Rational Software Corporation

Modeling Language Guide

RATIONAL ROSE® REALTIME

VERSION: 2002.05.20

PART NUMBER: 800-025112-000

WINDOWS/UNIX

Rational®
the software development company

support@rational.com
http://www.rational.com

Contents

Chapter 1 Modeling Language Guide 9

Real-time Notations to UML 10
Purpose 10
Building Blocks 10

Real-Time Systems 10
Support for real-time systems 11
Concurrency 11
Capsules and Ports 11
Capsule Structure Diagrams 12
Real-Time Specializations Overview 12
Executable Models 13
Services Library 14
Further Reading 15

Chapter 2 Elements 17

Structural 18
Behavioral 18
Grouping 18
Annotational 18

Use Cases 18

Actors 19

Flow of Events 20

Concrete and Abstract Use Cases 21

Use Case Instance 22

Use Case Packages 22

Use Case Concurrency 22

Classes 23

Interfaces 24

Attributes 25

Operations 25

Association Class 27

Utility Class 27

Instantiated Class Utility 28

Parameterized Class 28

Instantiated Class 29

Parameterized Utility Class 30

Capsules 30

Capsule Roles 33

Ports 34

Protocols 38

Cardinality and Capsule Structure 40

Substitutability 41

Multiple Containment 42

Actions, Messages, and Events 45

Actions 45

Call Event 46

Signal Event 46

State Machine 47
Events and signals 47
State machine variations 47
Overview 48

States 49

Transitions 51

Run-to-Completion 53

Initial Point and Initial Transition 54

Final State 55

Top State 55

History - Hierarchical State Machines 55

Group Transitions 57

Junction Points 57

Choice Points 61

Transition Selection Rules 63

Interactions 64

Components 65

Nodes 66

Packages 67

Notes 68

Chapter 3 Relationships 69
Real-time Notations 69

Association 70
Association Name 71
Association Ends 71
Association Multiplicity 71
Navigability 72
Aggregation 72

Composition 73
Visibility 74
Qualifiers 74
Constraints 75
Association Classes 75
Actor Communicates-Association 76
Connectors 76
Capsule Class Aggregation and Composition Relationships 77

Realization 79
Realization of Use Cases 79

Generalization 80
Actor Generalization 81
Include Relationship 81
Extend Relationship 82

Dependency 83
Component-Dependency Relationship 84

Chapter 4 Diagrams 85
Important visual relationships 85
Structure 86
Behavior 86
Real-time specialization 87

Use Case Diagram 87

Class Diagram 88

State Diagram 91

Collaboration Diagram 91

Capsule Structure Diagram 93

Sequence Diagram 95

Component Diagram 97

Deployment Diagram 99

Index 101

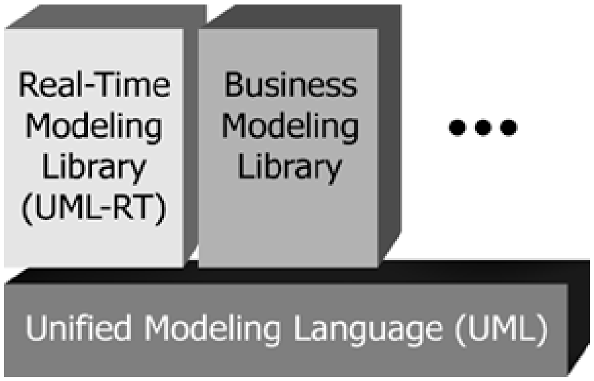
Chapter 1

Modeling Language Guide

The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, documenting, and executing software systems. Although conventional programming languages are good for expressing different algorithms, they cannot directly show the high-level features of a system. The UML is therefore a language for expressing high-level system properties that are best modeled graphically.

UML provides a base visual modeling language; however, it is not possible for the language to be sufficient for all domains. For this reason the UML has been designed open-ended to make it possible for the language to be extended. Without bloating the base language, new building blocks can be derived from the base to create ones that are specific to a domain.

Figure 1 Layered UML architecture


Real-Time Modeling Library (UML-RT)
Business Modeling Library
•••
Unified Modeling Language (UML)

Real-time Notations to UML

To address the category of systems that are characterized as complex, event-driven, and potentially distributed, a set of new modeling constructs has been added to a library of applied UML concepts. The notations that are based on the UML is primarily for the use of modeling the architectures of complex real-time systems.

Purpose

This document does not attempt to present the UML in its entirety. Rather, the goal of this document is to present, in detail, the real-time notations to the UML accompanied by a brief overview of the key UML modeling concepts you will need to understand when using Rational Rose RealTime.

Building Blocks

Building blocks include

- "Elements" on page 17 - are the basic object-oriented building blocks of the UML and real-time specializations. They are used to construct models.
- "Relationships" on page 69 - are used to join elements together in models.
- "Diagrams" on page 85 - help assemble related collections of elements together into a graphical depiction of all or part of a model.

Real-Time Systems

Models are important tools for developing complex systems. Models are used to represent the system at an abstract level, hiding unnecessary details. They enable communication about the composition and operation of a system. In Rational Rose RealTime, the model also serves as the basis for the implementation. All of the implementation details required to build an executable are either contained within the model, or generated from the model automatically.

Software developers have always used informal modeling techniques in the past, for example, drawing whiteboard diagrams and writing documents.

The evolution of the Unified Modeling Language (UML) has enabled developers to capture and communicate software designs with a common set of notations.

The UML uses visual notations to describe various views of an object model. Classes are the fundamental building blocks of this object model. The abstract structure, behavior and configuration of the software are described through diagrams. The implementation details of each class can be specified through various class properties.

Support for real-time systems

In addition to supporting the regular UML constructs, we have used the extensibility features of UML to define some new constructs that are specialized for real-time system development.

Concurrency

Most real-time systems must be capable of performing many simultaneous activities. External events are unpredictable, and the software must be able to handle interrupts and other external events at any time without dropping current work in progress.

Capsules and Ports

UML for Real Time provides built-in light weight concurrent objects, known as Capsules. Capsules are simply a pattern for providing light-weight concurrency directly in the modeling notation. A capsule is implemented in Rational Rose RealTime as a special form of class. Capsules allow the design of systems that can handle many simultaneous activities without incurring the high overhead of multitasking at the operating system level. In fact, many real-time projects end up hand-coding some kind of mechanism to handle multiple simultaneous transactions.

In addition to their concurrency properties, capsules are highly encapsulated and communicate through special message-based interfaces called Ports. A capsule sends and receives messages through its ports. The ports are in turn connected to other capsules, enabling the transmission of messages among capsules. The advantage of message-based interfaces is that a capsule has no knowledge of its environment outside of these interfaces, making it much more flexible and robust than regular objects.

Capsule Structure Diagrams

A new diagram has been introduced to specify the capsule's interface and its internal composition. The diagram is called a Capsule Structure Diagram, and is based on the UML 1.3 specification collaboration diagram. This is a specification type of diagram, and not an interaction diagram, as collaboration diagrams in other versions of Rational Rose are. The semantics around the capsule structure diagram allow Rational Rose RealTime to generate detailed code to implement the communication and aggregation relationships among capsules.

Real-Time Specializations Overview

Following is a list of the real-time specializations to the UML:

- "Capsules" on page 30
- "Protocols" on page 38
- "Ports" on page 34
- "Connectors" on page 76
- "Capsule Structure Diagram" on page 93

In addition to the new modeling elements, which are used during analysis and design, it was necessary to introduce new concepts and adapt others (components, processors) to support building, running, and debugging of models built in Rational Rose RealTime. Although you might be familiar with UML, it is worth while to understand the added concepts and how Rational Rose RealTime uses common UML constructs to build and deploy executable models:

- Using components to build a model

Observability options - Probes, Monitors, Message traces, Source breakpoints

Executable Models

The addition of the capsule and the formal semantics surrounding the capsule structure allows Rational Rose RealTime to generate, compile and run a complete C++ implementation based on a model containing capsules.

The ability to execute models has a revolutionary impact on the software development process. **The results are higher quality software, and shorter and more predictable delivery cycles.** Executing models is the surest way to find problems and issues that whiteboarding and document reviews do not find. Even high-level architectural models can be executed.

Use model execution to better understand the problem, to detect errors and problems in requirements and architecture specifications, to explore alternative designs quickly, and to test design models continuously during the development process.

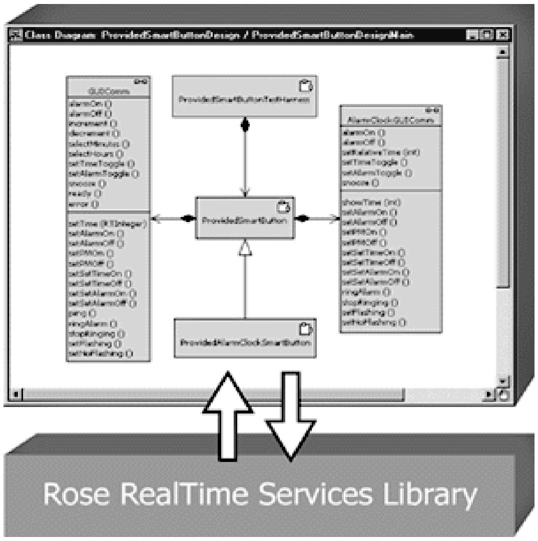
Process note: To make the best use of Rational Rose RealTime, you should aim to get your model running as often as possible. Making small, incremental changes and running your model each day will bring much better results than making widespread changes and working for weeks to get the model running again.

Services Library

To construct a Rational Rose RealTime model, two major parts are required:

- The structure and behavior of the model
- A Rational Rose RealTime Services Library (these are language specific)

Figure 2 The services library is a framework for real-time systems


Class Diagram: ProvidedSmartButtonDesign / ProvidedSmartButtonDesignMain
GUIComm
ProvidedSmartButtonTestHarness
AlarmClockGUIComm
ProvidedSmartButton
ProvidedAlarmClockSmartButton
Rose RealTime Services Library

The services library is essentially a framework for real-time systems. It includes functionality for controlling concurrency execution of finite state machines, for delivering messages, and for providing timing and logging services. A framework is a like a library of classes and operations used by an application, but with an inversion of control, meaning that the main control lies in the framework, and the framework invokes functions in the application to pass control to application objects as required. Application classes are subclassed from framework classes so that they inherit operations which are invoked by the framework.

There is no main() function in a Rational Rose RealTime model. The main() function is contained in the services library and takes care of creating the capsules in your model and kicking off the execution of their state machines. All you need to do is describe the capsules and define state machines for them and they will be automatically created and executed by the services library. The capsule state machines can, in turn, invoke operations on other classes (data classes), and send messages to other capsules. The services library is responsible for managing the creation and destruction of capsules, and the delivery of messages between capsules (even across threads).

The addition of the real-time specializations to the UML concepts allows the toolset to generate complete code for the model which is tied in to the services library. When you generate code for and compile a model in Rational Rose RealTime, the tool will link it with a services library compiled for the particular platform you are running on.

Further Reading

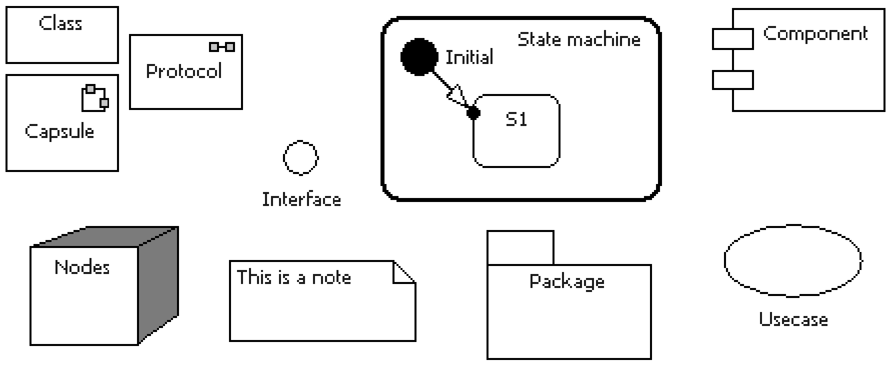
The details of the Services Library routines that can be called from within your model are described in the Language Add-in Language References.

Chapter 2

Elements

Elements are the basic object-oriented building blocks of the UML, and real-time notations. They are used to construct models. There are four kinds of elements: Structural, Behavioral, Grouping, and Annotational.

Figure 3   Elements overview



Structural

The structure of a system identifies the entities that are to be modeled. The primary relationships captured between structure elements are communication and containment relationships.

UML base	Classes, Interfaces, Use Cases, Components, Nodes
Real-time notations	Capsules

Behavioral

These elements represent the dynamic parts of the model that describe the changing state of a system over time.

UML base	State Machine, Interactions
Real-time notations	Protocols

Grouping

These are organizational parts of a model. There is only one kind of grouping element in UML.

- Packages

Annotational

These provide common ways of describing or annotating any element in a diagram.

- Notes

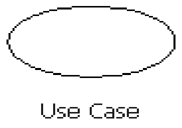
Use Cases

A use case is a description of a set of sequences of actions, called scenarios, that a system performs to yield an observable result of value to an actor. A use case describes what a system (subsystem, class, or interface) does but does not specify how the system internally performs its tasks. This is left for the use case realizations to show.

Use cases can be concrete or abstract. See “Concrete and Abstract Use Cases” on page 21.

Graphical notation

The graphical notation for a use case in a use case diagram is an ellipse.



Other features

Use cases can have attributes and operations that you may represent just as for classes. Since use cases are classifiers, you can also attach state machines, interaction diagrams, sequence diagrams and class diagrams as more ways of describing the behavior and scenarios described by a use case.

Relationships

Type	From a use case to a(n)
Include Relationship	use case
Diagrams	use case
Actor Generalization	use case
Actor Communicates-Association	Actors

Actors

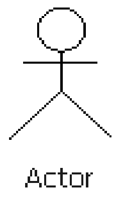
An actor is anything that exchanges data with the system. An actor can be a user, external hardware, or another system.

The difference between an actor and an individual system user is that an actor represents a particular class of user rather than an actual user. Several users can play the same role, which means they can be one and the same actor. In that case, each user constitutes an instance of the actor. Since actors represent system users, they help delimit the system and give a clearer picture of what it is supposed to do.

An actor specification is identical to a class specification with the addition of the stereotype field set to actor.

Graphical notation

The graphical notation for an actor is a stickman.



Relationships

Type	From an actor to a(n)
generalization-relationship	actor
Actor Communicates-Association	Use Cases

Flow of Events

The flow of events of a use case contains the most important information derived from use case modeling work. It should describe the use case's behavior clearly enough for an outsider to easily understand. Here are some guidelines for the contents of the flow of events:

1. Describe how the use case starts and ends.
2. Describe what data is exchanged between the actor and the use case.

3. Do not describe the user interface.
4. Detail the flow of events. Remember that test designers are to use these to identify test cases.

Structure

The two main parts of the flow of events are **basic flow** of events and **alternative flows** of events. The basic flow of events should cover what “normally” happens when the use case is performed. The alternative flows of events cover behavior of optional or exceptional character in relation to the normal behavior, and also variations of the normal behavior. You can think of the alternative flows of events as “detours” from the basic flow of events, some of which will return to the basic flow of events and some of which will end the execution of the use case.

Documenting

A flow of events document should be created for each use case. The format of the document can vary depending primarily on how formal they are. The Rational Unified Process provides templates for documenting flow of events. Alternatively, SoDA can be used.

Concrete and Abstract Use Cases

There is a distinction between concrete and abstract use cases. A concrete use case is initiated by an actor and constitutes a complete flow of events. “Complete” means that an instance of the use case performs the entire operation called for by the actor.

An abstract use case is never instantiated in itself. Abstract use cases are included in including, extending, or generalizing other use cases. When a concrete use case is initiated, an instance of the use case is created. This instance also exhibits the behavior specified by its associated abstract use cases. Thus, no separate instances are created from abstract use cases.

The distinction between the two is important because it is concrete use cases the actors will “see” and initiate in the system.

You indicate that a use case is abstract by writing its name in italics.

Use Case Instance

A use case instance is a specific flow of events through a use case. Many flows of events are possible and many may be similar. Related flows of events should be grouped into one use case.

Use Case Packages

A model structured into smaller units is easier to understand. It is easier to show relationships among the model's main parts if you can express them in terms of packages. A package is either the top-level package of the model, or stereotyped as a use case package. You can also let the customer decide how to structure the main parts of the model.

If there are many use cases or actors, you can use case packages to further structure the use case model. A use case package contains a number of actors, use cases, their relationships, and other packages; thus, you can have multiple levels of use case packages (packages within packages). The top-level package contains all top-level use case packages, all top-level actors, and all top-level use cases.

Use Case Concurrency

Instances of several use cases and several instances of the same use case work concurrently if the system permits it. In use case modeling, you can assume that instances of use cases can be active concurrently without conflict. The design model is expected to solve this problem, because use-case modeling does not describe how things work. One way to view this is to assume that only one use case instance is active at a time and that executing this instance is an atomic action. In use case modeling, the “interpreting machine” is considered infinitely fast, so that serialization of use case instances is not a problem.

Classes

A class is a design-time specification for one or more distinct objects with common structure, attributes, behavior, and operations. At run-time there are instances of classes, referred to as objects. Use the implementation language syntax and semantics when specifying Operations and Attributes.

Structure

The structural features of a class are defined by its attributes.

Behavior

An object can react differently to a specific message depending on what state it is in. You can specify the behavior of a class by drawing a State Diagram. For each state the object can enter, the diagram shows what messages it can receive, what operations will be carried out, and what state the object will be in thereafter.

Persistence

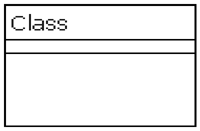
A persistent class represents instances where their state will be preserved when the instance is destroyed. Instances created from a transitory class have their state destroyed when the instance is destroyed.

Standard stereotypes of classes

- Utility Class
- Instantiated Class
- Parameterized Class
- Parameterized Utility Class
- Instantiated Class Utility
- metaclass

Graphical notation

A class icon is drawn as a three-part box, with the class name in the top part, a list of attributes (with optional types and values) in the middle part, and a list of operations (with optional argument lists and return types) in the bottom part.



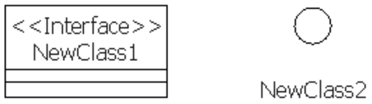
The attribute and operation sections of the class box can be suppressed to reduce detail in an overview. Suppressing a section makes no statement about the absence of attributes or operations, but drawing an empty section explicitly states that there are no elements in that part.

Interfaces

An interfaces is a model element that defines a set of behaviors (a set of operations) offered by a classifier model element. A classifier may realize one or more interfaces. An interface may be realized by one or more classifiers. Any classifiers that realize the same interfaces may be substituted for one another in the system. Each interface should provide an unique and well-defined set of operations.

Graphical notation

An interface is shown graphically as a circle with its name. Or it can be shown as a class with the <<interface>> stereotype.



Attributes

An attribute is a named property of an object. The attribute name is a noun that describes the attribute's role in relation to the object. An attribute can be scoped to a class or an instance and configured with a specific visibility. In addition you can specify its type, initial value, and multiplicity.

You should model the property of an object as an attribute only if it is a property of that object alone. Otherwise, you should model the property with an association or aggregation relationship to a class whose objects represent the property.

Changeability properties

Determines whether a value may be modified after the object is created. The possible values are

- changeable - no restrictions or modification
- frozen - the value may not be altered after the object is instantiated and its values initialized; no additional values may be added to a set
- add only - meaningful only if the multiplicity is not fixed to a single value; additional values may be added to the set of values, but once added, a value in the set cannot be removed or altered

Operations

An operation is a service that can be requested from an object to affect its behavior. The only way other objects can get access to or affect the attributes or relationships of an object is through its operations. The operations of an object are defined by its class. A specific behavior can be performed via the operations, which may affect the attributes and relationships the object holds and cause other operations to be performed. An operation corresponds to a member function in C++ or to a function or procedure in Ada.

Class or instance

An operation nearly always denotes object behavior. An operation can also denote behavior of a class, in which case it is a class operation. This can be modeled in the UML by modifying the scope of the operation.

Operations Have Parameters

In the specification of an operation, the parameters constitute formal parameters. Each parameter has a name and type. You can use the implementation language syntax and semantics to specify the operations and their parameters so that they will already be specified in the implementation language when coding starts. Use the implementation language syntax and semantics when specifying operations.

Properties

- abstract - if an operation is abstract, then it does not have an implementation defined, and one must be supplied by a descendant
- leaf (polymorphic) - if an operation is a leaf, then the operation cannot be overridden by a descendant; if false, the operation is polymorphic, and the implementation of the operation can be overridden by a descendant—that is, a leaf operation is mapped to a C++ a non-virtual operation)
- query - if an operation is query, then the execution of this operation leaves the state of the system unchanged, meaning that the operation has no side effects

These next properties address the concurrency semantics of an operation.

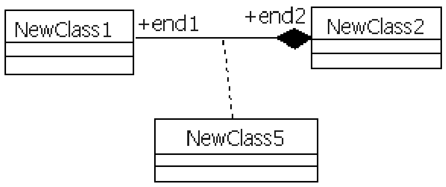
- sequential - callers of this operation must coordinate outside the object so that only one flow of control is in the object at a time; if simultaneous calls occur, then the semantics and integrity of the system cannot be guaranteed
- guarded - multiple calls from concurrent flows of control may occur simultaneously, but only one is allowed to commence; the others are blocked until the performance of the first operation is complete
- concurrent - multiple calls from concurrent flows of control may occur simultaneously, and all of them may proceed concurrently

Association Class

Use the association class to model properties of associations. The properties are stored in a class and linked to the association relationship. Link Attributes are degenerate association classes comprised only of attributes. To create an association class, create an association and an association class. Connect the association class to the association.

Graphical notation

An association class is a class linked to an association.

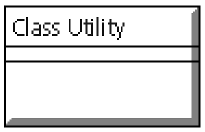


Utility Class

A utility class specifies a class whose attributes and operations are all class scoped. An instantiated utility class represents an instance of a utility class.

Graphical notation

The class utility is shown with the same three compartments as a class.

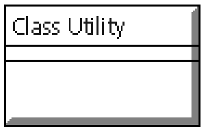


Instantiated Class Utility

An instantiated class utility is created by substituting actual values for the formal parameters of a parameterized class utility.

Graphical notation

An instantiated class utility is displayed as utility class.



Parameterized Class

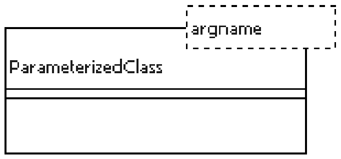
A parameterized class is a template for creating any number of instantiated classes that follow its format. A parameterized class declares formal parameters. You can use other classes, types, and constant expressions as parameters. You cannot use the parameterized class itself as a parameter. You must instantiate a parameterized class before you can create its objects.

In its simplest form, you use parameterized classes to build container classes.

You can also use parameterized classes to capture design decisions about the protocol of a class. The arguments of the parameterized class can be used to import classes or values that export a specific operation. In this form, a parameterized class denotes a family of classes whose structure and behavior are defined independently of its formal class parameters.

Graphical notation

A parameterized class is a class icon with a dashed-line box in the upper right corner. The parameters are automatically displayed there.



Instantiated Class

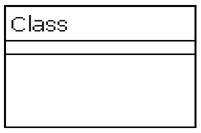
An instantiated class is a class formed from a parameterized class by supplying actual values for parameters. You create an instantiated class by supplying actual values for the formal parameters of the parameterized class. This instantiation process forms a concrete class in the family of the parameterized class. You must place the instantiated class at the client end of an instantiate relationship (accessible using **Create > Instantiated Class** on the **Tools** menu) that points to the corresponding parameterized class.

An instantiated class whose actual parameters differ from other concrete classes in the parameterized class' family forms a new class in the family.

To create an instantiated class, create a class and enter the class name with parameters in brackets to distinguish it from other forms of classes.

Graphical notation

An instantiated class is depicted as a class.

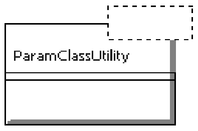


Parameterized Utility Class

A parameterized utility class is a set of operations or functions that are not associated with a higher level class and are defined in terms of formal parameters. Use a parameterized class utility as a template for creating instantiated class utilities. You must instantiate a parameterized class utility before you can create its objects.

Graphical notation

A parameterized class utility is a class icon with a dashed-line box in the upper right corner and a gray shadow at the lower edge of the rectangle. The parameters are automatically displayed there.



Capsules

Capsules are the fundamental modeling element of real-time systems. A capsule represents independent flows of control in a system. Capsules have much of the same properties as classes; for example, they can have operations and attributes. Capsules may also participate in dependency, generalization, and association relationships. However, they also have several specialized properties that distinguish them from classes.

The main characteristics that specializes capsules from other classes are summarized below:

Classes have...	Capsules have...	Details
public operations	public ports	Sending messages through public ports is the only method that capsules can communicate with other capsules. Classes cannot invoke operations directly on other capsules. Capsules can call operations on classes, but since a capsule does not have public operations classes cannot call operations on capsules.
public, protected, and private attributes	private attributes	A class's structure is defined by its attributes. The same goes for capsules; however, this structure is completely private, in the sense that no outside object can directly access these attributes. Capsules can have the following kinds of attributes: capsule roles, protected ports, and classes. ***Note:*** *The only public attributes of a capsule are its public ports.*
operation invocation	message passing	Messages are the sole means of communication between capsules. Messages are sent and received through ports.
behavior defined by methods	behavior defined by state machines	The action that a class performs when an operation is invoked as defined by the implementation of the operation. However, when a capsule receives a signal event the behavior is controlled by its state machine.

Structure

A capsule may have any number of attributes that define its structure or none at all. These attributes represent some properties of the capsule class that are shared by all instances. What differentiates a capsule from a class is how you can formally specify the internal organization of its structure, as a network of collaborating capsule roles. This collaboration is a specialized UML collaboration called a capsule collaboration.

Behavior

The behavior of a class is triggered by the invocation of a public operation on the class. Whereas, a capsules behavior is triggered by the receipt of a signal event.

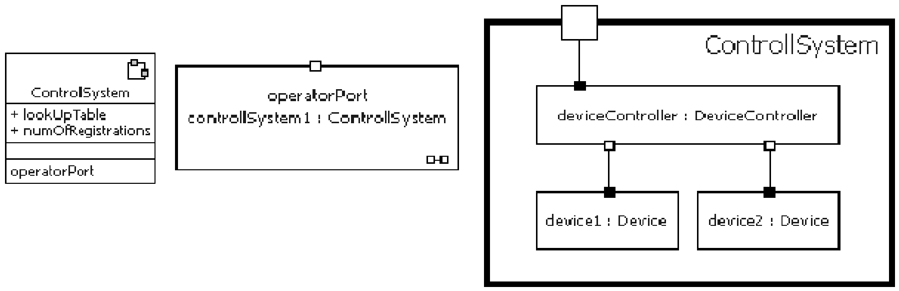
When a capsule receives a message from another capsule a signal event is generated and some response by the capsule is usually required. This typically involves performing some calculations, formulating a response, and sending one or more messages. The optional state machine associated with a capsule represents its behavior. It controls the operation of the capsule itself. The state machine is the only element that can access the protected parts of the capsule.

Logical threads of control

Capsules provide a very light weight modeling element for breaking a problem down into multiple threads of control. Each capsule instance has its own logical thread of control, though it may share an actual processing thread (known as a “physical thread”) with other instances.

Graphical notation

Since a capsule is a stereotype of a class, the stereotype icon appears in the name compartment of the class rectangle.



Relationships

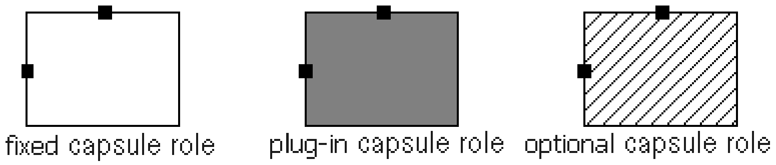
Type	From a capsule to a
generalization-relationship	capsule (class view)
aggregation & composition-relationship	capsule, protocol (class view)
dependency-relationship	class, capsule (class view)
association	class (class view)
connector	capsules (capsule collaboration view)

Capsule Roles

Capsule roles represent a specification of the type of capsules that can occupy a particular position in a capsule's collaboration, or structure. Capsule roles are strongly owned by the container capsule, and cannot exist independently of the container capsule. A capsule's structural decomposition usually includes a network of collaborating capsule roles joined by connectors.

Classification of capsule roles

Note that the classifications below are attributes of capsule roles rather than an attribute of the capsule classes that fill the role. This maximizes the reuse potential of capsule specifications.



- Fixed - By default, capsule roles are fixed, meaning that they are created automatically when their containing capsule is created, and are destroyed when the container is destroyed.

- Optional - Some capsule roles in the structure may not be created at the same time as their containing capsule. Instead, they may be created subsequently, when and if necessary, by the state machine of the capsule. And they can be destroyed before the container is destroyed.
- Plug-in - The structure of a capsule may contain plug-in capsule roles. These are, in effect, placeholders for capsule roles that are filled in dynamically. This is necessary because it is not always known in advance which specific objects will play those roles at run time. Once this information is available, the appropriate capsule instance (which is owned by some other composite capsule) can be “plugged” into such a slot and the connectors joining its ports to other capsule roles in the collaboration are automatically established. When the dynamic relationship is no longer required, the capsule is “removed” from the plug-in slot, and the connectors to it are taken down.

Cardinality

You can specify the cardinality of capsule roles as a shorthand structural method of grouping multiple copies of the same type of capsule role in a graphically compact and reusable pattern.

Substitutability

Optional and plug-in capsule roles can be designated as substitutable.

Ports

Ports are objects whose purpose is to send and receive messages to and from capsule instances. They are owned by the capsule instance in the sense that they are created along with their capsule and destroyed when the capsule is destroyed. Each port has its identity, which is distinct from the identity and state of their owning capsule instance.

Ports and Protocols

To specify which messages can be sent to and from a port, a port is associated with a protocol role. The protocol role is the specification of a set of the messages that can be received (in) and sent (out) from the port. The protocol role essentially defines the port type.

A protocol is the specification of communication patterns between capsules. In any communication scenario the exchange of messages (those being sent and those being received) are different depending from which end of the communicating participants you chose to view the exchange. A protocol specifies all the views of a communication, and each of these different views are what we call the protocol roles. Since a port plays the role of one participant in a communication relationship it's valid message sets (in and out) are defined by those in a specific protocol role. When creating a port, you must specify which participant (protocol role) this port will play in the protocol. Currently, Rational Rose RealTime only supports binary protocols, which involve just two participants, or two protocol roles. These roles are called the 'base' role and the 'conjugate'.

Communication Rules

In order for two ports to be connected by a connector, the ports must be compatible; that is, every signal in the 'Out' set of one protocol role must be in the 'In' set of the other protocol role. Each protocol role can have additional signals in the 'In' set; however, there can not be any signals in the ‘Out’ set that are not in the corresponding 'In' set on the other side. In other words, the set of 'In' signals on both sides of two connected ports must be equal to - or be a superset of - the set of 'Out' signals on the other side. In addition, the data class of an 'Out' signal must be the same as - or a subclass of - the data class of the corresponding 'In' signal of the other port.

Classification of Ports

Visibility

- Public - Public ports are ports that are part of a capsule’s interface. These ports are shown in a capsule collaboration diagram as being located on a capsules boundary. Public ports may be visible both from outside the capsule and inside.

- Protected - Protected ports are used to connect capsules to contained capsule roles. These ports are not visible from the outside of a capsule since they are not part of the capsule's interface.

Connector type

- Wired - Wired ports must be connected by a connector to other ports in order to send messages. In a capsule's structure these are the ports that are graphically connected to other ports.
- Non-wired - Non-wired ports are used to model dynamic communication channels. These ports cannot be connected with connectors to other ports. Unlike wired connections, which are established when a capsule is created and disconnected when destroyed, non-wired connections can be dynamically controlled. This is useful for modeling client/server designs where a shared service is shared by a large number of clients, and the clients are not known at design time.

Termination

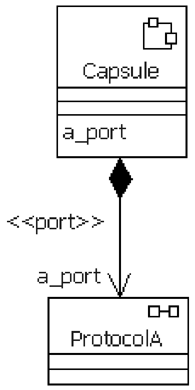
- Relay - Relay ports are by nature implicitly public and wired. They are used to model connections that funnel signal events directly to protected capsule components without being processed by the capsule itself. If a relay port is not connected to an internal component, all signal events arriving on that port are lost. Generally speaking, relay ports can be used to export the interfaces of contained capsule roles.
- End - End ports can be public or protected, wired or non-wired. Messages sent to an end port can be processed directly by the capsule's behavior. End ports are the ultimate destination of all signal events sent by capsules. These signals are generated in the state machines and received by state machines.

Graphical notation

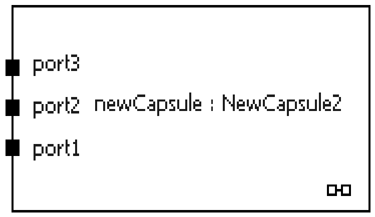
The notation for a port uses white and black squares to indicate which protocol role (base or conjugate), the port plays in a protocol. The white square is used to show conjugated ports.

Class view - in a class diagram, capsule ports are listed in a special labeled list compartment, which normally appears after the attribute and operator compartments. In addition, a stereotyped <<port>> association shows the relationship between the capsule and a protocol.

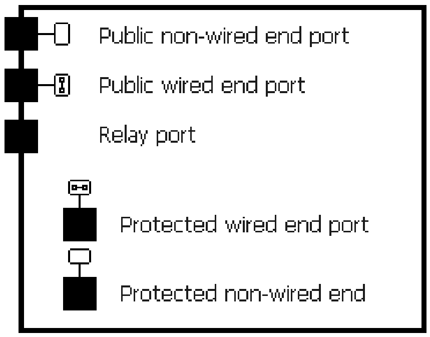
Note: *From the class view, you cannot tell the role of the port in the protocol.*



Capsule role view - only public ports are shown on capsule roles. Externally there is no distinction between relay and end ports. The name of the port and protocol role (base or conjugate) is displayed.



Capsule collaboration view - all ports are visible in the structure view. There physical placement, either on the capsules container or inside, differentiates between protected and public ports. End ports are shown with a line connected to a circle. The following illustration shows the symbols for the different port types.



Protocols

The set of messages exchanged between two objects conforms to some communication pattern called a protocol. It is basically a contractual agreement defining the valid types of messages that can be exchanged between the participants in the protocol. Therefore a protocol comprises a set of participants, each of which plays a specific role in the protocol.

Each such protocol role is specified by a unique name and a specification of messages that are received by that role as well as a specification of the messages that are sent by that role (either set could be empty). As an option, a protocol may have a specification of the valid communication sequences; a state machine may specify this. Finally, a protocol may also have a set of prototypical interaction sequences (these can be shown as sequence diagrams). These must conform to the protocol state machine, if one is defined.

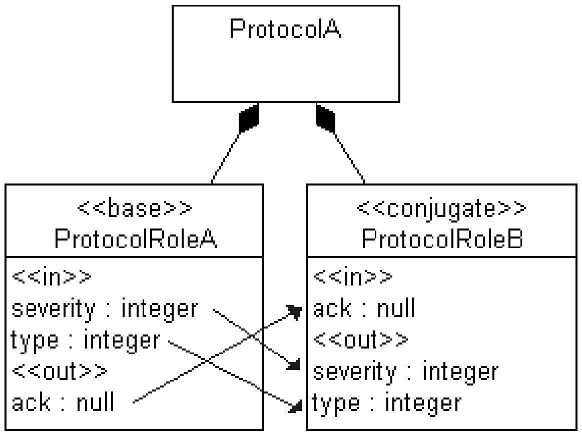
Protocol participants (protocol roles)

The fact that a protocol role is defined by what is “sent” and what is “received” implies that a protocol role is specified from the perspective of only one of the participants in a protocol. Thus, a protocol is composed of the different protocol roles, or perspectives of some communication pattern.

Binary protocols

Binary protocols, involving just two participants, are by far the most common and the simplest to specify. One advantage of these protocols is that only one role, called the base role, needs to be specified when defining the protocol. The other side of the communication pattern, called the conjugate, can be derived from the base role simply by inverting the incoming and outgoing message sets. This inversion operation is known as conjugation.

Figure 4   A binary protocol as a composite of two roles



When working with binary protocols there is no need to explicitly define the conjugate role.

Ports and protocols

Protocols are primarily used to identify the type of a port. Ports play the role of one participant in a communication relationship, so technically the type of a port is specified by the protocol role and not the protocol.

Graphical notation

A binary protocol can be shown using the standard notation for classifiers with an explicit stereotype label and two optional specialized list compartments for incoming and outgoing signal sets, as shown in Figure 4. The state machine and interaction diagrams associated with a protocol are represented using the standard UML notation.

Cardinality and Capsule Structure

You can specify the cardinality of capsule roles and ports as a shorthand structural method of grouping multiple copies of the same type of element in a graphically compact and reusable pattern.

Note: *Cardinality is an attribute of the capsule role and not of the capsule class. The decision of how many instances of a capsule are needed is not a property of the capsule, rather it is based on the needs of the application.*

We often refer to capsule roles or ports that have a specified cardinality as being replicated, meaning that there are several copies.

Cardinality rules for capsule roles

- fixed - all instances of a replicated fixed capsule role are created automatically when the container capsule is created.
- optional - instances of replicated optional capsule roles are created dynamically. Their number can vary from zero (0) up to the specified cardinality.
- plug-in - these slots are filled in dynamically. Their number can vary from zero (0) up to the specified cardinality.

Cardinality and ports

In order to connect replicated capsule roles, use cardinality to group ports on a capsule that share a common protocol. The cardinality factor on a port determines the number of instances of a port.

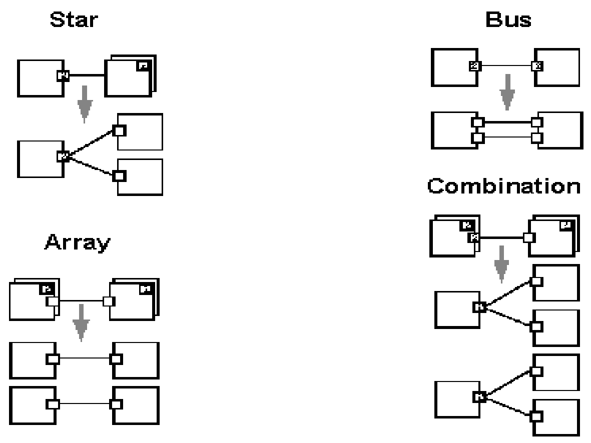
Unspecified cardinality

Cardinality values do not have to be specified. They can be left open ended by using the asterisk character '*'.

Common structural patterns

Different capsule collaboration capsule role configurations can be achieved using combinations of cardinality applied to both ports and capsule roles.

Figure 5 Structural patterns with cardinality



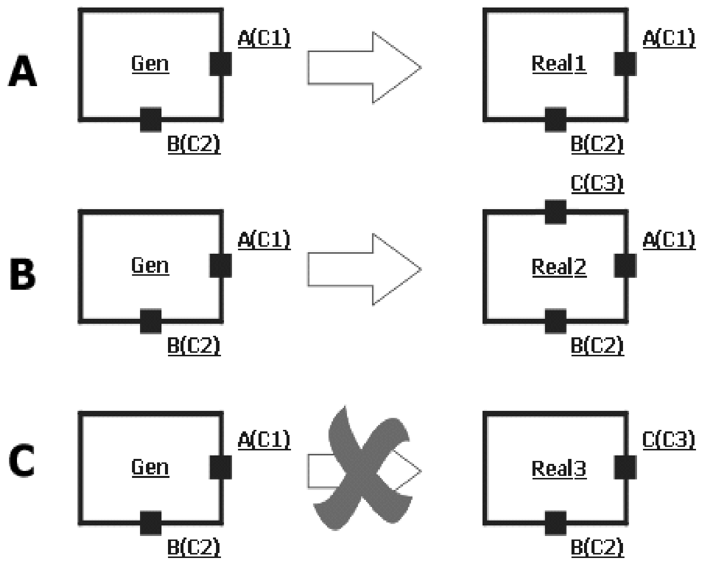
Substitutability

Substitutability is a property of an optional or plug-in capsule role. It allows the capsule role to be instantiated with either an instance of the capsule class represented by the capsule role or any other compatible capsule class. Two capsules are compatible if they have the same interface, or are subclasses of the same superclass. Since a capsule's interface is defined by its public ports, compatible capsules must have matching public ports.

Substitutability rules

To be compatible with its superclass, a subclass must have a matching compatible port for every connected interface of the superclass reference. In the following example we use the notation A(C1) to represent a port A of type protocol C1. Let **Gen** be a capsule role belonging a generic (abstract) class in some design and let Real1, Real2, and Real3 be capsule roles belonging to subclasses of this class. We illustrate the compatibility rules with three different cases in Figure 6.

Figure 6 Compatibility rules



Multiple Containment

Multiple containment allows you to represent capsule roles that are simultaneously part of two or more capsule collaborations. Specifying that two different capsule roles are actually bound to the same run-time instance can simplify the structure of the system by allowing it to be decomposed into different views. Figure 7, “Multiple containment example," on page 44 shows an example design of a telephone switching system, which has a **VoiceCallHandler** capsule that is responsible for processing calls, and a **MaintenanceSystem** capsule that is responsible for maintaining system integrity, both of which contain the same **voiceLineHandler** capsule role.

When to use multiple containment

In order to understand the need for this, it is necessary to examine the meaning of encapsulation in object-oriented design in general. When two or more capsule roles are placed together in a common capsule, the intent is to capture some user-defined relationship between these components. The simplest example of a relationship between objects is pure physical containment; for example, a shelf contains a particular card. When we move into the domain of software, however, the types of relationships that exist can be quite diverse. In communications, for instance, when two terminals are connected to each other in order to exchange information, they are involved in a call relationship. The object-oriented approach encourages us to capture such identifiable relationships as distinct objects. Note that, in physical terms, there is no real entity corresponding to a call; however, it is quite useful to think of it in that way.

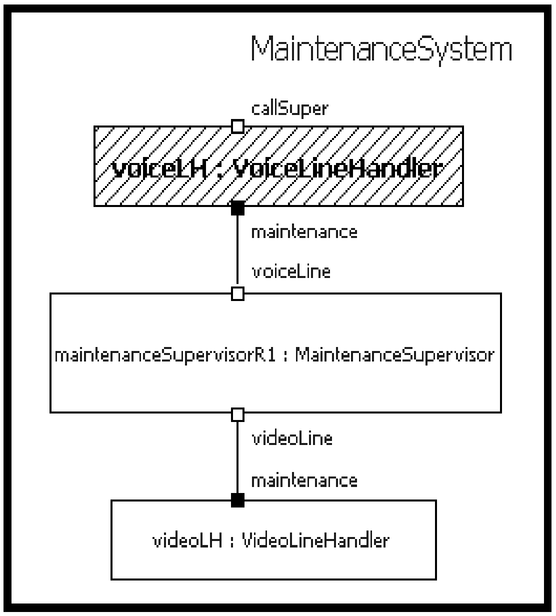
Once relationships such as these are captured in unique addressable objects, then it is possible to conceive of operations over such objects, such as terminating a particular call or adding another party to it. To the entities invoking the operations, the structure and implementation within such objects are typically of no concern. Following this line of thought leads us to conclude that these objects are in fact like any other software objects: entities with a set of externally accessible operations and an encapsulation shell that hides their internals. Therefore, capsules can be used to represent arbitrary user-defined relationships between their component actors.

With this explanation of capsules, the need for multiple containment is more apparent. It is required to capture situations where a capsule role is involved in multiple simultaneous relationships with capsule roles in another containment. In our communications example, a terminal can simultaneously be involved in a call and be part of a device subsystem.

Multiple containment example

Each port on a capsule can only be used in one of the decomposition views. For example, in the example below, if the **callSuper** port on the voiceLineHandler were used in both the **VoiceCallHandler** and **MaintenanceSystem**, it would lead to a conflict, so it is not allowed.

Figure 7 Multiple containment example


MaintenanceSystem
callSuper
voiceLH : VoiceLineHandler
maintenance
voiceLine
maintenanceSupervisorR1 : MaintenanceSupervisor
videoLine
maintenance
videoLH : VideoLineHandler

Actions, Messages, and Events

There are several ways for objects to communicate with each other. Regardless of the mechanism used there is always someone sending the message, a sender, and someone receiving the message, a receiver. When two objects communicate something must be passed between them. The thing that is passed between two objects must convey what the sender wants done, pass some optional information the receiver might need to complete the requested behavior, and also specify the way in which the thing will be communicated from the sender to the receiver.

- Messages - A message is a specification of the *thing* that will be communicated between two objects. A signal is a special type of message that is sent asynchronously. An instance of a message can be called a stimulus or message instance. However, message instance and message are commonly used interchangeably.
- Actions - An action represents the sending, or dispatching of a message by a *sender*. The type of action can specify the type of request (for example, invoke operation, or signal event to be raised).
- Events - An event represents the reception of a message by the receiver. When the message is received an event is raised in the *receiver* object. There are two kinds of events: call events and signal events.

Graphical notation

The communication between objects is modeled in interaction diagrams by linking the sender and receiver with an association.

Actions

Actions are the things the behavior does when a transition is taken. They represent executable atomic computations that are written as statements in a detail-level programming language and incorporated into a state machine. Actions are atomic, in the sense that they cannot be interrupted by the arrival of a higher priority event. An action therefore runs to completion.

Actions may be composed of any number of operation calls, creation or destruction of other objects, or the sending of messages to other objects.

An action may be attached to the following parts of a state machine:

1. A transition (including a transition to an initial state)
2. A state, as an entry action
3. A state, as an exit action

Blocking actions

Because of run-to-completion semantics, any action that blocks—for example, by invoking a synchronous operation call—will effectively block the entire state machine from handling any new events that may arrive. Further event processing will only resume once the action has completed.

Call Event

A call event represents the reception of a request to synchronously invoke a specific operation.

Flow of control

A call event is sent synchronously, meaning that when an object invokes an operation on another, control passes from the sender to the receiver. Once the receiver has finished processing the event, the operation returns, and control is returned to the sender.

Signal Event

A signal event represents the reception of a particular asynchronous message. A signal event may be sent by the action of a state transition in a state machine. The execution of an operation can also send signals.

The receipt of a signal event usually triggers a state transition in a state diagram.

Flow of control

When an object sends a signal event, it is sent asynchronously, meaning that control does not transfer to the receiver. Once the signal event is sent, the sending object can continue its action and start processing other events, instead of being blocked until the receiver has finished processing the sent event.

State Machine

State machines are used to model the dynamic aspects of a system. Whereas interactions model a set of roles that work together to perform some common behavior, a state machine models the behavior of a particular class, protocol, or capsule. State machines can be used to describe a use-case flow of events, or to completely specify the behavior of classes that receive events.

Events and signals

Event-driven means that behavior is stationary until it is activated by the arrival of an anticipated signal on one of its interfaces. After responding to an event, the process reverts to a stable state in which it is ready to receive the next event. An event is generated whenever a message is received by an object. An event is an occurrence that may cause the state of an object to change; it has no duration and can precede or follow another event. A message can be used to convey either information or data values from one object to another.

State machine variations

Because state machines are used to describe the behavior of several different kinds of elements, there are some small differences between what is allowed in each one:

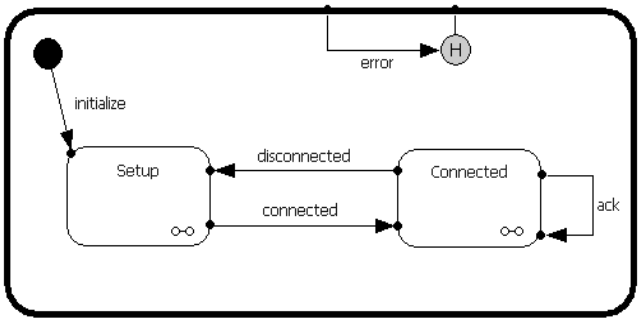
- Class and Use Case state diagrams - All trigger events are simple uninterpreted text.
- Protocol state diagrams - All trigger events are signals defined on the protocol.

- Capsule state diagrams - All trigger events are defined by a port and signal pair.
  - Limitation: final states are not allowed.
  - Limitation: junction points do not support the continuation kind attribute; that is, if a transition is not continued, it defaults to history (except for internal transitions)

Overview

A state diagram is a directed graph of states connected by transitions. A state diagram describes the life history of objects of a given class. A state machine contains exactly one initial state and initial transition, one top state, one or more states, choice points, and the state transitions between them.

Figure 8 Example state diagram



States

A state is a condition during the life time of an object where it is ready to process events. A state machine is essentially composed of a top state, which can itself contain any number of other states. A state has the following parts:

- Name - A state must have a name so that it can be distinguished from other states that are in the same context.
- Entry/Exit actions - Actions that are executed on entering and exiting the state - an entry action is executed whenever a state is entered, regardless of which incoming transition was taken. Similarly, an exit action is taken whenever we leave the state from whatever outgoing transition.

Hierarchical states

A state can be composed of other states, called **substates**. This allows modeling of complex state machines by abstracting away detailed behavior into multiple levels.

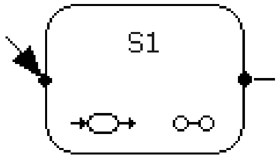
States that do not contain substates are called simple states. A state that has substates is called a composite state. States may be nested to any level.

Graphical notation

States can be viewed from two different perspectives: an external view and an internal view. The external view shows a state as a substate of its container state machine. The external view is in effect an abstract view of the state. The internal view shows a states internal details — its implementation.

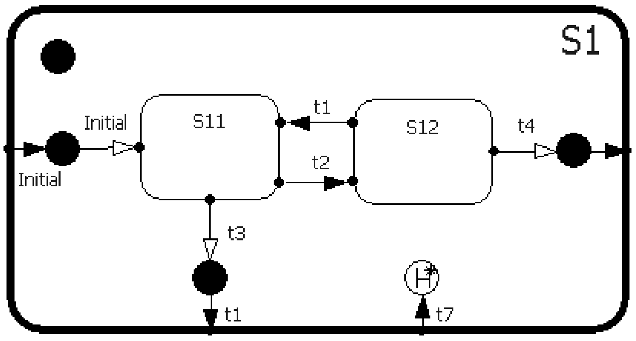
External view

In this view a state is shown as a rounded rectangle with a name compartment. Optional graphical clues can be shown at the bottom of the state rectangle to show that a state is a composite state and to show if entry or exit actions have been defined. All junction points are shown as solid dots. Details regarding the state transitions, whether they terminate or continue within the composite state, are not shown.



Internal view

In this view you see the details of the composite state. The states border is seen as a bolder rounded rectangle that encapsulates its internal details. Within this view you can see if incoming transitions from the external view end on the state, go to history, or continue to other substates.



Transitions

A transition is a relationship between two states: a source state and a destination state. It specifies that when an object in the source state receives a specified event and certain conditions are met, the behavior will move from the source state to the destination state.

A transition has the following parts:

Trigger

With the exception of the initial transition all behavior in a state machine is triggered by the arrival of events on one of an object's interfaces. Therefore, a trigger defines which events from which interfaces will cause the transition to be taken.

The trigger is associated with the interface on which the triggering event is expected to arrive. Moreover, a transition can have multiple triggers such that an event that satisfies any one of the triggers will cause the transition to be taken.

Guard Condition

Each trigger can have a boolean expression associated with it which will be evaluated before the transition is triggered. This is referred to as a guard condition. If the expression evaluates to True, then this trigger will cause the transition to be taken; if the expression evaluates to False, the transition is not taken. If no guard condition is specified, the condition defaults to True. Guard conditions are coded using a detail level language which must evaluate to a boolean result, although it can contain an arbitrarily complex expression.

Actions

The actions in a behavior are where an object does work. For example, it can perform operation calls, create and destroy other objects, and sends signals to other objects. It is important to understand that a transition cannot be interrupted by the arrival of an event. A transition is therefore said to run-to-completion.

Kinds of transitions

- **Normal transitions** - transitions that originate and terminate on different states.

Following are three kinds of self-transitions that are characterized by originating and terminating on the same state, having no continuing segments, and not ending on a continuing junction point.

- **Inner self transitions** - Where the exit and entry code is not executed for the state on which it originates and terminates.
- **Inner internal self transitions** - Where the transition executes without exiting or re-entering the state in which it is defined. And in addition the exit and entry actions of all states which where exited and re-entered are not executed. These kinds of transitions are similar to having global operations defined on a state machine; when taken do not change the state of the system.
- **External self transitions** - Where the exit and entry code is executed for the state on which it originates and terminates.

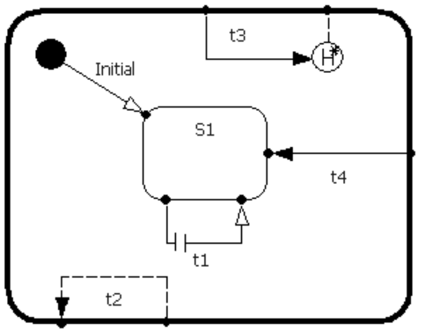
Junction points

Junction points provide a way of correlating different segments of a transition that span multiple hierarchical contexts. They are located on the boundary of a state, and represent either the source or destination of a transition segment.

All transitions originate and terminate on a junction.

Graphical notation

A transition is rendered as a solid directed line from a source state to a target state. It can be decorated in different ways: a black arrow head indicates that actions have been associated with the transition; a broken transition line indicates that no triggers have been defined for the transition; a dashed line indicates an internal self-transition.



Run-to-Completion

The processing of a single event at a time by a state machine is known as a run-to-completion step. Events are dispatched and processed by the state machine, one at a time.

Simplifying concurrency

If preemption were allowed in a state machine, the handling of a high-priority event could possibly modify some internal variables that were in the process of being modified as a result of the previous low priority event. When the low priority processing is resumed these variables would be changed, leading to errors.

With the run-to-completion approach, handling of the current event does not allow any interruptions, even by the arrival of higher priority events - hence, avoiding the internal concurrency problem. The advantage of this model is simplicity. The biggest disadvantage is that processing of events cannot take too long to ensure a timely processing of higher priority events.

Initial Point and Initial Transition

An initial point is a special point which explicitly shows the beginning of the state machine. You connect the initial point to a start state. Where the start state will be the first active state in the objects state machine. The transition from the initial point to the start state, initial transition, is the first transition taken before any other transition. Only one initial state is allowed in each state diagram.

Note: *The transition from an initial point to the start state can have an action; however, the other transition features, including a guard condition and trigger event, are not allowed.*

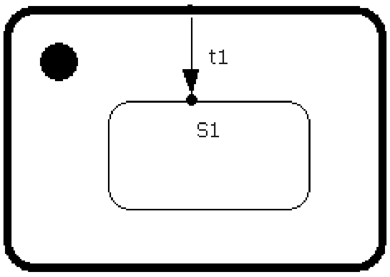
Transitions to and from the initial state

Only one outgoing transition can be placed from the initial point.

There can be several incoming transitions to the initial state. In this case the initial state acts like a junction point which forces the behavior back through the initial transition. If the initial transition is used to completely initialize an object, then any incoming transition to the initial state will effectively reset the behavior of an object without having to destroy then re-create it.

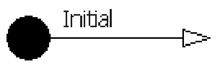
No initial transition

A state machine does not require an initial transition. In the example below, when the state machine is created we are in the Top state until the triggering event for t1 is received.



Graphical notation

The initial state icon is a small filled circle:



Final State

A final state if a special kind of state signifying that the enclosing composite state is completed. If the enclosing state is the top state, then it means that the entire state machine has completed. A final state cannot have any outgoing transitions.

Final states cannot be added to a capsule state diagram.

Top State

The top-level state that is the root of the state machine containment hierarchy. There is exactly one state in every state machine that is the top state.

The top state cannot have exit and entry actions, or outer self-transitions.

History - Hierarchical State Machines

The history of a state is defined as the substate that was the last current substate the last time the state was active. In the case of simple states, they are always the last active state.

History is useful when dealing with situations where an event takes control away from the current state and initiates a separate behavior sequence for handling the new event. The new sequence can involve new states and transitions. However, once completed, we often want to resume from the point before the interruption occurred.

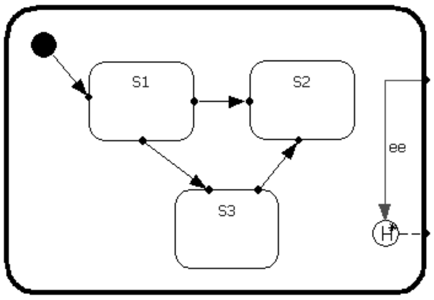
Continuation kinds - shallow history, deep history and default

When a transition terminates on shallow history, the active substate becomes the most recently active substate prior to this entry. Whereas, deep history implies returning to history at all state hierarchy levels.

Example

A common use of history is shown in the state machine in Figure 9. In this case transition **ee** is a self-transition that has a trigger for an event that none of the substates can handle. When that event occurs the self-transition will fire then go to history, meaning that it will revert to the last active substate. The effect is to perform event handling without changing the state of the system.

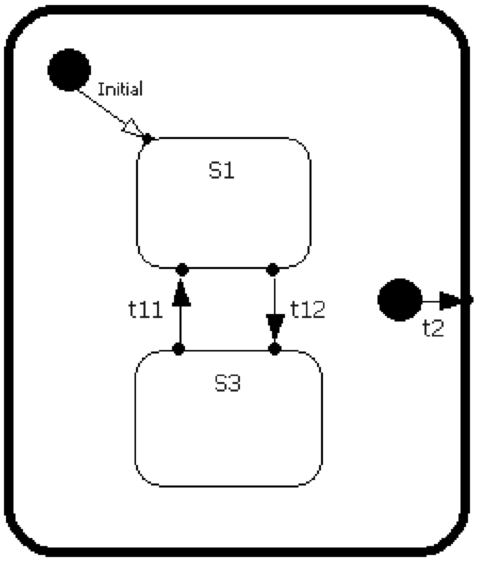
Figure 9 An example use of history



Note: *Beware of how entry and exit actions are called when the ee transition is taken. For example, if the current active state is S2, when ee is triggered, the exit action for S2 will be taken, then the actions for ee will execute, and finally the entry action for S2 will be executed.*

Group Transitions

Group transitions are transitions from hierarchical states that are common to all the substates within that state. Thus common behavior that is normally represented by equivalent transitions from every state, can be represented by a single transition originating from the containing hierarchical state.



Junction Points

Junction points are located on the boundary of states, and represent either the source or the destination of a transition segment. Junction points are split into those that terminate on the state boundary, history junctions, and those which are notations that continue within the state. Similarly transitions outgoing from a hierarchically nested state are divided into those that terminate on the enclosing state and those that continue from the state boundary to a target state.

Note: *Every transition, whether composed of multiple segments or of a single segment, eventually terminates to a junction point.*

Transition segments

Transitions that span multiple hierarchies, thus cross state boundaries, change context on the way from the source to the destination state. Therefore they must be partitioned into different segments. Each transition segment has a distinct name, and only the originating segment has a trigger defined. The sum of all transition segments is called the transition chain.

Note: *Transition chains are executed in one single run-to-completion step.*

Joining transition segments

Two or more transition segments can converge on to a single junction point. This allows multiple transitions to be defined, which perform the same action in response to an event, and share the same destination state. In the example below t4 and t9 both terminate on the same junction point.

Note: *Only the originating transition segments can have triggers defined. For example, although t6 can have actions, it cannot have a trigger.*

Continuation

When a transition terminates on a composite state (there are no other transitions continuing from the junction point), the behavior of the state machine at this point is determined by the **Continuation kind** property of the terminating junction point.

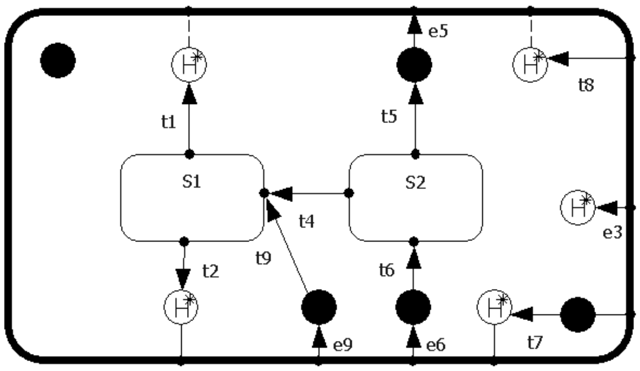
This selection specifies the semantics for how the state history will be used when there is no continuing transition. There are three options:

- Default - specifies that the default (initial) transition should be taken.
- History - specifies that the state should return to shallow history.
- Deep History - specifies that the state should return to deep history, meaning that all substates also return to history. This is the behavior for all capsule state machines, so it is automatically selected.

Note: *The default for capsule state machines is to always go to deep history, so deep history will be automatically selected for capsule states, and the selections will be grayed out.*

Graphical notation

Junction points are displayed differently depending whether a state is shown from an abstract view, as a substate of another containing state, or whether shown from a detailed view, the inside of a composite state. Junction points viewed from the abstract state view are always shown as a solid dot. However, from the detail view, different junctions are shown with graphical clues.



Continuing junctions

Type of junction	Example	Shown as...
internal	t8 originates from an internal junction. The junction is not visible from the abstract view of this state.	a small circle on the state boundary
external	t7 originates from an external junction. This junction is visible from the external view of the state so that it can eventually be connected to. Until an external junction has been connected it behaves like an internal junction.	a solid circle connected with a solid line to the container state's boundary.
external incoming	t6 originates from an external incoming junction. The label of the transition segment that is attached to the junction is shown as e6. t6 cannot have a trigger.	a solid circle connected with a solid line to the container state's boundary with an arrow at the circle.
external outgoing	t5 terminates on an external outgoing junction. The next transition segment is shown as e5. t5 has a trigger.	a solid circle connected with a solid line to the container state's boundary with an arrow at the container state boundary.

Terminating junctions (to history)

Type of junction	Example	Shown as...
internal	t8 terminates on an internal terminating junction. This junction is not visible from the external view of the state.	a circle with the letter 'H' (for history) connected to the container states boundary with a dotted line.
external	t7 terminates on an external terminating junction. This junction is visible from the external view of the state. If this junction is ever used to connect another transition segment, it will become a continuing junction, since it will no longer terminate a transition.	a circle with the letter 'H' (for history) connected to the container states boundary with a solid line.
external incoming	e3 terminates on an external terminating junction. This junction is visible from the external view of the state.	a circle with the letter 'H' (for history) connected to the container states boundary with a solid line with an arrow at the circle.

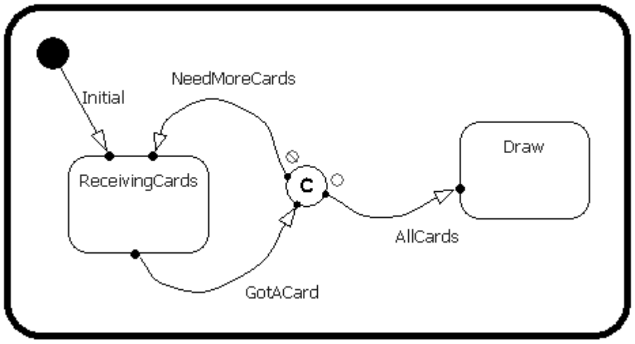
Choice Points

Choice points allow a single transition to be split into two outgoing transition segments, each of which can terminate on a different state. The decision of which branch to take is made after the transition is taken.

Choice points are motivated mainly by practical considerations: it often happens that the decision on which state to terminate a transition can only be made following certain calculations. Each choice point has an associated boolean predicate that is evaluated after the incoming transition action is executed. Depending on the truth value of this predicate, one or the other branch is taken.

Example

A very common use of choice points is to count events. That is when the decision of transferring from one state to the next depends on the number of events that have occurred. For example, if a player can only draw cards once he has received 5 cards from a dealer, you could model this behavior the following way:

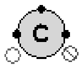


The player would keep track of how many cards he has received and every time a new card is received would test if he has enough cards.

Graphical notation

A choice point is rendered as a circle with one incoming point, and two outgoing for the true and false transitions. The choice point is shown with a 'C' in the middle if the boolean predicate is defined

.



Transition Selection Rules

When an event is ready to be processed by a state machine, a search for a candidate transition takes place to determine which one will be taken. A transition is said to be enabled if its trigger is satisfied by the current event, meaning that the transition has the same event and interface specified as the current event, and the guard condition evaluates to true.

The search order is defined by the following algorithm:

1. The search begins in the innermost current active state.
2. Within the scope of the innermost current active state, transitions are evaluated sequentially. If a transition is enabled, the search terminates and the corresponding transition is taken.
3. If no transition is enabled in the current scope, the search in step 2 is repeated for the next higher scope, one level up in the state hierarchy.
4. If the top-level state has been reached and no transitions are enabled, then the current event is discarded and the state of the behavior remains unchanged.

When a transition is enabled the algorithm continues with the following:

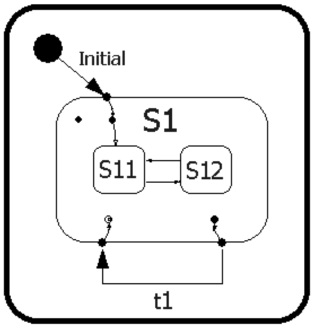
1. If the enabled transition is not an internal transition, then execute the exit actions of all substates starting with the deepest history up to the current scope.
2. If the enabled transition is an internal transition, then none of the substates exit and entry actions are executed.
3. Execute the enabled transition (this includes all the transition segments) actions. This chain ultimately terminates on a simple state. Note that executing the transition may include executing other state entry and exit actions as well.
4. The terminating simple state becomes the current active state.

Example

In the following example t1 is a group transition outside of S1. If S11 is the current active state when the current event enabled transition t1, then the following actions would be executed as part of the transition chain.

1. Exit action for S11
2. Exit action for S1
3. Action code for t1
4. Entry action for S1
5. Entry action for S11

S11 remains the current active state after the transition is complete.



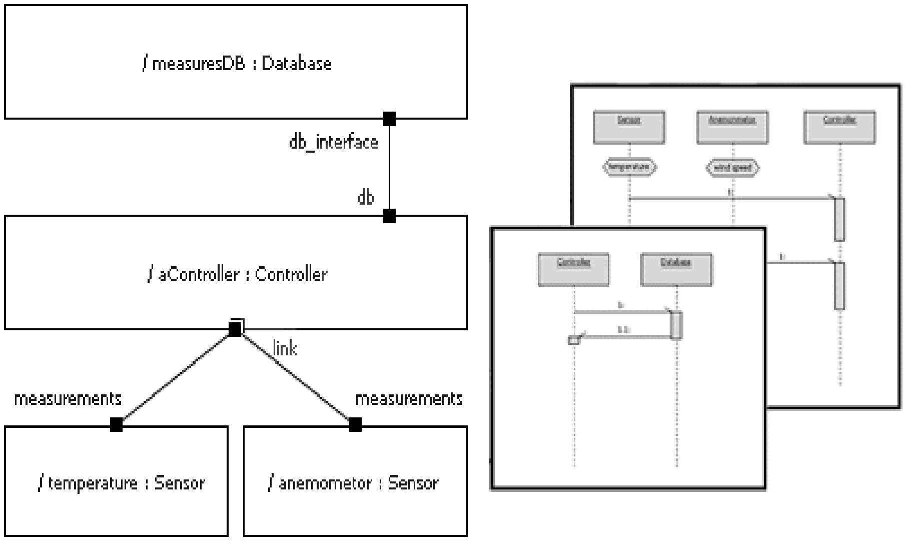
Interactions

An interaction is a behavior that consists of a set of messages exchanged among a set of objects or roles to accomplish a specific purpose. Interactions can be given in two different forms: either a specification level (showing classifier roles, association roles, and messages) or at the instance level (showing objects or instances, links, and stimuli). Interactions are important to modelers because they clarify the roles things play in a particular scenario, and thus provide input for determining interfaces (protocols and the public operations of a class).

Interactions model the dynamic aspects of a model. Interactions can be modeled in two views: in sequence diagrams and in collaborations.

Collaborations are the constraining element to a set of sequences. The sequences show all the different communication scenarios that can occur between the instances or roles in the collaboration.

Figure 10   A collaboration with the set of sequences that define the different scenarios of the collaboration



Components

Components are used to model the physical elements that may reside on a node, such as executables, libraries, source files, and documents. The component, therefore, represents the physical packaging of the logical elements, such as classes and capsules.

Mapping from logical to physical

The mapping from design - that is, classes and capsules - to source code and executables is not an easy task. It is during this phase that the majority of errors are introduced into a system. In addition, there is always the risk that the implementation diverges from the original design. However, since the UML is a well-formed language, with the help of tools, models can be automatically generated into a lower level language and compiled into an executable. With automatic compilation of your design, the model becomes the system.

In the context of automatic executable generation, a component is used to configure sets of design elements that are to be generated and built. In addition, several configuration parameters relating to the generation of the executable, such as dependencies and compiler preferences, are maintained by the component.

Component instances

A component can have instances. An instance of a component can be a single executable, or a library that can reside on a number of different nodes. To allow for this, specific component instances can be assigned to nodes. Component instances are not shown in the component diagram. They can only be shown in the deployment diagram.

Organization

Components can be organized by placing them in packages.

Relationships

Type	From a component to a(n)
dependency-relationship	component
realization-relationship	interface

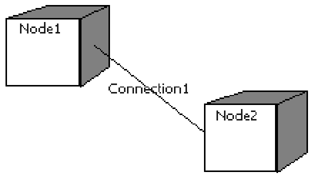
Nodes

Nodes represent physical devices, more specifically, a computational resource having memory and processing capability. Component instances reside and run on nodes. Use nodes to model the topology of the hardware on which your system executes.

Connections

The most common relationship between nodes is an association. In the context of deployment, an association represents a physical connection between nodes.

Figure 11 A simple deployment diagram



Packages

A design package is a collection of classes, relationships, use-case realizations, diagrams, and other packages. It is used to structure the design model by dividing it into smaller parts. Packages are used primarily for model organization.

Note: *Although packages can serve as a unit of configuration management, for pragmatic reasons it is preferable to manage versioning at the class level, than at the package level. Rational Rose RealTime supports class level versioning.*

The big picture

The design model can be structured into smaller units to make it easier to understand. By grouping design model elements into packages, then showing how those groupings relate to one another, it is easier to understand the overall structure of the model.

Package Content Visibility

A class contained in a package can be public or private. A public class can be associated by any other class. A private class can only be associated with classes contained within the same the package.

A package interface consists of a package's public classes. The package interface (public classes) isolates and implements the dependencies on other packages. In this way, parallel development is simplified because you can establish interfaces early on, and the developers need to know only about changes in the interfaces of other packages.

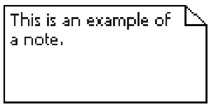
Notes

A note captures the assumptions and decisions applied during analysis and design. Notes may contain any information, including plain text, fragments of code, or references to other documents. A note holds an unlimited amount of text and can be sized accordingly.

Notes behave like labels. They are available on all diagram toolboxes. Notes only show up where they have been placed on diagrams. They are not considered part of the model. Notes may be deleted like any other item on a diagram.

Graphical notation

The shape of a note is a rectangle with a folded edge in the upper right hand corner.



Relationships

A note may be unconnected, meaning that it applies to the diagram as a whole. You can also attach a note via a note anchor to any item or items that can be selected in a diagram.

Chapter 3

Relationships

Most often model elements must collaborate with other elements in a number of ways. Relationships allow representation of how elements stand in relation to others.

There are four main kinds of relationships in the base UML:

- "Association" on page 70 - represent structural relationships between elements
- "Realization" on page 79 - represent the relationship between an interface and its implementation
- "Generalization" on page 80 - link generalizations with their specializations
- "Dependency" on page 83 - represent using relationships between elements

The relationships expressed above are in simple form. However, they also have a number of properties that allow them to be used to model relationships with an increased level of detail. This increased level of detail is necessary in order for total source code generation of a model.

Real-time Notations

In addition to the base UML relationships, the real-time specialization is a specialized association role relationship:

- "Connectors" on page 76 - is a specialized association role that captures the static communication relationships between capsule roles.

Association

An association is a structural relationship that is used to connect one element to another. Associations can be used during analysis to initially identify general relationships between classes. As your model evolves, you will add additional properties to associations to make them more specific.

Association roles

The relationships between roles in a collaboration diagram are called association roles. These define the required communication links between the roles in a collaboration.

Graphical notation

The graphical notation of an association will depend on the amount of detail that has been added to describe it.

Figure 12 A simple association relationship

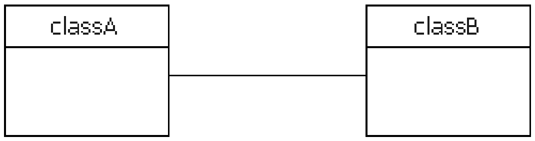
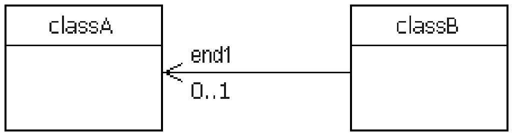


Figure 13 Association with role name, multiplicity,& navigation



Additional properties

You can assign a variety of additional properties to association relationships. They include

- association name
- association ends
- association multiplicity
- navigability
- aggregation
- composition
- visibility
- qualifiers
- constraints
- association classes
- Actor communicates-association
- Connectors

Association Name

You can name associations to describe the nature of the relationship between the elements it links. Although relationships can have names, you won't necessarily need to include one if the association includes association ends.

Association Ends

The end of an association where it connects to an element is called an association end. End names can be used instead of association names to describe the role an element plays in the relationship.

Association Multiplicity

It defines the number of objects that participate in an association relationship. There are two multiplicity indicators for each association, one at each end.

Multiplicity is a specification of the range of allowable cardinalities that a set may assume. The multiplicity is written as a range or as an explicit value.

1	Exactly one
0..*	Zero or more
1..*	One or more
0..1	Zero or one
7..9	Specific range
6	A specific number

Navigability

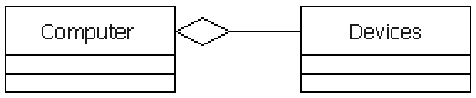
The navigability property on an association end indicates that it is possible to navigate from a associating class to the target class using the association. This may be implemented in a number of ways: by direct object references, by associative arrays, hash-tables, or any other implementation technique that allows one object to reference another. Navigability is indicated by an open arrow, which is placed on the target end of the association line next to the target class (the one being navigated to). The default value of the navigability property is true.

Aggregation

Aggregation is a special form of association that specifies the whole-part relationship between an aggregate (whole) and the component (part). There are many examples of aggregation relationships: an Elevator contains Doors, within a company Departments are made-up of Employees, a Computer is composed of a number of Devices. To model this, the aggregate (Elevator) has an aggregation association to its constituent parts (Doors).

Graphical notation

A hollow diamond is attached to the end of an association path on the side of the aggregate (the whole) to indicate aggregation.



Composition

If there is strong inter-dependency relationship between the aggregate and the parts—where the definition of the aggregate is incomplete without the parts—then a composition should probably be used instead of a plain aggregation.

Aggregate or association?

Aggregation should be used only in cases where there is a composition relationship between classes, where one class is composed of other classes, where the "parts" are incomplete outside the context of the whole. Consider the case of an order: it makes no sense to have an order which is "empty" and consists of "nothing". The same is true for all aggregates: Departments must have Employees, Families must have Family Members, and so on.

If the classes can have independent identity outside the context provided by other classes, and if they are not parts of some greater whole, then the association relationship should be used. In addition, when in doubt, an association may be more appropriate. Aggregations are generally obvious, and choosing aggregation is only done to help clarify. It is not something that is crucial to the success of the modeling effort.

Composition

Composition is a form of aggregation with strong ownership and coincident lifetime of the part with the aggregate. The multiplicity of the aggregate end may not exceed one (i.e. it cannot be shared). The aggregation is also unchangeable, that is once established, its links cannot be changed. By implication, a composite aggregation forms a “tree” of parts, with the root being the aggregate, and the “branches” the parts.

A composition should be used over “plain” aggregation when there is strong inter-dependency relationship between the aggregate and the parts; where the definition of the aggregate is incomplete without the parts.

Graphical notation

A solid filled diamond is attached to the end of an association path to indicate composition. In this example, the Customer Interface is composed of several other classes. In this example the multiplicities of the aggregations are also specified.

A CustomerInterface object knows which Receipt Printer, Keypad, and Speaker objects belong to it.

Visibility

There are circumstances in which you will want to limit the visibility of the association relative to elements outside the association.

The visibility property can be used to control the visibility of elements owned by packages and the visibility of the features (attribute or operation) of a classifier. Possible values are

- Public - public access specifies that any outside classifier with visibility to classifier can use the feature. This is the default visibility.
- Protected - protected access means that classifier features are accessible only to descendants, friends, or to the classifier itself.
- Private - private access means that the features of the classifier are accessible only to the classifier itself.

Qualifiers

Qualifiers are used to further restrict and define the set of instances that are associated to another instance; an object and a qualifier value identify a unique set of objects across the association, forming a composite key. Qualification usually reduces the multiplicity of the opposite end; the net multiplicity shows the number of instances of the related class associated with the first class and a given qualifier value.

Qualifiers are drawn as small boxes on the end of the association attached to the qualifying class. They are part of the association, not the class. A qualifier box may contain multiple qualifier values; the qualification is based on the entire list of values. A qualified association is a variant form of association attribute.

Constraints

The basic constructs of associations, are usually sufficient in describing most structural relationships you will encounter. But they cannot describe them all. In UML, you can use constraints, on association ends and associations, to capture important conditions of the association. The constraint is an expression of some semantic condition that must be preserved while the system is in a steady state.

Graphical notation

Constraints are shown in curly braces '{', '}'.

Implementing constraints

In practice, the constraints identified in the model should be verified in your system. To be valuable, constraints shouldn't simply be a modeling aid. An approach to implementing constraints is to use assertions (if your programming language supports them) to verify post conditions, pre-conditions, and invariants.

Association Classes

An association class is an association that also has class properties (such as attributes, operations, and associations). It is shown by drawing a dashed line from the association path to a class symbol that holds the attributes, operations, and associations for the association. The attributes, operations, and associations apply to the original association itself. Each link in the association has the indicated properties. The most common use of association classes is the reconciliation of many-to-many relationships (see example below). In principle, the name of the association and class should be the same, but separate names are permitted if necessary. A degenerate association class just contains attributes for the association; in this case you can omit the association class name to de-emphasize its separateness.

Actor Communicates-Association

Use cases and actors interact by sending signals to one another. To indicate such interactions we use a communicates-association between use case and actor. A use case has at most one communicates-association to each actor, and an actor has at most one communicates-association to each use case, no matter how many signal transmissions there are. The complete network of such associations is a static picture of the communication between the system and its environment.

Communicates-associations are not given names. Because there can be only one communicates-association between a use case and an actor, you need only specify the start and end points to identify a particular communicates-association.

Connectors

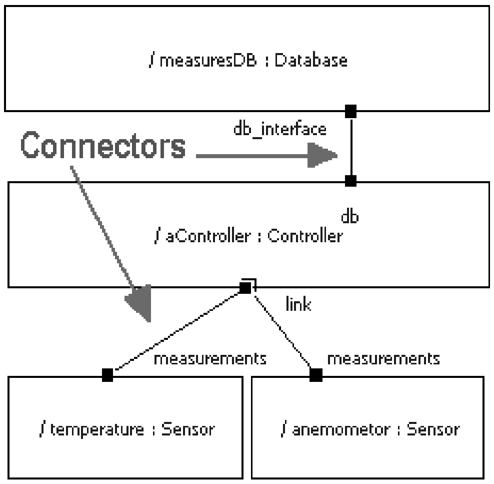
Connectors really capture the key communication relationships between capsule roles. They interconnect capsule roles that have similar public interfaces, which are called ports. A key feature of connectors is that they can only interconnect compatible ports.

Connectors only exist in the context of a capsule collaboration.

Graphical notation

A connector is shown as a line between ports in a collaboration diagram.

Figure 14 A capsule collaboration shown with 3 capsule roles connected with connectors



Capsule Class Aggregation and Composition Relationships

Relationships between capsule classes

Capsule roles in a class diagram are shown by composition relationships between capsule classes. Depending on the attributes of the capsule role (fixed, optional, or plug-in), the aggregation can be shown as a composition. For example plug-in and optional capsule roles are shown with aggregation relationships whereas fixed capsule roles are shown with composition. The capsule role name is shown as the end name of the association. Cardinality of the capsule role is also displayed.

Relationships between capsule classes and protocol classes

Ports can also be modeled in the class diagram using a stereotyped <<port>> composition relationship between a protocol and a capsule class. The port name is specified using the association end name. The cardinality can also be specified.

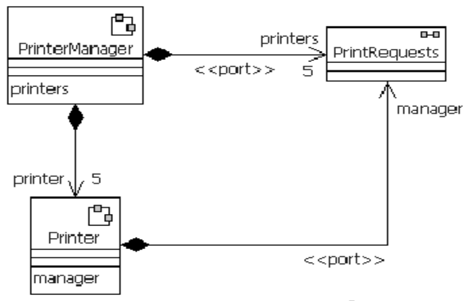
Class diagram shows a different perspective

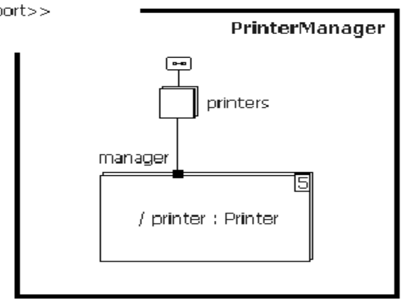
The decomposition of a capsule can also be shown in a capsule collaboration. In addition to the information shown in the class diagram the capsule collaboration diagram specifies the precise interconnection topology between capsule roles, indicated by connectors.

Example

The following diagrams show the same model, but it is shown from both the class diagram and then the capsule collaboration diagram perspectives.

Figure 15 Class diagram





Realization

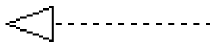
A realization relationship defines a contract between classifiers where the contract is set of behaviors. There are two elements in the UML that can be realized: interfaces and use cases. Simply put, interfaces and use cases specify behavior without detailing the implementation. The classifier who will realize the interface or use case is responsible for providing the implementation.

Realizations are a good way of separating the specification from the implementation.

Realization is a form of generalization, in which only behavior is inherited.

Graphical notation

Realizations are represented as a cross between a dependency and generalization as a hashed line with a large open arrowhead.



Realization of Use Cases

In an executing system, an instance of a use case does not correspond to any particular object in the implementation model. Instead it corresponds to a specific flow of events that is executed as sequence of events between implementation objects.

By creating a society of classes and other elements that work together to implement the behavior of the use case we realize a use case.

Note: *The relationship between the use case and its realization may not be visualized explicitly, although the tools that are used to manage your models maintain this relationship.*

Using interaction diagrams

Use cases are realized by describing its flow of events in interaction diagrams. You should describe each flow variant in a separate diagram. Start by describing the basic flow, then describe variants such as exceptional flows, error handling, and time-out handling.

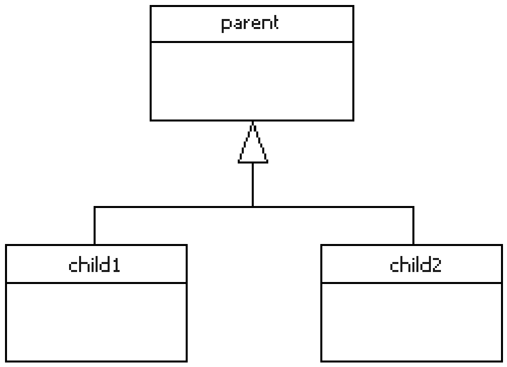
Note: *The focus of a system's architecture is to find the minimal set of well-structured interactions for all use cases in a system.*

Generalization

A generalization relationship between classes shows a relationship between a general element, called the superclass or parent, and a more specific element, called the subclass or child. With a generalization relationship, the child will inherit all the structure and behavior defined in the parent. The child may also add new structure or behavior.

Graphical notation

A generalize relationship is a solid line with an arrowhead pointing to the superclass.

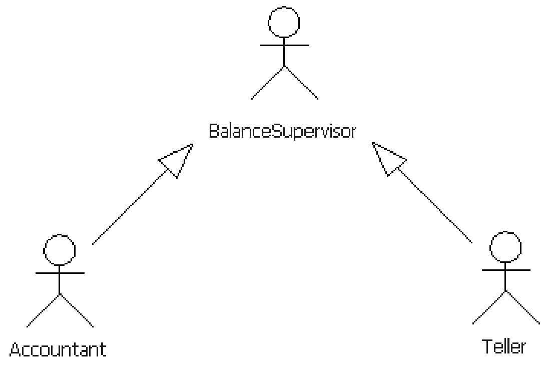


Details

Generalization is a static relationship, meaning that it can only be visualized in a class diagram or a use case diagram. In addition, generalization can only link same types of elements. A capsule cannot be a superclass of a class.

Actor Generalization

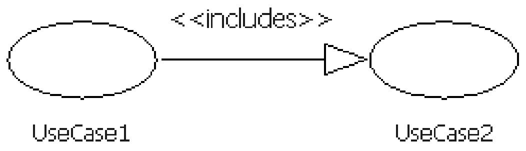
Several actors can play the same role in a particular use case. To make the model clearer, you can represent the different kinds of users user by subclassing. Each inherited actor represents one of the user's roles relative to the system.



Include Relationship

The include-relationship connects a base use case to an inclusion use case. The inclusion use case is always abstract. It describes a behavior segment that is inserted into a use-case instance that is executing the base use case. The base use case has control of the relationship to the inclusion and can depend on the result of performing the inclusion, but neither the base nor the inclusion may access each other's attributes. The inclusion is in this sense encapsulated, and represents behavior that can be reused in different base use cases.

Figure 16   Example includes relationship

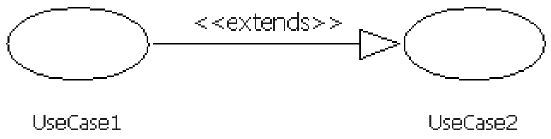


You can use the include-relationship to:

1. Factor out behavior from the base use case that is not necessary for the understanding of the primary purpose of the use case, only the result of it is important.
2. Factor out behavior that is in common for two or more use cases.

Extend Relationship

The extend-relationship connects an extension use case to a base use case. It is used to model part of a use case that a user may see as optional.



You can use the extensions for several purposes:

1. To show that a part of a use case is optional, or potentially optional, system behavior. In this way, you separate optional behavior from mandatory behavior in your model.
2. To show that a subflow is executed only under certain (sometimes exceptional) conditions, such as triggering an alarm.
3. To show that there may be a set of behavior segments of which one or several may be inserted at an extension point in a base use case. It will depend on the interaction with the actors during the execution of the base use case which of the behavior segments are inserted and in what order.

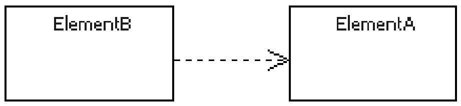
The specialization is conditional, which means its execution is dependent on what has happened while executing the base use case. The base use case does not control the conditions for the execution of the specialization, those conditions are described within the extend-relationship. The specialization use case may access and modify attributes of the base use case. The base use case, however, cannot see the specializations and may not access their attributes. The base use case is implicitly modified by the specializations. You can also say that the base use case defines a modular framework into which specializations can be added, but the base does not have any visibility of the specific specializations. The base use case should be complete in and of itself, meaning that it should be understandable and meaningful without any references to the specializations. However, the base use case is not independent of the specializations, since it cannot be executed without the possibility of following the specializations.

Dependency

A dependency relationship is used to specify that a change in the specification of one element may affect another element that uses it, but not necessarily the reverse. Dependency relationships are used to model dependencies that have not been implicitly captured by the other types of relationships in your model.

Graphical notation

A dependency relationship is a dotted line with an arrowhead at one end. The arrowhead points to the supplier class. In this example, Element B is dependent on class A.



Applications

Dependency relationships can have different shades of meaning depending on which elements are part of the relationship. The different meanings can be shown in your diagram by applying stereotypes to the relationship.

Example uses:

1. A client class accesses a value, constant or variable, defined in a supplier class/interface (class diagram).
2. An operations of a client class invoke operations of a supplier class/interface (class diagram).
3. Operations of a client class have signatures whose return class or arguments are instances of a supplier class/interface (class diagram).

4. A component requires a compilation dependency on another component (component diagram).
5. A use case includes or extends another use case (use case diagram).
6. To show the layering of a system, you can add dependencies between packages.

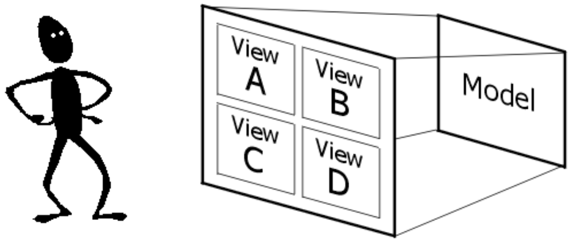
Component-Dependency Relationship

An important use of a dependency relationship is to represent compilation dependencies between components. A compilation dependency exists from one component to the components that are needed to compile the component. In C++, for example, the compilation dependencies are indicated with #include statements. In Ada, compilation dependencies are indicated by the with clause. In Java the compilation dependency is indicated by the import statement. In general there should be no cyclical compilation dependencies.

Chapter 4

Diagrams

Diagrams allow you to assemble related collections of elements together into a graphical depiction of all or part of a model. Each diagram provides a view into the elements that make up your model. In this way the user of the model can decide to see only the views of the underlying model that are of interest.



Important visual relationships

Although each type of diagram shows different views of the model, they all show common relationships between the elements. The most important of these relationships are:

- Connections - provide some clue as to the relationships between elements.

- Containments - are shown as symbols with a boundary. For example, a capsule’s collaboration is shown as a box with a heavy border to represent the boundary of the capsule. Capsule roles within the boundary are contained by the capsule. And elements placed directly on the boundary are interface elements, visible from outside the element.
- Visual attachment - symbols being near or far from one another, for example, represent layering in a system.

Structure

- “Class Diagram” on page 88 (static structure) - is a high level generalization of a system that shows a set of elements and their general relationships.
- “Collaboration Diagram” on page 91 (dynamic structure) - captures a desired pattern of interactions between a set of objects, emphasizing the structural organization of the objects.
- “Component Diagram” on page 97 - captures the static implementation view of a system.
- “Deployment Diagram” on page 99 - captures the configuration of run-time processing nodes and the components that run on them.

Behavior

- “Sequence Diagram” on page 95 - captures interactions between a set of objects, emphasizing the logical ordering of messages.
- “State Diagram” on page 91 - captures the dynamic aspects of an event-driven system, and is best used for modeling the behavior of event-driven classes.
- “Use Case Diagram” on page 87 - captures the context and intended behavior of the system, a subsystem, or a class.

Real-time specialization

In addition to the base UML diagrams, the “Capsule Structure Diagram” on page 93 is a specialized form of the collaboration diagram with formal semantics that enable complete code generation:

- “Capsule Structure Diagram” on page 93 - captures structural patterns that specify the communication relationships between a capsule's objects. The communication takes place in order to accomplish a task, or the behavior of the capsule. The diagram also shows the interface elements of a capsule.

Use Case Diagram

A use case diagram shows actors and use cases together with their relationships. The individual use cases represent functionality, or requirements of functionality of a system, a class, or a capsule.

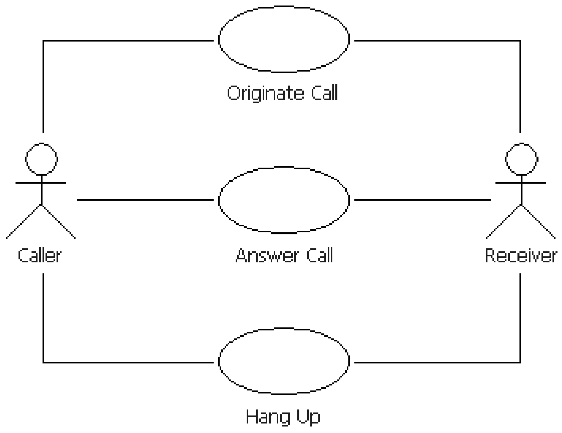
Use case diagrams can be organized into (and owned by) use case packages, showing only what is relevant within a particular package.

It is recommended that you include each actor, use case, and relationship in at least one of the diagrams. If it makes the use case model clearer. They can be part of several diagrams and you can show them several times in the same diagram.

Graphical notation

A use case diagram is a graph of actors, use cases, use case packages, and the relationships between these elements.

Example



Class Diagram

Class diagrams show the static structure of the model. Although it is called a class diagram, it may also contain other elements besides classes that exist in a model, such as capsules, protocols, packages, their internal structure, and their relationships to other elements. Class diagrams do not show temporal information.

Class diagrams may be organized into (and owned by) packages, but the individual class diagrams are not meant to represent the actual divisions in the underlying model. A package may then be represented by more then one class diagram.

A model element can appear in more than one class diagram.

Graphical notation

The basic notation for elements in a class diagram is using a solid-outline rectangle with three compartments separated by a horizontal line. The top compartment is used to display the name of the element, and other optional properties such as stereotypes and icons. The bottom compartments, or list compartments, are used to show string representations of an elements features. For example operations and attributes are commonly represented. However, other optional list compartments can show other features. For example, a capsule has a list compartment for ports and capsule roles.

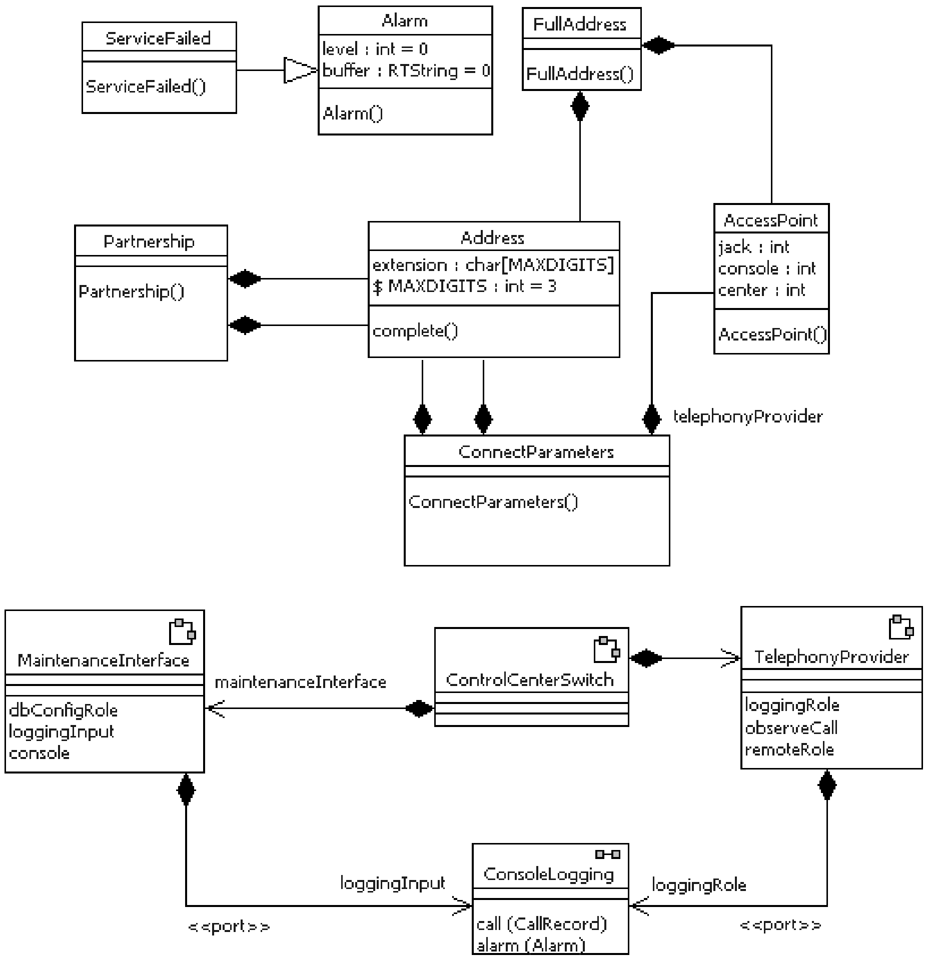
Relationships are shown as lines connecting two element symbols in the diagram. The lines may have a number of graphical representations to show their properties.

Example

The following class structures are suitable for illustration in class diagrams, but you will not use all of them in all situations. Each class structure should have its own class diagram.

1. The most important classes and their relationships. Diagrams of this type can function as an object model summary and are of great help in reviewing the model. These diagrams are likely to be included in the logical view of the architecture.
2. Functionally related or coherent classes.
3. Classes that belong to the same package.
4. Important aggregation hierarchies.
5. Important structures of entity objects, including class structures with association, aggregation and generalization relationships. If possible you should create a class diagram that contains all the classes of the long-lived objects and their relationships. This kind of diagram is especially useful in reviewing what is stored in the system, and the storage structures.
6. Packages and their dependencies, possibly illustrating their layering.
7. Classes that participate in a specific use-case realization.
8. A single class, its attributes, operations, and relationships with other classes.

Figure 17 A class diagram showing Aggregation hierarchies



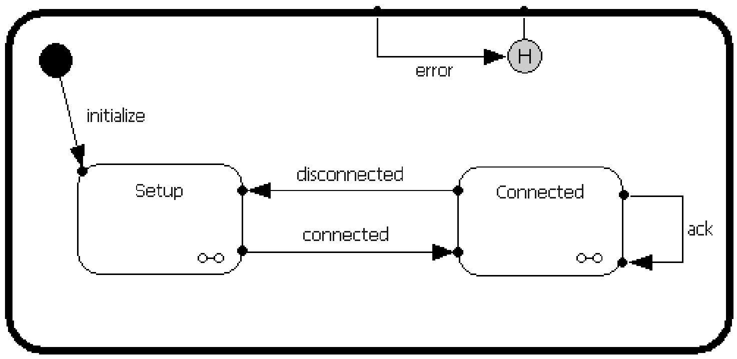
State Diagram

A state diagram shows the sequence of states that an object or an interaction goes through during its life in response to received messages, together with its responses and actions. A state machine is a graph of states and transitions that describes the response of an object of a given class to the receipt of outside stimuli. State diagrams show a state machine and are especially useful in modeling event-driven systems.

Graphical notation

A statechart diagram represents a state machine. The states are represented by state symbols and the transitions are represented by arrows connecting the state symbols. States may also contain subdiagrams, or other state machines that represent different hierarchical state levels.

Example



Collaboration Diagram

Collaboration diagrams show the communication patterns among a set of objects or roles to accomplish a specific purpose. The diagram can be shown in two different forms: either a specification level (showing classifier roles, association roles, and messages) or at the instance level (showing objects or instances, links, and stimuli).

Collaborations are the constraining element to a set of sequences. The sequences show all the different communication scenarios that can occur between the instances or roles in the collaboration, while the collaboration shows the connection topology between the elements.

To model the explicit time related sequence of interactions between objects, use a sequence diagram.

It is important to understand that a collaboration defines a set of interactions that are meaningful for a given purpose, for example, to accomplish a certain task. However, a collaboration does not identify a global relationships between model elements.

Roles and objects

The participants in a collaboration define the roles that objects play in an interaction. The role describes the type of object that can play the role, such as an object with the required interface.

Graphical notation

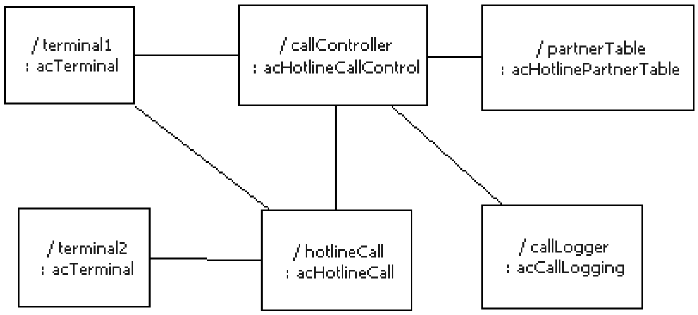
A collaboration is shown as a graph of classifier roles together with connected lines called association roles. Normally, only the name of the compartment is shown. The name compartment contains the string:

```
role name : classifier name
```

A communication relationship can be shown between roles in a collaboration by adding an association role, a solid line connecting two role boxes.

Example

A collaboration may be attached to a class or a use case to describe the context in which their behavior occurs. For example, by showing the roles objects play to perform the behavior of a use case or operation.



A collaboration can also be used to formally specify the composite structure of a capsule. This specialized collaboration, called a capsule structure, shows roles or capsule roles, and their connectors. In addition, the collaboration visually shows the capsules interfaces by placing them on the boundary of the collaboration.

Capsule Structure Diagram

A capsule structure diagram is a specialized collaboration diagram. This diagram is used for the same purpose as the general collaboration, that is to specify a pattern of communication between objects. However in a capsule structure the communication pattern is owned by a particular capsule and represents the composite structure of its capsule roles, ports, and connectors.

It is important to understand that a capsule structure defines a set of interactions that are meaningful for a given purpose, that is for the implementation of it's behavior (e.g. a capsules behavior is actually the composite behavior of all its components). However the collaboration does not identify global relationships between its capsule role.

Differences between a general collaboration and a capsule structure

- capsule roles - The roles in a capsule structure are restricted to capsule roles; association roles are not allowed. When a capsule role is shown in a capsule structure diagram, its public ports are shown. This allows connectors to be created between capsule roles.

  Note: *You can model the collaboration of a capsule's attributes that are not capsule classes by using a normal collaboration diagram.*

- ports - Since capsules communicate with each other via ports (and not operation invocation), a capsule structure shows a capsule's ports. Ports can be placed on the boundary of the collaboration to show that they are externally visible (public interfaces), or contained within the boundary to show that they are protected (not accessible from outside the capsule).
- capsule boundary - A capsule structure diagram shows a visual representation of the capsule's encapsulation shell. This shell shows both the implicit containment relationship between capsule roles and a capsule, and visually identifies the ports which are interfaces.
- connectors - In a general collaboration communication between objects is modeled using an association role between two classifier roles. In a capsule structure, communication relationships are explicitly shown between capsule ports.
- code generation - A capsule's collaboration is a formal specification which allows for the source code implementation to be automatically generated. The semantics of a general collaboration are not formal enough to result in automatic code generation.

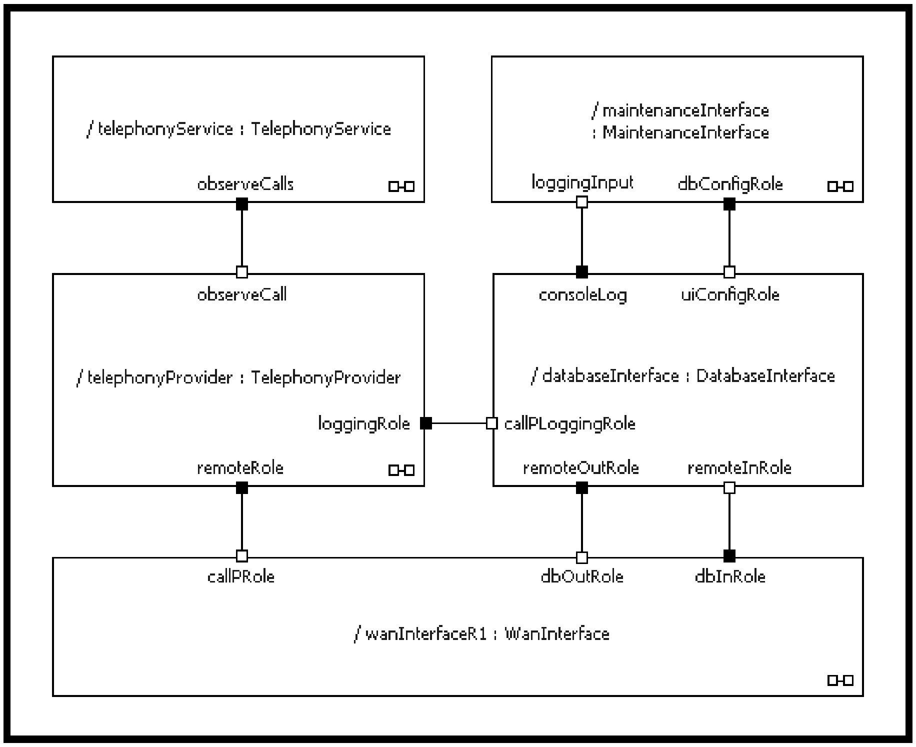
Graphical notation

A capsule structure is shown as a box with a heavy border, which represents the capsule's boundary. Capsule roles are shown inside the boundary as composite parts. Ports are shown as rectangles and connectors as solid lines connecting ports.

Example

The following capsule structure shows the capsule roles which make up a control center switching software.

Figure 18 Capsule structure diagram example


/ telephonyService : TelephonyService
observeCalls
/ maintenanceInterface : MaintenanceInterface
loggingInput
dbConfigRole
observeCall
consoleLog
uiConfigRole
/ telephonyProvider : TelephonyProvider
loggingRole
/ databaseInterface : DatabaseInterface
callPLoggingRole
remoteRole
remoteOutRole
remoteInRole
callPRole
dbOutRole
dbInRole
/ wanInterfaceR1 : WanInterface

Sequence Diagram

An interaction is a pattern of communication among objects at run-time. A sequence diagram is used to show this interaction from the perspective of showing the explicit ordering messages. Sequence diagrams are often used to show specific communication scenarios of a collaboration.

Sequence diagrams are particularly important to designers because they clarify the roles of objects in a flow and thus provide basic input for determining class responsibilities and interfaces.

Graphical notation

A sequence diagram has two dimensions, the vertical dimension represents time, and the horizontal dimension represents the different objects in the interaction.

Object box

In a sequence diagram each object that participates in the interaction is represented by a rectangular box at the top of the diagram. The name field maps to the name of an object which conforms to a role in a collaboration.

Lifelines

These are the dashed vertical lines that descend from the object box. They represent the existence of the object at a particular time. When an object is created or destroyed, then its lifeline start or stops at the appropriate point. The object symbol is drawn at the top of the lifeline. If the object is destroyed, then its destruction is marked on the lifeline by a large 'X'.

Focus of control

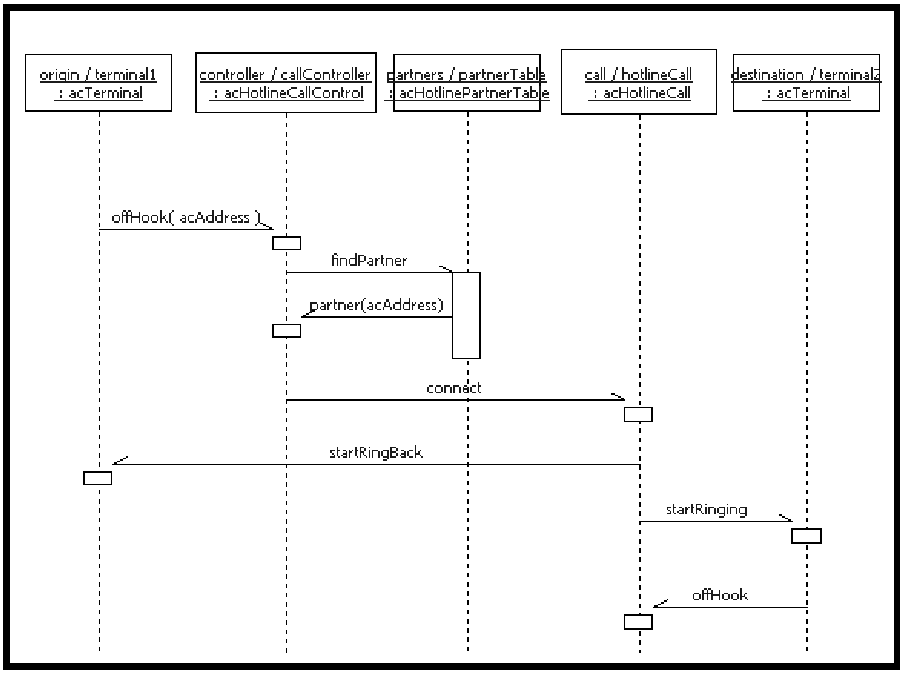
An activation, or focus of control, shows the period during which an object is performing an action. It represents both the duration of the action and the control relationship between the activation and its callers.

Messages

A message is the specification of a communication between objects that convey information with the expectation that activity will occur upon receipt. A message instance is shown as a line from the lifeline of one object to the lifeline of another. In the case of a message sent by an object to itself, the arrow may start and finish on the same lifeline. The arrow is named with the name of the message. The arrow head of the message can be shown in different ways to convey the different types of message communication.

Example

Figure 19 An abbreviated call setup scenario


origin / terminal1 : acTerminal
controller / callController : acHotlineCallControl
partners / partnerTable : acHotlinePartnerTable
call / hotlineCall : acHotlineCall
destination / terminal2 : acTerminal
offHook(acAddress)
findPartner
partner(acAddress)
connect
startRingBack
startRinging
offHook

Component Diagram

A component diagram shows the dependencies among software components. A software module may be represented as a component. Some components exist at compile time, some exist at link time, some exist at run time, and some exist at more than one time. A compile-only component is one that is only meaningful at compile time. The run-time component in this case would be an executable program. A component diagram has only a type form, not an instance form. To show component instances, use the deployment diagram.

Graphical notation

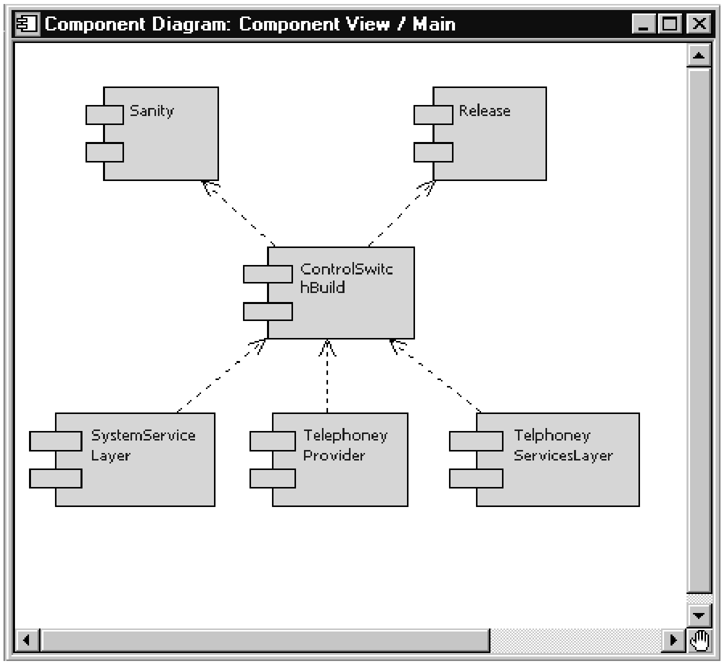
A component diagram is a graph of components connected by dependency relationships. Components can be connected to components by physical containment representing composition relationships. Components can also be organized in component packages. Component diagrams contain:

- component packages
- components
- dependency relationships

You can create one or more component diagrams to depict the component packages and components at the top level of the component view, or to depict the contents of each component package. Such component diagrams belong to the component package that they depict.

Example

Figure 20 Example Component diagram



Deployment Diagram

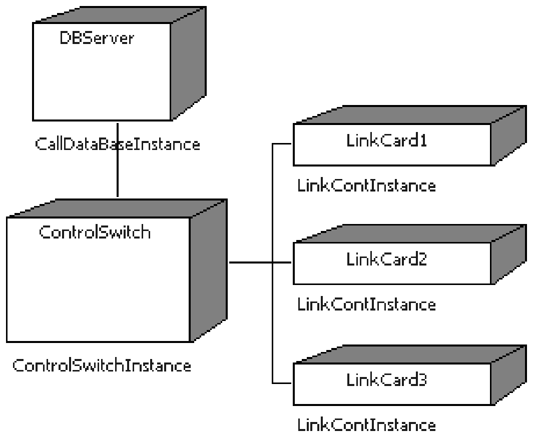
The deployment diagram provides a basis for understanding the physical distribution of the run-time processes across a set of processing nodes. There is only one deployment view of the system. Nodes may contain component instances, which indicates that the component runs on the node.

Graphical notation

A deployment diagram is a graph of nodes connected by a communication association called a connection. The deployment diagram is used to show which components will run on which nodes.

Example

Figure 21 Example Deployment diagram



Index

A

Actions 45
actions 45
    blocking 46
    Notation 45
Actions, messages, and events 45
actor
    generalization 81
Actor generalization 81
Actors 19
    graphical notation 20
    relationships 20
Additional properties 70
Aggregate or association? 73
Aggregation 72
aggregation 72
    Graphical notation 73
Annotational 18
Association 70
    class 27
    roles 71
association 70
    Additional properties 71
    aggregation 72
    composition 73
    ends 71
    Graphical notation 70
    multiplicity 71
    name 71
    navigability 72
    roles 70
Association class 27
association class 27, 75
    Graphical notation 27
Association ends 71
Attributes 25
attributes 25
    Changeability properties 25
    private (capsules) 31

B

Behavioral 18
Binary protocols 39
Blocking actions 46
Building blocks 10

C

Call event 46
call event 46
    flow of control 46

Capsule aggregation and composition relationships 77
Capsule class aggregation and composition relationships 77
capsule collaboration and general collaboration, differences between 94
Capsule collaboration diagram 93
Capsule roles 33
capsule roles 33
  Cardinality 34
  Substitutability 34
capsule structure diagram 93
  capsule boundary 94
  capsule roles 94
  code generation 94
  connectors 94
  Differences between a general collaboration and a capsule collaboration 94
  Example 95
  Graphical Notation 94
  ports 94
Capsules 30
capsules 30
  aggregation & composition-relationship 33
  association 33
  Behavior 32
  connector 33
  dependency-relationship 33
  generalization-relationship 33
  Graphical notation 32
  Logical threads of control 32
  message passing 31
  private attributes 31
  public ports 31
  Relationships 33
  roles 33
  Structure 31
Capsules and ports 11
cardinality 40
  capsule roles 34
  Common structural patterns 41
  rules for capsule roles 40
  unspecified 40
Cardinality and capsule structure 40
cardinality and ports 40
Cardinality rules for capsule roles 40
choice point 61
  example 62
  Graphical notation 62
Choice points 61
class 23
  Behavior 23
  Graphical notation 24
  instantiated 29
  parameterized 28
  parameterized utility 30
  Persistence 23
  Standard stereotypes of classes 23
  Structure 23
  utility 27
Class diagram 78
Class diagram shows a different perspective 78
class utility
  instantiated 28
  parameterized 30
Classes 23
Classification of capsule roles 33
Collaboration diagram 91
  Example 93

Graphical Notation 92
Roles and objects 92
Common structural patterns 41
communication rules for ports 35
Component diagram 97
component diagram 97
Example 98
graphical notation 98
Component instances 66
Component-dependency relationship 84
component-dependency Relationship 84
Components 65
components 65
Mapping from logical to physical 66
Organization 66
Relationships 66
Composition 73
composition 73
Graphical notation 74
Concrete and abstract use cases 21
Concurrency 11
concurrency
simplifying 53
use case 22
Connections 85
Connectors 76
connectors 76
Graphical notation 77
Containments 86
Continuation kinds - shallow history, deep history and default 56

D

Dependency 83
dependency 83
Applications 83
example uses 83
Graphical notation 83
Dependency relationship 83
Deployment diagram 99
deployment diagram 99
Example 99
Graphical Notation 99
deployment diagram, a simple 67
Diagrams
Capsule collaboration 93
Collaboration 91
Component 97
Deployment 99
diagrams
Behavior 86
capsule structure 93
component 97
deployment 99
important visual relationships 85
real-time specialization 87
sequence 95
state 91
Structure 86
Differences between a general collaboration and a capsule collaboration 94

E

End ports 36
event
call 46
signal 46
events 45
Example 56, 89

Example includes relationship 81
Example state diagram 48
Example uses
  83
Executable Models 13
Extend Relationship 82
external incoming junction 60
external junction 60
external outgoing junction 60

F

final state 55
flow of control
  call event 46
  signal event 47
flow of events 20
  Documenting 21
  Structure 21

G

general collaborations and capsule collaboration, differences between 94
Generalization 80
generalization
  actor 81
  Details 80
  Graphical notation 80
  relationships 80
Graphical notation 19, 20
Group transitions 57
group transitions 57
Grouping 18
guarded condition (trigger) 51

H

Hierarchical State Machines 55
Hierarchical states 49
History - hierarchical state machines 55
history, an example use of 56

I

Include relationship 81
include relationship 81
initial
  point 54
  transition 54
Initial point and initial transition 54
initial state
  transition from 54
Initial state and initial transition 54
initial transition
  Graphical notation 55
  none 54
instance
  use case 22
Instantiated class 29
instantiated class 29
  Graphical notation 29
Instantiated class utility 28
instantiated class utility 28
  Graphical notation 28
Interactions 64
interactions 64
Interfaces 24
interfaces 24
  Graphical notation 24
internal junction 60

J

Joining transition segments 58
junction point
  continuation 58
  Continuing junctions 60
  Graphical notation 59
  joining transition segments 58
  terminating junction types 61
  transition segments 58
Junction points 52, 57
junction points 57

L

logical threads 32

M

message passing
  capsules 31
messages 45
  sequence diagram 96
Modeling Language Reference 9
Modeling language Reference 9
Multiple containment 42
  example 44
Multiple containment example 44
multiplicity
  association 71

N

Nodes 66
nodes 66
  Connections 67
Notation 88
Notes 68
notes 68
  Graphical notation 68
  Relationships 68

O

Operations 25
operations 25
  Class or instance 26
  Have Parameters 26
  Properties 26
Other features 19

P

package 67
  Content Visibility 67
  overview 67
Packages 67
packages
  use case 22
Parameterized class 28
parameterized class 28
  Graphical notation 29
Parameterized utility class 30
parameterized utility class 30
  Graphical notation 30
Persistence 23
point
  initial 54
Ports 34
ports 34
  classification of 35
  Communication rules 35
  Connector type 36
  End 36

Graphical notation 36
Non-wired 36
public 31
Relay 36
Termination 36
Visibility 35
Wired 36
Ports and protocols 35, 39
Protocol participants (protocol roles) 39
protocols 38
binary protocols 39
Graphical notation 40
participants (protocol roles) 39
public ports 31
Purpose 10

Q
qualifiers 74

R
Realization 79
realization 79
Graphical notation 79
of use cases 79
Using interaction diagrams 79
Realization of use cases 79
Real-time extension 69
Real-time extension to UML 10
Real-time notations 69
Real-time notations to UML 10
Real-time specialization 87
Real-Time Specializations Overview 12
Relationships 19, 20, 66, 69
relationships 69
actor communicates-Association 76
aggregation 72
association 70
association classes 75
capsule class aggregation 77
component-Dependency 84
composition 73, 77
connectors 76
constraints 75
dependency 83
Example 78
extend 82
generalization 80
include 81
qualifiers 74
realization 79
visibility 74
Relationships between capsule classes 77
Relationships between capsule classes and protocol classes 78
relay ports 36
Run-to-completion 53
run-to-completion 53
Simplifying concurrency 53

S
Sequence diagram 95
sequence diagram
Example 97
Focus of control 96
Graphical Notation 96
Lifelines 96
Messages 96
Object box 96

Signal event 46
signal event 46
  flow of control 47
State diagram 91
state diagram 91
  Example 91
  graphical notation 91
State machine 47
state machine 47
  Events and signals 47
  Overview 48
  variations 47
State Machines
  Hierarchical 55
States 49
states 49
  Entry and Exit actions 49
  External view 50
  final 55
  Graphical notation 49
  hierarchical 49
  Internal view 50
  top 55
Structural 18
Structural patterns with cardinality 41
structural patterns, common 41
Substitutability 41
substitutability 41
  rules 42
Support for real-time systems 11

T

Terminating junctions (to history) 61
threads
  control (capsules) 32
Top state 55
top state 55
transition 51
  Actions 51
  Example 64
  Graphical notation 53
  group 57
  Guard Condition 51
  initial 54
  junction points 52
  selection rules 63
  to and from the initial state 54
  Trigger 51
  types 52
Transition segments 58
Transition selection rules 63
Transitions 51
trigger 51

U

Unspecified cardinality 40
Use Case
  concurrency 22
  Graphical Notation 87
  instance 22
  packages 22
Use case concurrency 22
Use Case diagram
  diagrams
    Use Case 87
  Example 88
Use case diagram 87
Use case instance 22
Use case packages 22

Use Cases
    abstract 21
    concrete 21
    graphical notation 19
    other features 19
    relationships 19
Use cases 18
Utility class 27
utility class 27
    Graphical notation 27

V

visibility 74
Visual attachment 86

W

When to use multiple containment? 43
wired ports 36