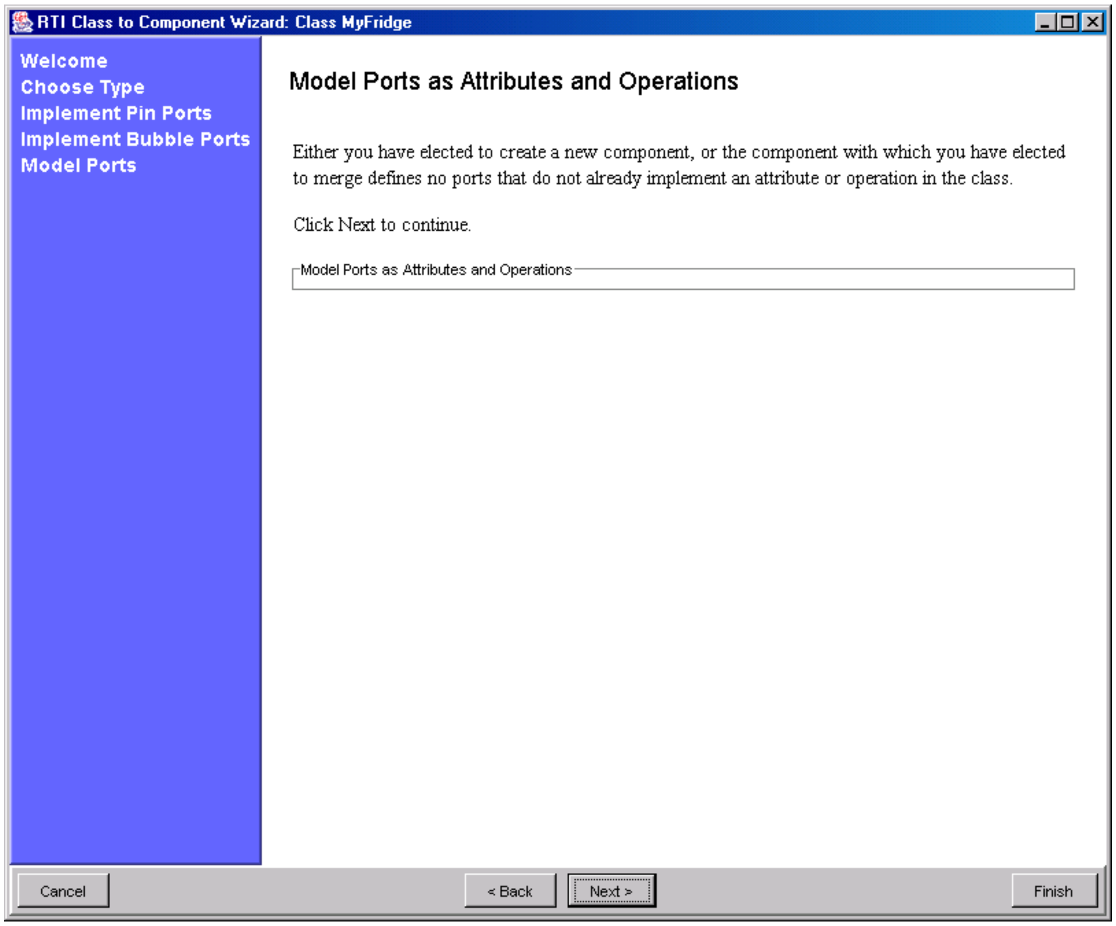
Type

Repository Name:

Method Name: makeIce

The next screen, seen in Figure 7.5, is used to create class attributes and operations for ports in existing components. Therefore, this screen is only used if you selected an existing component at the start of the wizard (see Figure 7.2). For example, suppose that you have started implementing the components for the Fridge application. While doing so, you realize that one of the components needs another port. You add it and, after debugging, want to reflect the additional port in the class model element. To do this, you invoke the wizard again to synchronize the component and the class. The screen in Figure 7.5 shows the new port in the list and allows you to create a new operation for it in the class model element.

Figure 7.5 **Modeling Ports as Attributes and Operations**



Model Ports as Attributes and Operations

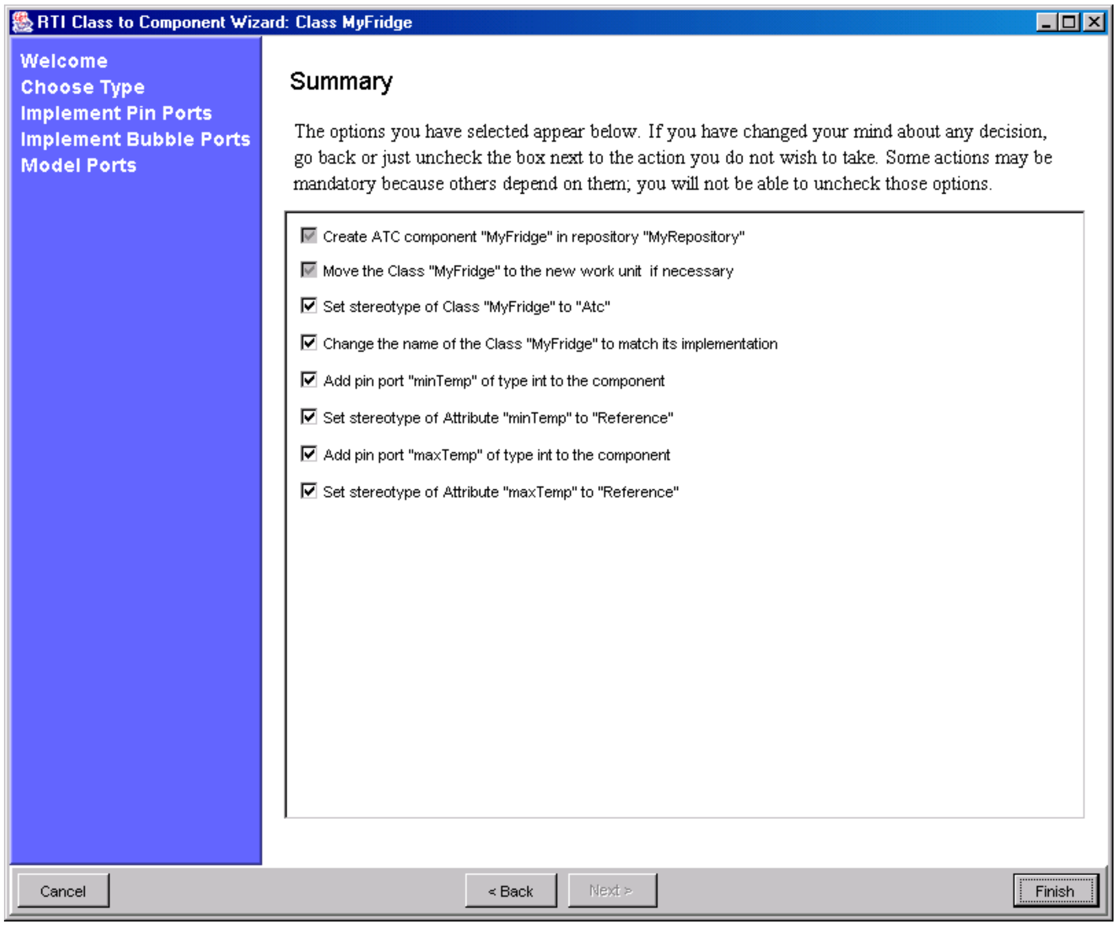
Either you have elected to create a new component, or the component with which you have elected to merge defines no ports that do not already implement an attribute or operation in the class.

Click Next to continue.

Model Ports as Attributes and Operations

The wizard's last screen, seen in Figure 7.6, presents a summary and gives you a chance to make any final changes before the component is created or updated. When you are ready to proceed, click **Finish**.

Figure 7.6 **Summary Window**



Summary

The options you have selected appear below. If you have changed your mind about any decision, go back or just uncheck the box next to the action you do not wish to take. Some actions may be mandatory because others depend on them; you will not be able to uncheck those options.

- [✓] Create ATC component "MyFridge" in repository "MyRepository"
- [✓] Move the Class "MyFridge" to the new work unit if necessary
- [✓] Set stereotype of Class "MyFridge" to "Atc"
- [✓] Change the name of the Class "MyFridge" to match its implementation
- [✓] Add pin port "minTemp" of type int to the component
- [✓] Set stereotype of Attribute "minTemp" to "Reference"
- [✓] Add pin port "maxTemp" of type int to the component
- [✓] Set stereotype of Attribute "maxTemp" to "Reference"

**Note:** To encourage software reuse, *Constellation* dictates that a work unit may contain at most a single component definition. Thus, when you choose to create a new component from a class, the wizard will place that component in a new work unit and move the original class to that work unit.

The wizard will create or update the component and create a “hyperlink” between the component and the class model element, as seen in Figures 7.7 and 7.8.

Figure 7.7 **Class Model Element after Implementation**

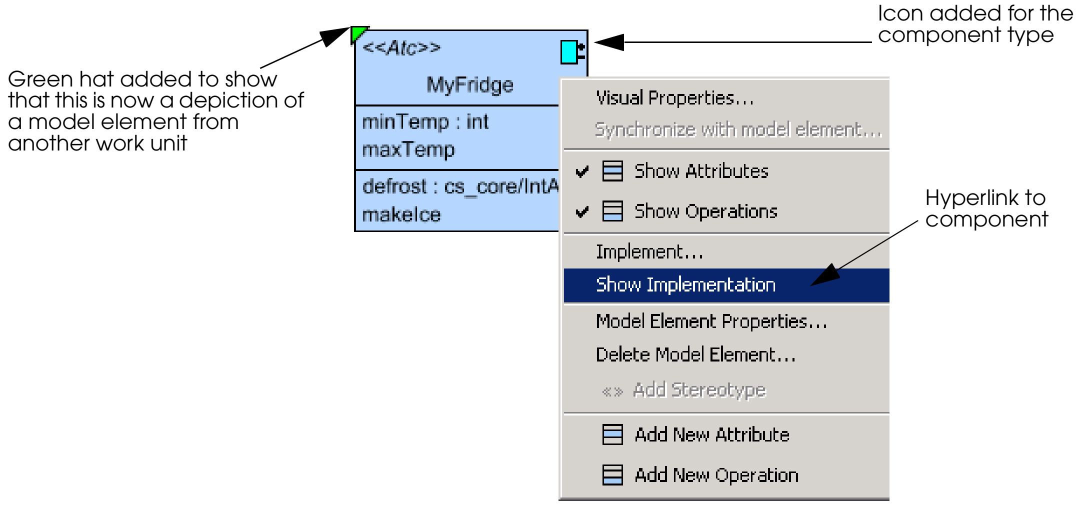
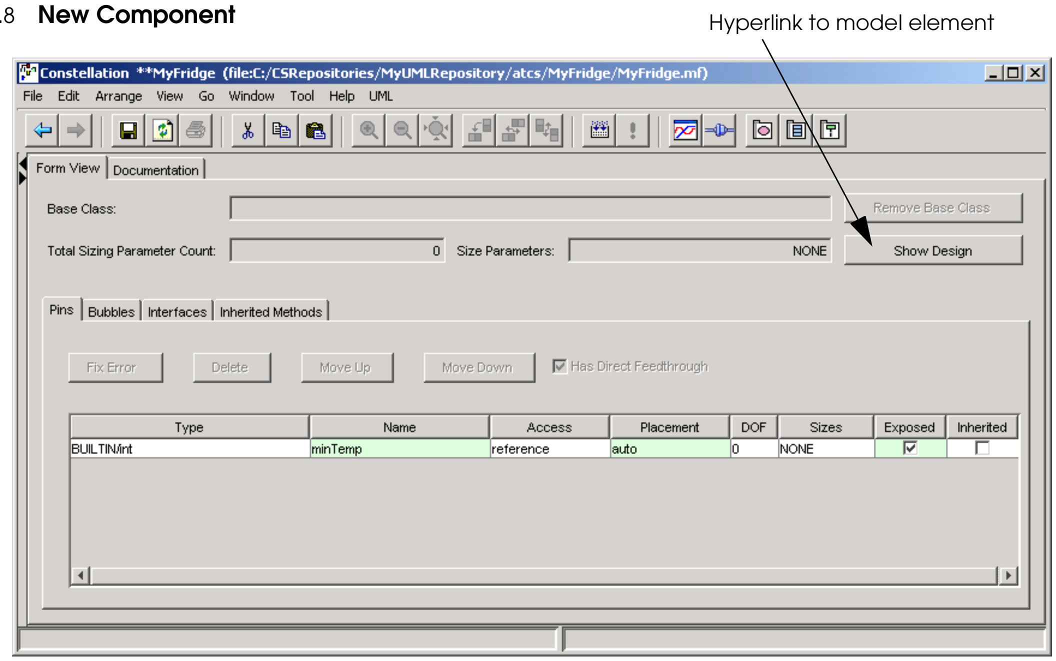


Figure 7.8 **New Component**



Base Class:

Total Sizing Parameter Count: 0 Size Parameters: NONE

☑ Has Direct Feedthrough

| Type | Name | Access | Placement | DOF | Sizes | Exposed | Inherited |
|---|---|---|---|---|---|---|---|
| BUILTIN/int | minTemp | reference | auto | 0 | NONE | ☑ | ☐ |

Right-click a depiction of the source class model element in any diagram or in the model browser and choose **Show Implementation** to load and display the component created from it.

Click the **Show Design** button in the component's form view to display the class diagram from which the wizard was last invoked.

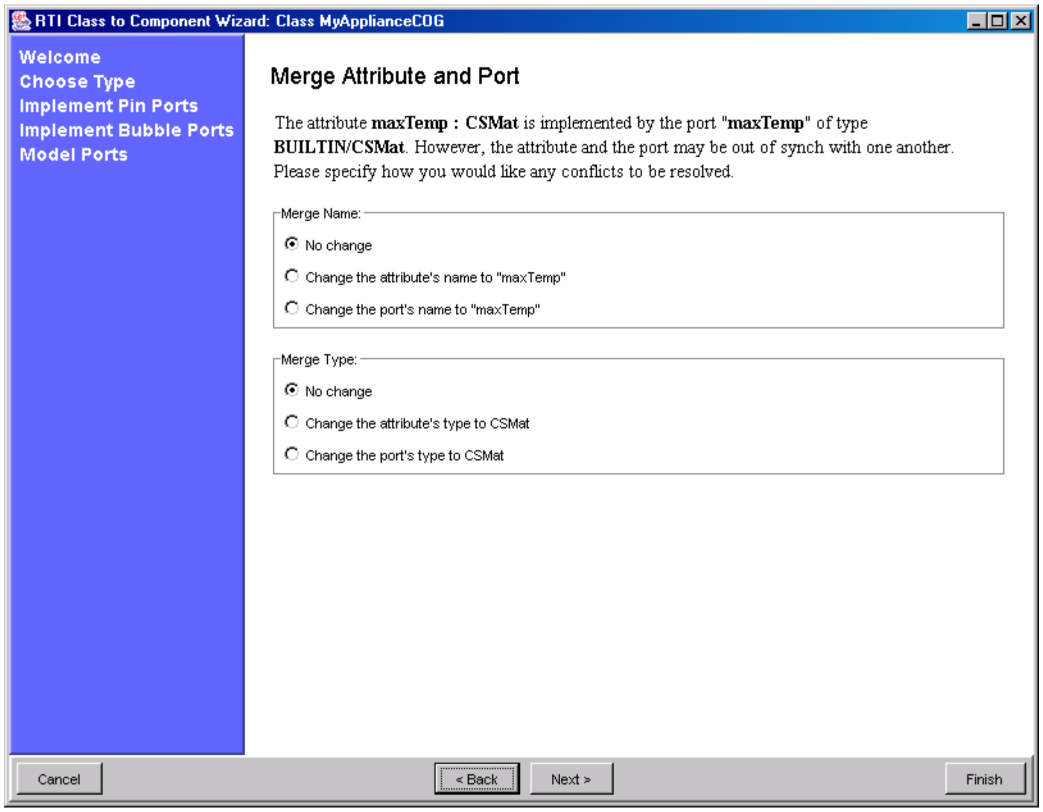
### 7.2.2 Synchronizing a Class and a Component

As time passes, your implementation and its original design may become out of sync. You may add new ports to the component, change the class's name, or add an attribute that wasn't originally there. To synchronize them again, invoke the wizard a second time: right-click the class and choose **Implement**. This time, you will see slightly different screens. As before, you will have the opportunity to create new attributes and operations for any new ports, and new ports that will correspond to any new attributes and operations. For those attributes, operations, and ports that are already associated because of a previous wizard invocation, you will have the opportunity to synchronize their names and types. Figure 7.9 shows an example.

### 7.2.3 Documenting an Existing Component

At some point, you may have a class diagram in which you need to include a component for which no corresponding class model element exists. "Reverse engineering" a component into a class is actually a special case of synchronizing a class and a component. Simply create a new class and immediately invoke the wizard on it. Rather than selecting the name and repository for a new component, enter the name and repository of your existing component. Then proceed as before.

Figure 7.9 **Merging**



Merge Attribute and Port

The attribute **maxTemp : CSMat** is implemented by the port "**maxTemp**" of type **BUILTIN/CSMat**. However, the attribute and the port may be out of synch with one another. Please specify how you would like any conflicts to be resolved.

Merge Name:

- [✓ No change]
- Change the attribute's name to "maxTemp"
- Change the port's name to "maxTemp"

Merge Type:

- [✓ No change]
- Change the attribute's type to CSMat
- Change the port's type to CSMat

# Index