UNIT-II

I. BASIC STRUCTURAL MODELING

Contents:

1. **Classes**
2. **Relationships**
3. **Common Mechanisms**
4. **Diagrams**

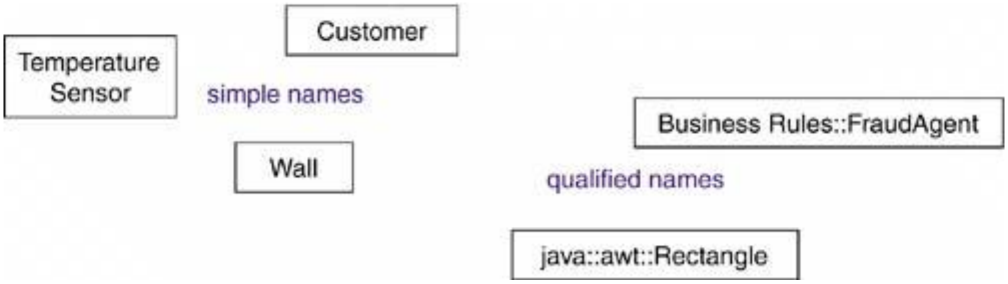
1. Classes:

Terms and Concepts:

A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Graphically, a class is rendered as a rectangle.

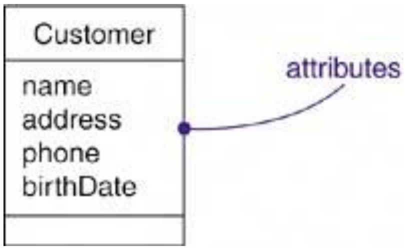
Names

Every class must have a name that distinguishes it from other classes. A *name* is a textual string. That name alone is known as a simple name; a qualified name is the class name prefixed by the name of the package in which that class lives. A class may be drawn showing only its name



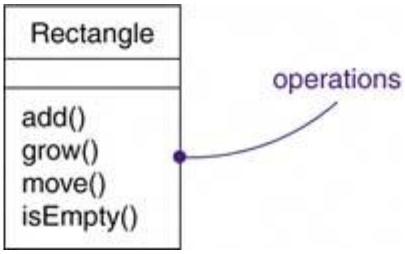
Attributes

An *attribute* is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all. An attribute represents some property of the thing you are modeling that is shared by all objects of that class. For example, every wall has a height, width, and thickness; you might model your customers in such a way that each has a name, address, phone number, and date of birth

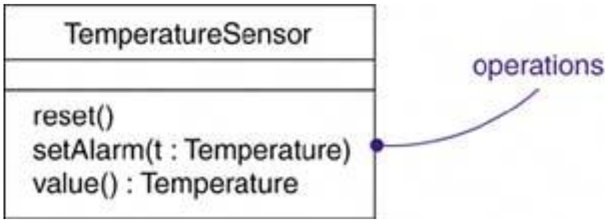


Operations

An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior. In other words, an operation is an abstraction of something you can do to an object that is shared by all objects of that class. A class may have any number of operations or no operations at all. For example, in a windowing library such as the one found in Java's awt package, all objects of the class Rectangle can be moved, resized, or queried for their properties. Often (but not always), invoking an operation on an object changes the object's data or state. Graphically, operations are listed in a compartment just below the class attributes. Operations may be drawn showing only their names

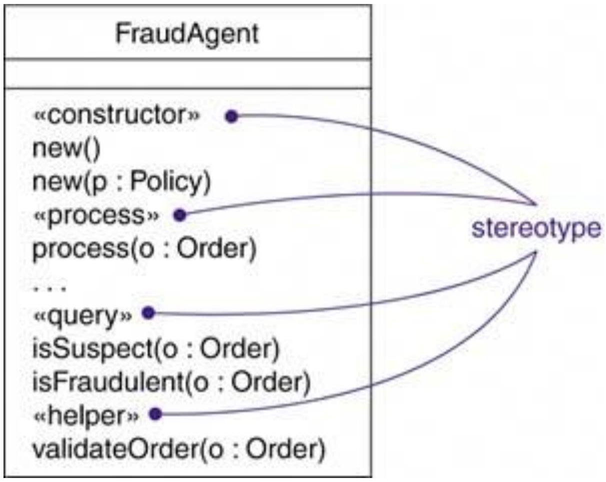


You can specify an operation by stating its signature, which includes the name, type, and default value of all parameters and (in the case of functions) a return type



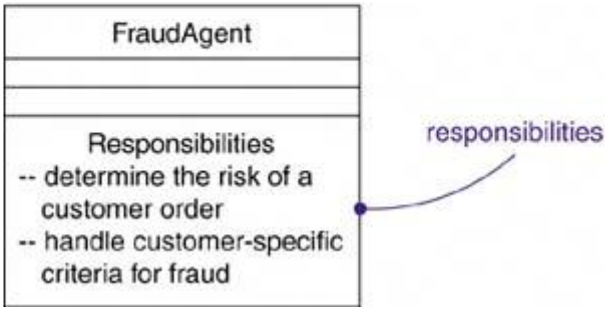
Organizing Attributes and Operations

When drawing a class, you don't have to show every attribute and every operation at once. In fact, in most cases, you can't (there are too many of them to put in one figure) and you probably shouldn't (only a subset of these attributes and operations are likely to be relevant to a specific view). For these reasons, you can elide a class, meaning that you can choose to show only some or none of a class's attributes and operations. You can indicate that there are more attributes or properties than shown by ending each list with an ellipsis ("...").



Responsibilities

A *responsibility* is a contract or an obligation of a class. When you create a class, you are making a statement that all objects of that class have the same kind of state and the same kind of behavior. At a more abstract level, these corresponding attributes and operations are just the features by which the class's responsibilities are carried out. A Wall class is responsible for knowing about height, width, and thickness; a FraudAgent class, as you might find in a credit card application, is responsible for processing orders and determining if they are legitimate, suspect, or fraudulent; a TemperatureSensor class is responsible for measuring temperature and raising an alarm if the temperature reaches a certain point.


FraudAgent
Responsibilities
-- determine the risk of a customer order
-- handle customer-specific criteria for fraud
responsibilities

Common Modeling Techniques

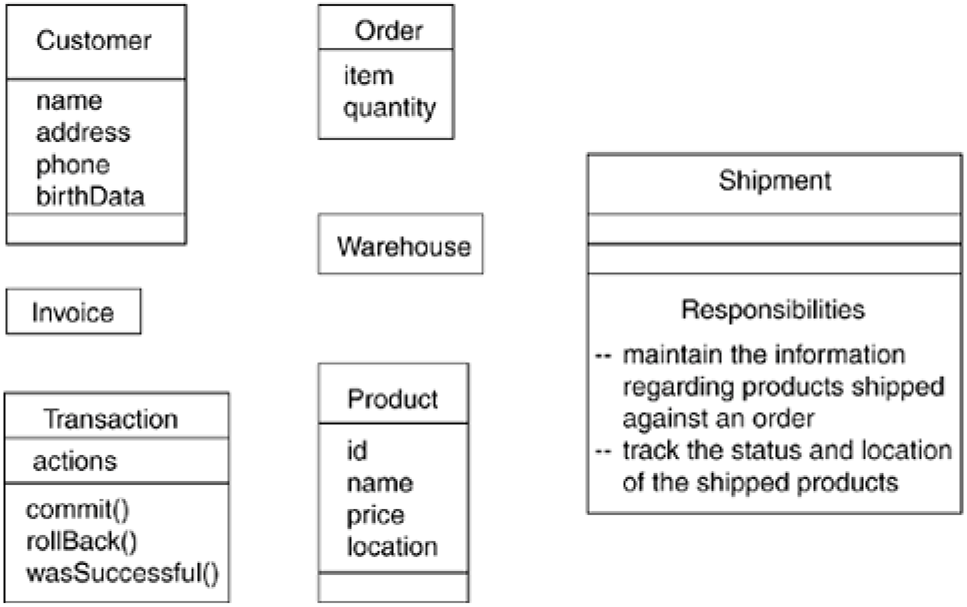
Modeling the Vocabulary of a System

You'll use classes most commonly to model abstractions that are drawn from the problem you are trying to solve or from the technology you are using to implement a solution to that problem. Each of these abstractions is a part of the vocabulary of your system, meaning that, together, they represent the things that are important to users and to implementers.

To model the vocabulary of a system,

- Identify those things that users or implementers use to describe the problem or solution. Use CRC cards and use case-based analysis to help find these abstractions.
- For each abstraction, identify a set of responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all your classes.
- Provide the attributes and operations that are needed to carry out these responsibilities for each class.

A set of classes drawn from a retail system, including Customer, Order, and Product. This figure includes a few other related abstractions drawn from the vocabulary of the problem, such as Shipment (used to track orders), Invoice (used to bill orders), and Warehouse (where products are located prior to shipment). There is also one solution-related abstraction, TRansaction, which applies to orders and shipments.



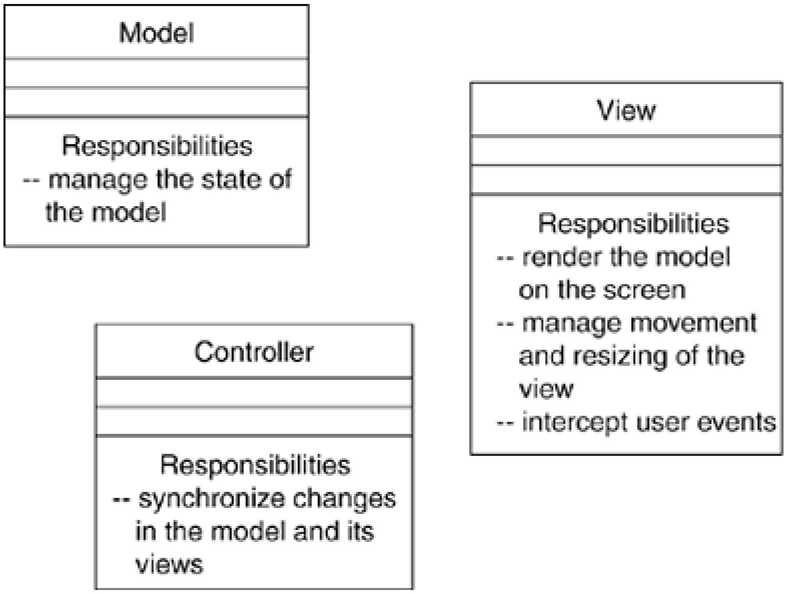
Modeling the Distribution of Responsibilities in a System

Once you start modeling more than just a handful of classes, you will want to be sure that your abstractions provide a balanced set of responsibilities.

To model the distribution of responsibilities in a system,

- Identify a set of classes that work together closely to carry out some behavior.
- Identify a set of responsibilities for each of these classes.
- Look at this set of classes as a whole, split classes that have too many responsibilities into smaller abstractions, collapse tiny classes that have trivial responsibilities into larger ones, and reallocate responsibilities so that each abstraction reasonably stands on its own.

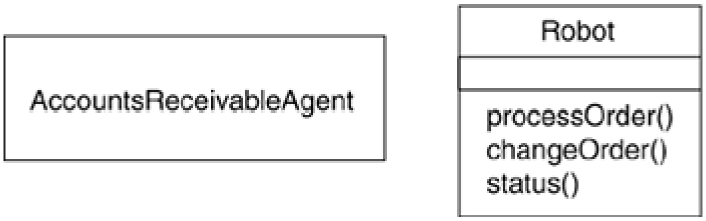
- Consider the ways in which those classes collaborate with one another, and redistribute their responsibilities accordingly so that no class within a collaboration does too much or too little.


Model
Responsibilities
-- manage the state of the model
View
Responsibilities
-- render the model on the screen
-- manage movement and resizing of the view
-- intercept user events
Controller
Responsibilities
-- synchronize changes in the model and its views

Modeling Nonsoftware Things

To model nonsoftware things,

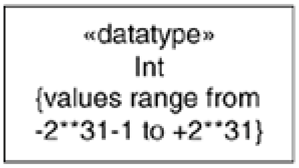
- Model the thing you are abstracting as a class.
- If you want to distinguish these things from the UML's defined building blocks, create a new building block by using stereotypes to specify these new semantics and to give a distinctive visual cue.
- If the thing you are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node as well, so that you can further expand on its structure.

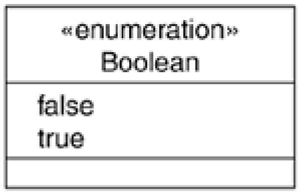


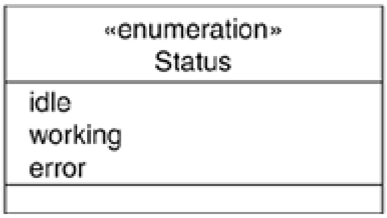
Modeling Primitive Types

To model primitive types,

- Model the thing you are abstracting as a class or an enumeration, which is rendered using class notation with the appropriate stereotype.
- If you need to specify the range of values associated with this type, use constraints.







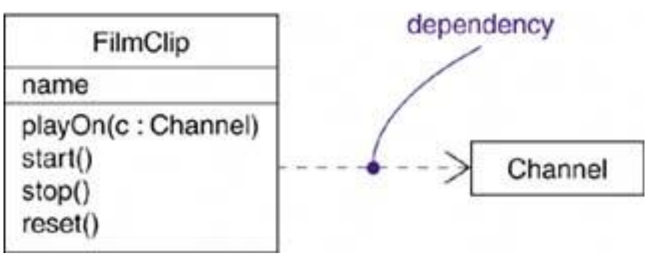
2.Relationships:

Terms and Concepts

A *relationship* is a connection among things. In object-oriented modeling, the three most important relationships are dependencies, generalizations, and associations. Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships.

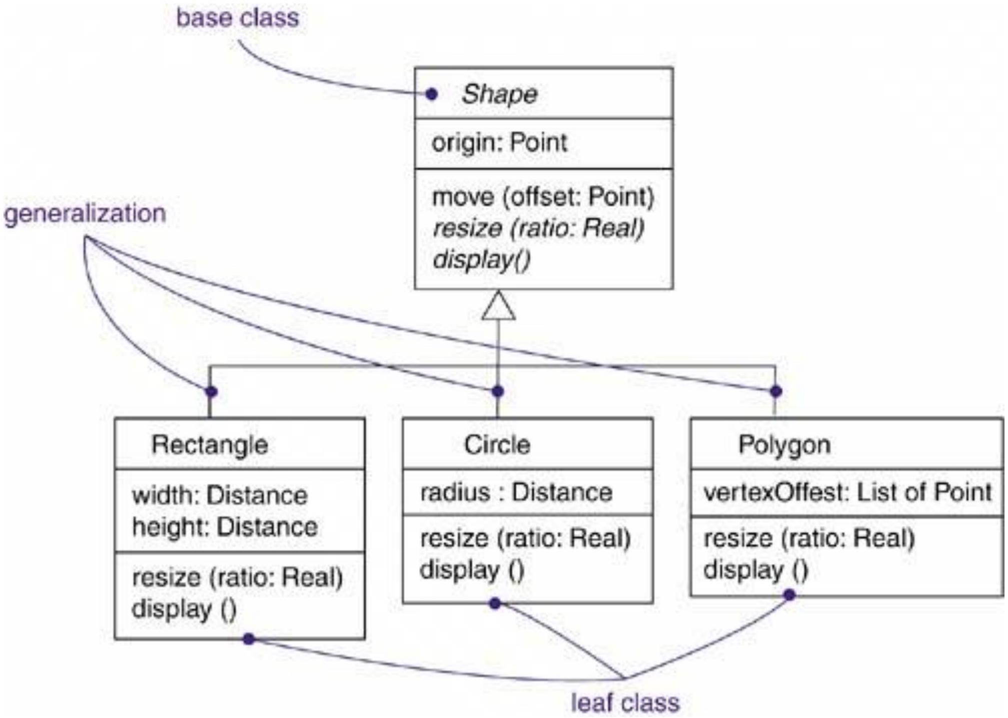
Dependencies

A *dependency* is a relationship that states that one thing (for example, class Window) uses the information and services of another thing (for example, class Event), but not necessarily the reverse. Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on. Choose dependencies when you want to show one thing using another.



Generalizations

A *generalization* is a relationship between a general kind of thing (called the superclass or parent) and a more specific kind of thing (called the subclass or child). Generalization is sometimes called an "is-a-kind-of" relationship: one thing (like the class BayWindow) is-a-kind-of a more general thing (for example, the class Window). An objects of the child class may be used for a variable or parameter typed by the parent, but not the reverse



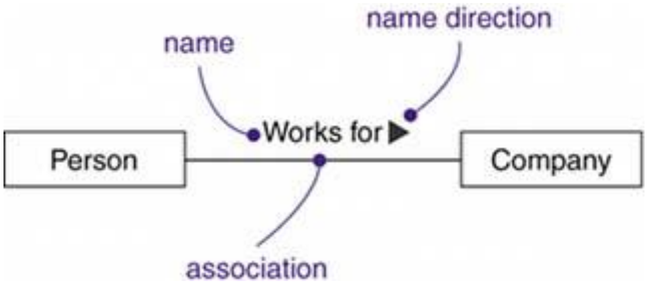
Associations

An *association* is a structural relationship that specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can relate objects of one class to objects of the other class. It's quite legal to have both ends of an association circle back to the same class. This means that, given an object of the class, you can link to other objects of the same class. An association that connects exactly two classes is called a binary association. Although it's not as common, you can have associations that connect more than two classes; these are called *n-ary associations*.

Beyond this basic form, there are four adornments that apply to associations.

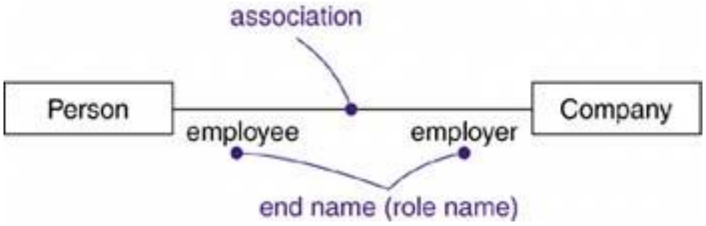
Name

An association can have a name, and you use that name to describe the nature of the relationship. So that there is no ambiguity about its meaning, you can give a direction to the name by providing a direction triangle that points in the direction you intend to read the name.



Role

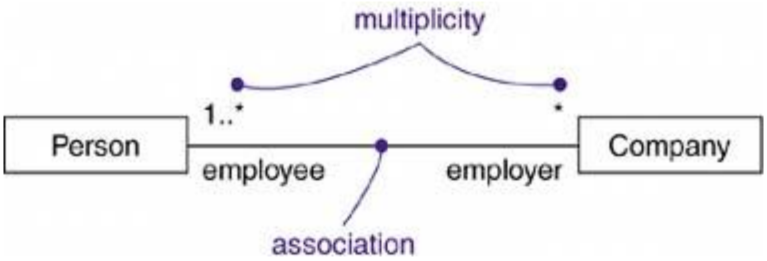
When a class participates in an association, it has a specific role that it plays in that relationship; a role is just the face the class at the far end of the association presents to the class at the near end of the association. You can explicitly name the role a class plays in an association. The role played by an end of an association is called an end name (in UML1, it was called a role name). the class Person playing the role of employee is associated with the class Company playing the role of employer.



Multiplicity

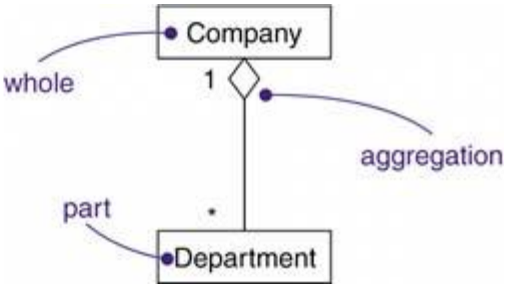
An association represents a structural relationship among objects. In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association. This "how many" is called the multiplicity of an association's role. It represents a range of integers specifying the possible size of the set of related objects.

The number of objects must be in the given range. You can show a multiplicity of exactly one (1), zero or one (0..1), many (0..*), or one or more (1..*). You can give an integer range (such as 2..5). You can even state an exact number (for example, 3, which is equivalent to 3..3).



Aggregation

A plain association between two classes represents a structural relationship between peers, meaning that both classes are conceptually at the same level, no one more important than the other. Sometimes you will want to model a "whole/part" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts"). This kind of relationship is called aggregation, which represents a "has-a" relationship



Common Modeling Techniques

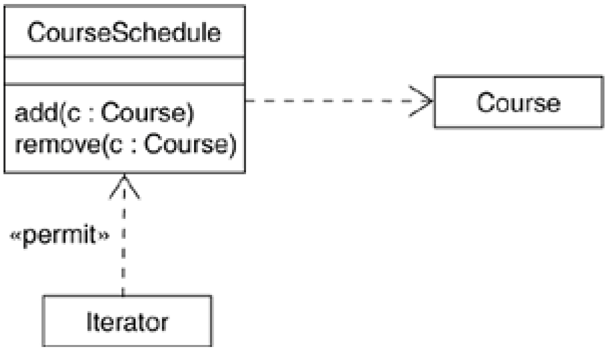
Modeling Simple Dependencies

A common kind of dependency relationship is the connection between a class that uses another class as a parameter to an operation.

To model this using relationship,

- Create a dependency pointing from the class with the operation to the class used as a parameter in the operation.

a set of classes drawn from a system that manages the assignment of students and instructors to courses in a university. This figure shows a dependency from CourseSchedule to Course, because Course is used in both the add and remove operations of CourseSchedule.

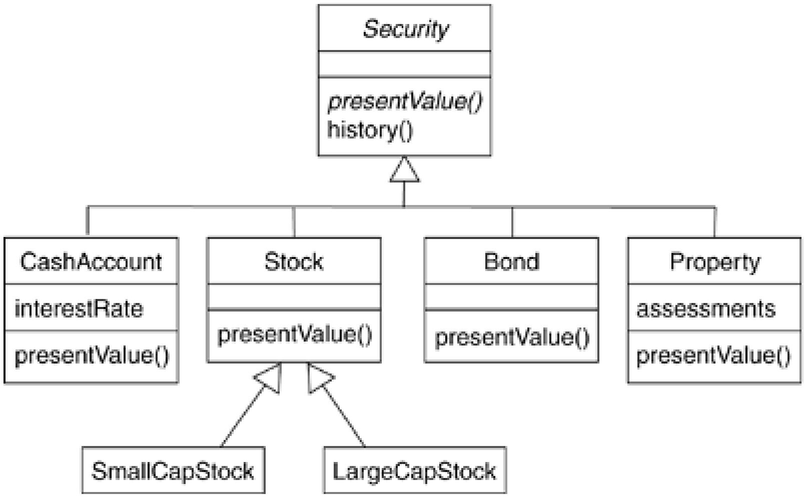


Modeling Single Inheritance

In modeling the vocabulary of your system, you will often run across classes that are structurally or behaviorally similar to others. You could model each of these as distinct and unrelated abstractions. A better way would be to extract any common structural and behavioral features and place them in more-general classes from which the specialized ones inherit.

To model inheritance relationships,

- Given a set of classes, look for responsibilities, attributes, and operations that are common to two or more classes.
- Elevate these common responsibilities, attributes, and operations to a more general class. If necessary, create a new class to which you can assign these elements (but be careful about introducing too many levels).
- Specify that the more-specific classes inherit from the more-general class by placing a generalization relationship that is drawn from each specialized class to its more-general parent.



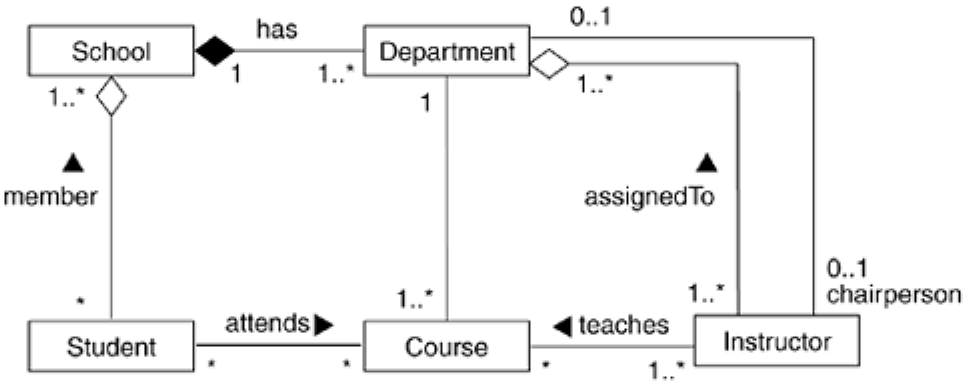
Modeling Structural Relationships

When you model with dependencies or generalization relationships, you may be modeling classes that represent different levels of importance or different levels of abstraction. Given a dependency between two classes, one class depends on another but the other class has no knowledge of the one.

To model structural relationships,

- For each pair of classes, if you need to navigate from objects of one to objects of another, specify an association between the two. This is a data-driven view of associations.
- For each pair of classes, if objects of one class need to interact with objects of the other class other than as local variables in a procedure or parameters to an operation, specify an association between the two. This is more of a behavior-driven view of associations.

- For each of these associations, specify a multiplicity (especially when the multiplicity is not *, which is the default), as well as role names (especially if they help to explain the model).
- If one of the classes in an association is structurally or organizationally a whole compared with the classes at the other end that look like parts, mark this as an aggregation by adorning the association at the end near the whole with a diamond.



3.Common Mechanisms:

Terms and Concepts

A *note* is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment.

A *stereotype* is an extension of the vocabulary of the UML, allowing you to create new kinds of building blocks similar to existing ones but specific to your problem. Graphically, a stereotype is rendered as a name enclosed by guillemets (French quotation marks of the form « »), placed above the name of another element.

Optionally the stereotyped element may be rendered by using a new icon associated with that stereotype.

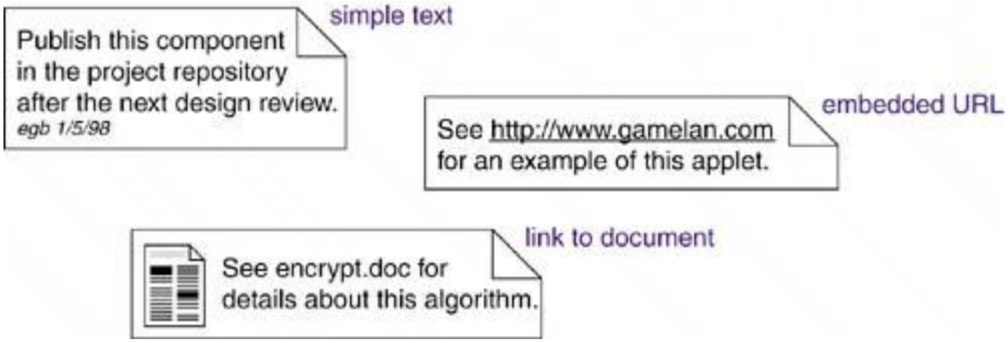
A *tagged value* is a property of a stereotype, allowing you to create new information in an element bearing that stereotype. Graphically, a tagged value is rendered as a string of the form name = value within a note attached to the object.

A *constraint* is a textual specification of the semantics of a UML element, allowing you to add new rules or to modify existing ones. Graphically, a constraint is rendered as a string enclosed by brackets and placed near the associated element or connected to that element or elements by dependency relationships. As an alternative, you can render a constraint in a note.

Notes

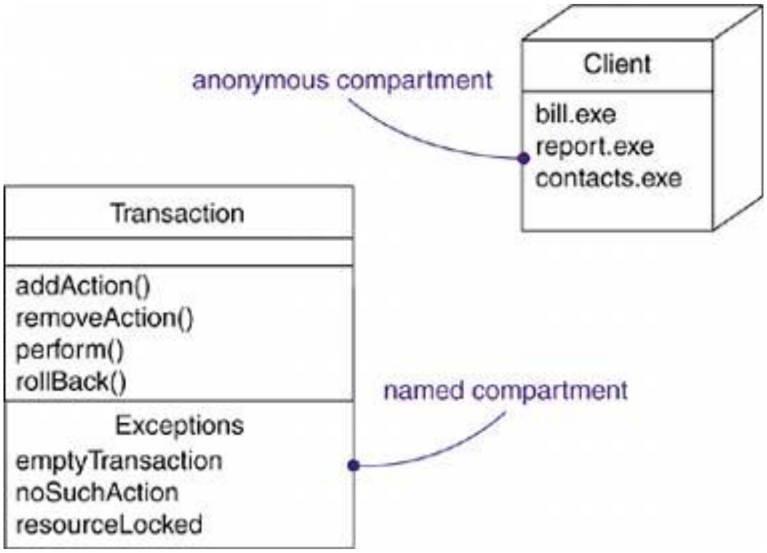
A note that renders a comment has no semantic impact, meaning that its contents do not alter the meaning of the model to which it is attached. This is why notes are used to specify things like requirements, observations, reviews, and explanations, in addition to rendering constraints.

A note may contain any combination of text or graphics


Publish this component
in the project repository
after the next design review.
egb 1/5/98
simple text
See http://www.gamelan.com
for an example of this applet.
embedded URL
See encrypt.doc for
details about this algorithm.
link to document

Other Adornments

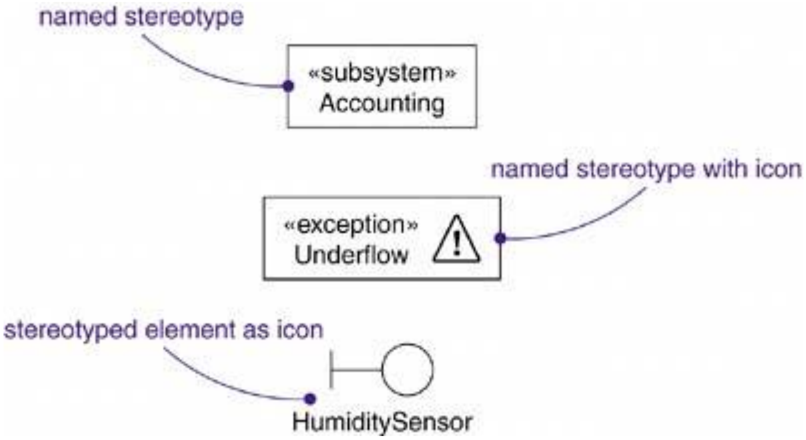
Adornments are textual or graphical items that are added to an element's basic notation and are used to visualize details from the element's specification



Stereotypes

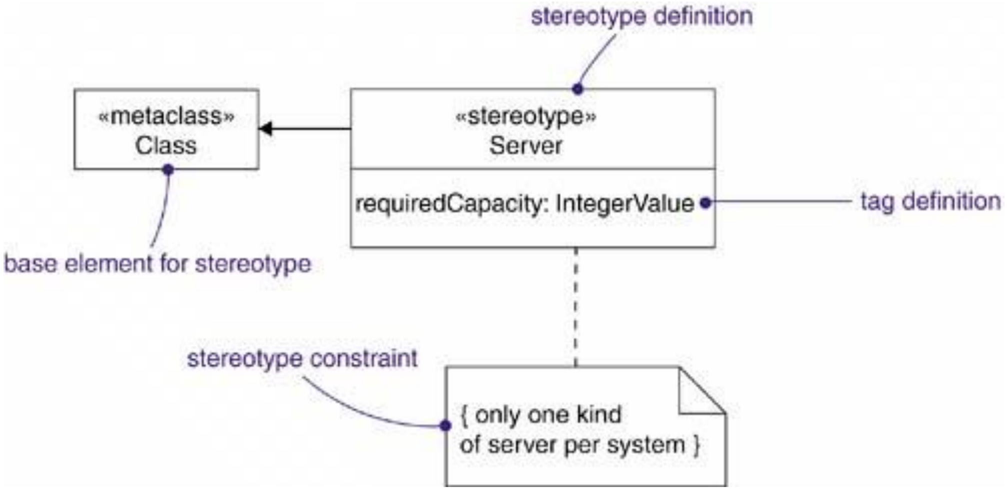
The UML provides a language for structural things, behavioral things, grouping things, and notational things. These four basic kinds of things address the overwhelming majority of the systems you'll need to model.

In its simplest form, a stereotype is rendered as a name enclosed by guillemets (for example, «name») and placed above the name of another element.



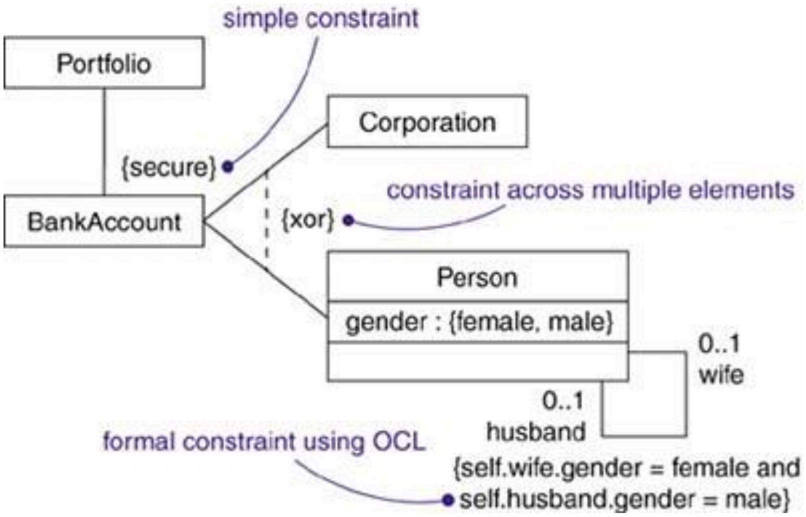
Tagged Values

Every thing in the UML has its own set of properties: classes have names, attributes, and operations; associations have names and two or more ends, each with its own properties; and so on. With stereotypes, you can add new things to the UML; with tagged values, you can add new properties to a stereotype.



Constraints

Everything in the UML has its own semantics. Generalization (usually, if you know what's good for you) implies the Liskov substitution principle, and multiple associations connected to one class denote distinct relationships. With constraints, you can add new semantics or extend existing rules. A constraint specifies conditions that a run-time configuration must satisfy to conform to the model.


simple constraint
Portfolio
Corporation
{secure}
constraint across multiple elements
BankAccount
{xor}
Person
gender : {female, male}
0..1
wife
0..1
husband
formal constraint using OCL
{self.wife.gender = female and
self.husband.gender = male}

- stereotype Specifies that the classifier is a stereotype that may be applied to other elements

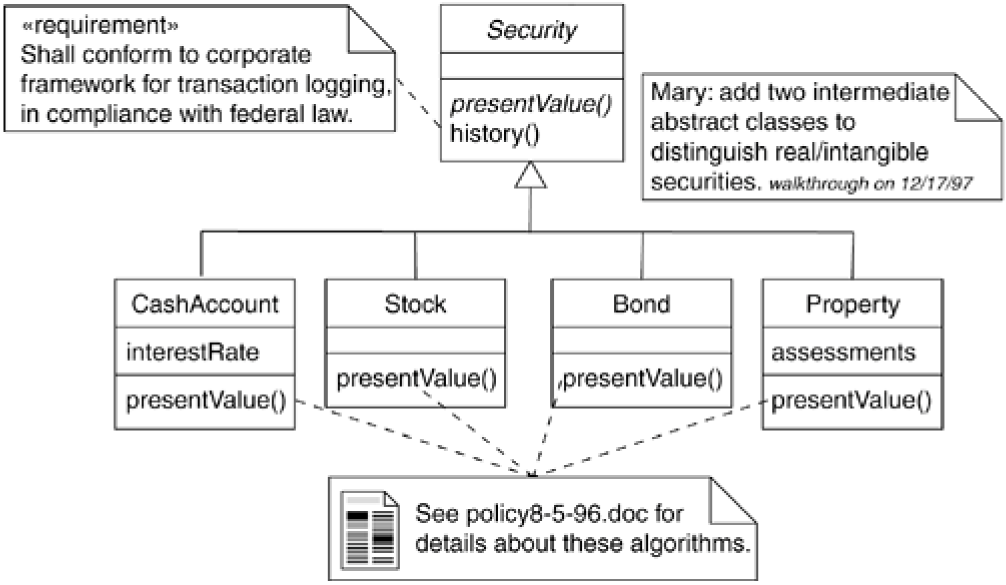
Common Modeling Techniques

Modeling Comments

The most common purpose for which you'll use notes is to write down free-form observations, reviews, or explanations.

To model a comment,

- Put your comment as text in a note and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a note to its elements using a dependency relationship.
- Remember that you can hide or make visible the elements of your model as you see fit. This means that you don't have to make your comments visible everywhere the elements to which it is attached are visible. Rather, expose your comments in your diagrams only insofar as you need to communicate that information in that context.
- If your comment is lengthy or involves something richer than plain text, consider putting your comment in an external document and linking or embedding that document in a note attached to your model.
- As your model evolves, keep those comments that record significant decisions that cannot be inferred from the model itself, andunless they are of historic interestdiscard the others.

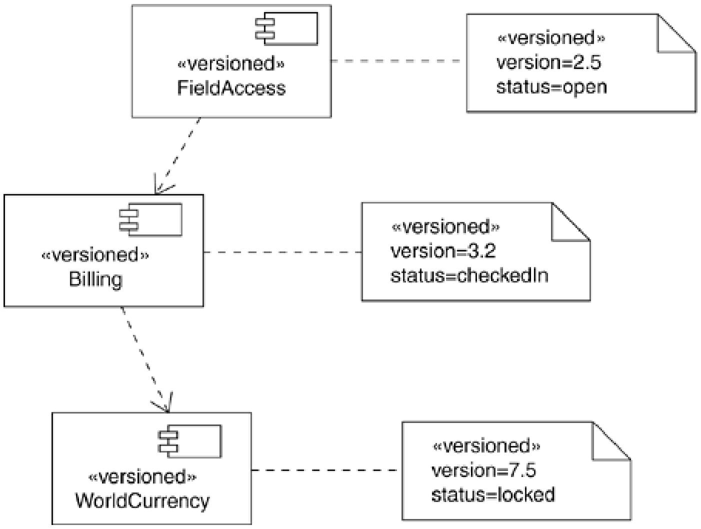

«requirement»
Shall conform to corporate framework for transaction logging, in compliance with federal law.
Security
presentValue()
history()
Mary: add two intermediate abstract classes to distinguish real/intangible securities. walkthrough on 12/17/97
CashAccount
interestRate
presentValue()
Stock
presentValue()
Bond
presentValue()
Property
assessments
presentValue()
See policy8-5-96.doc for details about these algorithms.

Modeling New Properties

The basic properties of the UML's building blocksattributes and operations for classes, the contents of packages

To model new properties,

- First, make sure there's not already a way to express what you want by using basic UML.
- If you re convinced there's no other way to express these semantics, define a stereotype and add the new properties to the stereotype. The rules of generalization applytagged values defined for one kind of stereotype apply to its children.

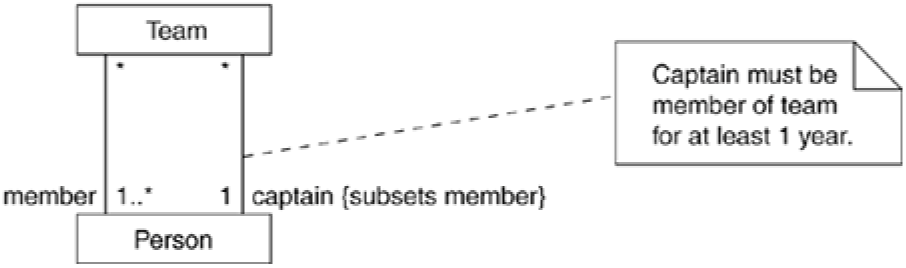


Modeling New Semantics

When you create a model using the UML, you work within the rules the UML lays down. However, if you find yourself needing to express new semantics about which the UML is silent or that you need to modify the UML's rules, then you need to write a constraint.

To model new semantics,

- First, make sure there's not already a way to express what you want by using basic UML.
- If you re convinced there's no other way to express these semantics, write your new semantics in a constraint placed near the element to which it refers. You can show a more explicit relationship by connecting a constraint to its elements using a dependency relationship.
- If you need to specify your semantics more precisely and formally, write your new semantics using OCL.



4.Diagrams:

Terms and Concepts

A *system* is a collection of subsystems organized to accomplish a purpose and described by a set of models, possibly from different viewpoints.

A *subsystem* is a grouping of elements, some of which constitute a specification of the behavior offered by the other contained elements.

A *model* is a semantically closed abstraction of a system, meaning that it represents a complete and self-consistent simplification of reality, created in order to better understand the system. In the context of architecture,

A *view* is a projection into the organization and structure of a system's model, focused on one aspect of that system.

A *diagram* is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).

In modeling real systems, no matter what the problem domain, you'll find yourself creating the same kinds of diagrams, because they represent common views into common models. Typically, you'll view the static parts of a system using one of the following diagrams.

1. Class diagram
2. Component diagram
3. Composite structure diagram
4. Object diagram
5. Deployment diagram
6. Artifact diagram

You'll often use five additional diagrams to view the dynamic parts of a system.

1. Use case diagram
2. Sequence diagram
3. Communication diagram
4. State diagram
5. Activity diagram

Structural Diagrams

The UML's structural diagrams exist to visualize, specify, construct, and document the static aspects of a system. You can think of the static aspects of a system as representing its relatively stable skeleton and scaffolding. Just as the static aspects of a house encompass the existence and placement of such things as walls, doors, windows, pipes, wires, and vents, so too do the static aspects of a software system encompass the existence and placement of such things as classes, interfaces, collaborations, components, and nodes.

The UML's structural diagrams are roughly organized around the major groups of things you'll find when modeling a system.

1.Class diagram	Classes, interfaces, and collaborations
2.Component diagram	Components
3.Object diagram	Objects
4.Deployment diagram	Nodes

Behavioral Diagrams

The UML's behavioral diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system. You can think of the dynamic aspects of a system as representing its changing parts. Just as the dynamic aspects of a house encompass airflow and traffic through the rooms of a house, so too do the dynamic aspects of a software system encompass such things as the flow of messages over time and the physical movement of components across a network.

The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.

1.Use case diagram	Organizes the behaviors of the system
2.Sequence diagram	Focuses on the time ordering of messages
3.Collaboration diagram	Focuses on the structural organization of objects that send and receive messages
4.State diagram	Focuses on the changing state of a system driven by events
5.Activity diagram	Focuses on the flow of control from activity to activity

Common Modeling Techniques

Modeling Different Views of a System

When you model a system from different views, you are in effect constructing your system simultaneously from multiple dimensions.

To model a system from different views,

- Decide which views you need to best express the architecture of your system and to expose the technical risks to your project. The five views of an architecture described earlier are a good starting point.
- For each of these views, decide which artifacts you need to create to capture the essential details of that view. For the most part, these artifacts will consist of various UML diagrams.
- As part of your process planning, decide which of these diagrams you'll want to put under some sort of formal or semi-formal control. These are the diagrams for which you'll want to schedule reviews and to preserve as documentation for the project.
- Allow room for diagrams that are thrown away. Such transitory diagrams are still useful for exploring the implications of your decisions and for experimenting with changes.

For example, if you are modeling a simple monolithic application that runs on a single machine, you might need only the following handful of diagrams.

■Use case view	Use case diagrams
■Design view	Class diagrams (for structural modeling)
■Interaction view	Interaction diagrams (for behavioral modeling)
■Implementation view	Composite structure diagrams
■Deployment view	None required

Similarly, if yours is a client/server system, you'll probably want to include component diagrams and deployment diagrams to model the physical details of your system.

Finally, if you are modeling a complex, distributed system, you'll need to employ the full range of the UML's diagrams in order to express the architecture of your system and the technical risks to your project, as in the following.

■Use case view	Use case diagrams
	Sequence diagrams
■Design view	Class diagrams (for structural modeling)
	Interaction diagrams (for behavioral modeling)
	State diagrams (for behavioral modeling)
	Activity diagrams
■Interaction view	Interaction diagrams (for behavioral modeling)
■Implementation view	Class diagrams
	Composite structure diagrams
■Deployment view	Deployment diagrams

Modeling Different Levels of Abstraction

Not only do you need to view a system from several angles, you'll also find people involved in development who need the same view of the system but at different levels of abstraction

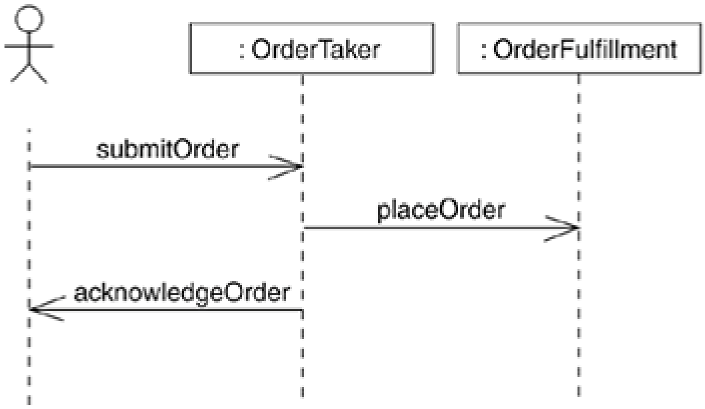
To model a system at different levels of abstraction by presenting diagrams with different levels of detail,

- Consider the needs of your readers, and start with a given model.
- If your reader is using the model to construct an implementation, she'll need diagrams that are at a lower level of abstraction, which means that they'll need to reveal a lot of detail. If she is using the model to present a conceptual model to an end user, she'll need diagrams that are at a higher level of abstraction, which means that they'll hide a lot of detail.
- Depending on where you land in this spectrum of low-to-high levels of abstraction, create a diagram at the right level of abstraction by hiding or revealing the following four categories of things from your model:
    1. Building blocks and relationships: Hide those that are not relevant to the intent of your diagram or the needs of your reader.
    2. Adornments: Reveal only the adornments of these building blocks and relationships that are essential to understanding your intent.

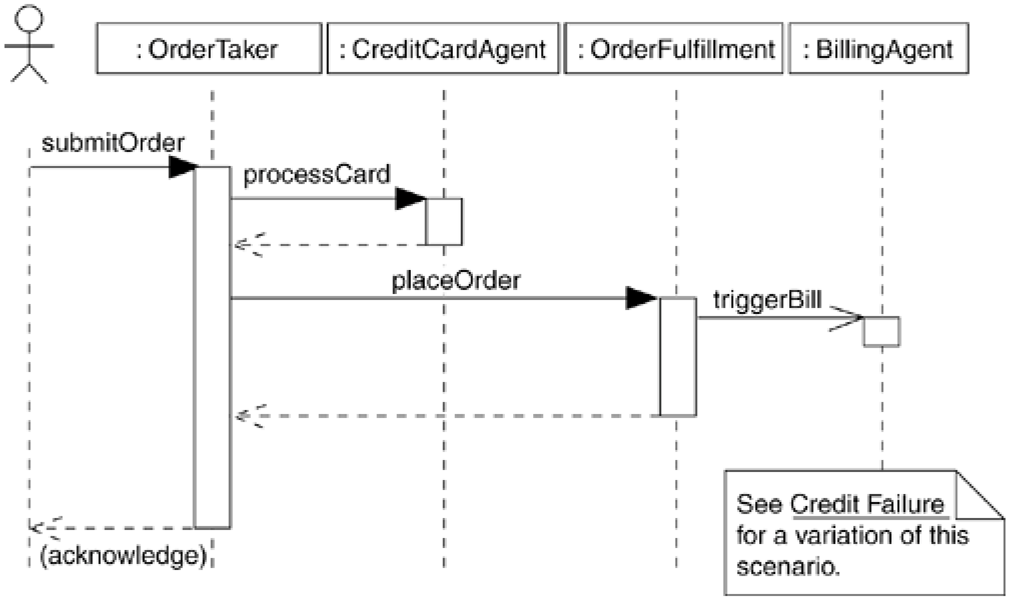
3. Flow: In the context of behavioral diagrams, expand only those messages or transitions that are essential to understanding your intent.
4. Stereotypes: In the context of stereotypes used to classify lists of things, such as attributes and operations, reveal only those stereotyped items that are essential to understanding your intent.

To model a system at different levels of abstraction by creating models at different levels of abstraction,

- Consider the needs of your readers and decide on the level of abstraction that each should view, forming a separate model for each level.
- In general, populate your models that are at a high level of abstraction with simple abstractions and your models that are at a low level of abstraction with detailed abstractions. Establish trace dependencies among the related elements of different models.
- In practice, if you follow the five views of an architecture, there are four common situations you'll encounter when modeling a system at different levels of abstraction:
  1. Use cases and their realization: Use cases in a use case model will trace to collaborations in a design model.
  2. Collaborations and their realization: Collaborations will trace to a society of classes that work together to carry out the collaboration.
  3. Components and their design: Components in an implementation model will trace to the elements in a design model.
  4. Nodes and their components: Nodes in a deployment model will trace to components in an implementation model.



Higher Level of Abstraction



Lower level of Abstraction

Modeling Complex Views

To model complex views,

- First, convince yourself that there is no meaningful way to present this information at a higher level of abstraction, perhaps eliding some parts of the diagram and retaining the detail in other parts.
- If you've hidden as much detail as you can and your diagram is still complex, consider grouping some of the elements in packages or in higher-level collaborations, then render only those packages or collaborations in your diagram.
- If your diagram is still complex, use notes and color as visual cues to draw the reader's attention to the points you want to make.
- If your diagram is still complex, print it in its entirety and hang it on a convenient large wall. You lose the interactivity that an online version of the diagram brings, but you can step back from the diagram and study it for common patterns.

1. ADVANCED STRUCTURAL MODELING

Terms and Concepts

A *class diagram* is a diagram that shows a set of classes, interfaces, and collaborations and their relationships. Graphically, a class diagram is a collection of vertices and arcs.

Common Properties

A class diagram is just a special kind of diagram and shares the same common properties as do all other diagramsa name and graphical content that are a projection into a model. What distinguishes a class diagram from other kinds of diagrams is its particular content.

Contents

Class diagrams commonly contain the following things:

- Classes
- Interfaces
- Dependency, generalization, and association relationships

Common Uses

You use class diagrams to model the static design view of a system. This view primarily supports the functional requirements of a systemthe services the system should provide to its end users.

When you model the static design view of a system, you'll typically use class diagrams in one of three ways.

1. To model the vocabulary of a system

Modeling the vocabulary of a system involves making a decision about which abstractions are a part of the system under consideration and which fall outside its boundaries. You use class diagrams to specify these abstractions and their responsibilities.

2. To model simple collaborations

A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. For example, when you re modeling the semantics of a transaction in a distributed system, you can't just stare at a single class to understand what's going on. Rather, these semantics are carried out by a set of classes that work together. You use class diagrams to visualize and specify this set of classes and their relationships.

3. To model a logical database schema

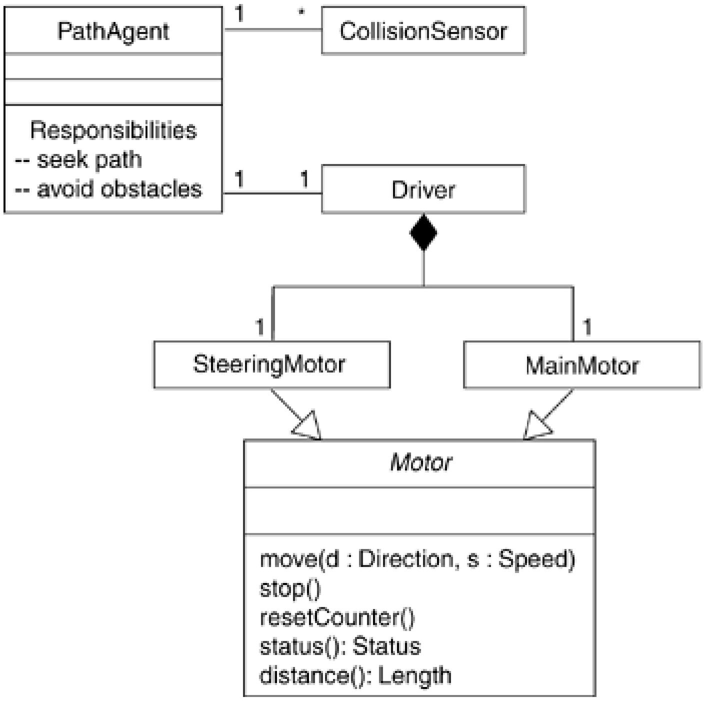
Think of a schema as the blueprint for the conceptual design of a database. In many domains, you'll want to store persistent information in a relational database or in an object-oriented database. You can model schemas for these databases using class diagrams.

Common Modeling Techniques

Modeling Simple Collaborations

To model a collaboration,

- Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.
- For each mechanism, identify the classes, interfaces, and other collaborations that participate in this collaboration. Identify the relationships among these things as well.
- Use scenarios to walk through these things. Along the way, you'll discover parts of your model that were missing and parts that were just plain semantically wrong.
- Be sure to populate these elements with their contents. For classes, start with getting a good balance of responsibilities. Then, over time, turn these into concrete attributes and operations.

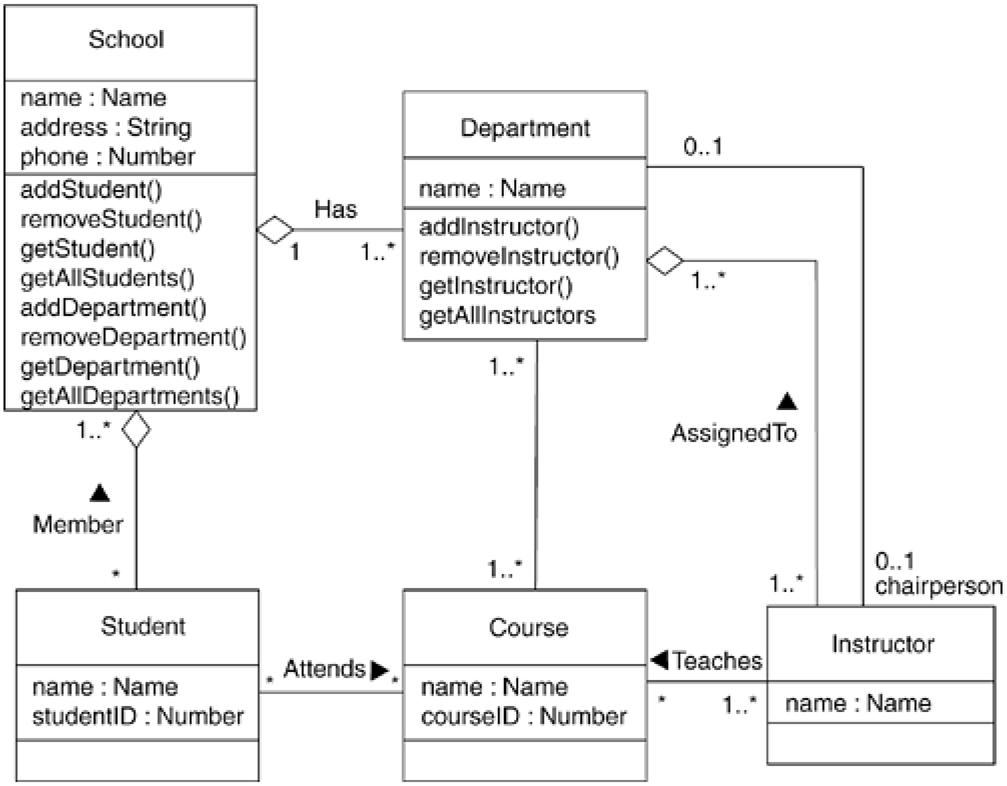


Modeling a Logical Database Schema

To model a schema,

- Identify those classes in your model whose state must transcend the lifetime of their applications.
- Create a class diagram that contains these classes. You can define your own set of stereotypes and tagged values to address database-specific details.

- Expand the structural details of these classes. In general, this means specifying the details of their attributes and focusing on the associations and their multiplicities that relate these classes.
- Watch for common patterns that complicate physical database design, such as cyclic associations and one-to-one associations. Where necessary, create intermediate abstractions to simplify your logical structure.
- Consider also the behavior of these classes by expanding operations that are important for data access and data integrity. In general, to provide a better separation of concerns, business rules concerned with the manipulation of sets of these objects should be encapsulated in a layer above these persistent classes.
- Where possible, use tools to help you transform your logical design into a physical design.

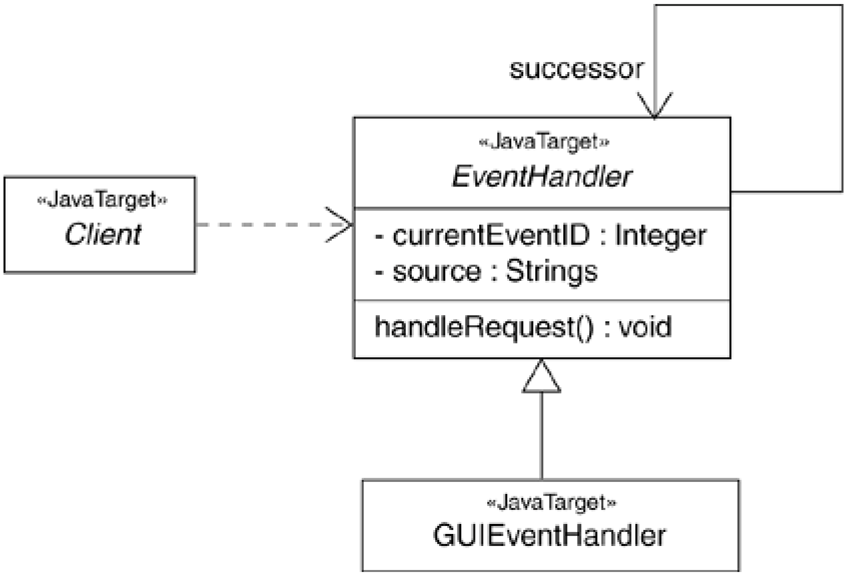


Forward and Reverse Engineering

Forward engineering is the process of transforming a model into code through a mapping to an implementation language. Forward engineering results in a loss of information, because models written in the UML are semantically richer than any current object-oriented programming language. In fact, this is a major reason why you need models in addition to code. Structural features, such as collaborations, and behavioral features, such as interactions, can be visualized clearly in the UML, but not so clearly from raw code.

To forward engineer a class diagram,

- Identify the rules for mapping to your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
- Depending on the semantics of the languages you choose, you may want to constrain your use of certain UML features. For example, the UML permits you to model multiple inheritance, but Smalltalk permits only single inheritance. You can choose to prohibit developers from modeling with multiple inheritance (which makes your models language-dependent), or you can develop idioms that transform these richer features into the implementation language (which makes the mapping more complex).
- Use tagged values to guide implementation choices in your target language. You can do this at the level of individual classes if you need precise control. You can also do so at a higher level, such as with collaborations or packages.
- Use tools to generate code.



All of these classes specify a mapping to Java, as noted in their stereotype. Forward engineering the classes in this diagram to Java is straightforward, using a tool. Forward engineering the class EventHandler yields the following code.

```
public abstract class EventHandler {

  EventHandler successor;
  private Integer currentEventID;
  private String source;

  EventHandler() {}
  public void handleRequest() {}

}
```

Reverse engineering is the process of transforming code into a model through a mapping from a specific implementation language. Reverse engineering results in a flood of information, some of which is at a lower level of detail than you'll need to build useful models. At the same time, reverse engineering is incomplete. There is a loss of information when forward engineering models into code, and so you can't completely recreate a model from code unless your tools encode information in the source comments that goes beyond the semantics of the implementation language.

To reverse engineer a class diagram,

- Identify the rules for mapping from your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
- Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or modify an existing one that was previously forward engineered. It is unreasonable to expect to reverse engineer a single concise model from a large body of code. You need to select portion of the code and build the model from the bottom.
- Using your tool, create a class diagram by querying the model. For example, you might start with one or more classes, then expand the diagram by following specific relationships or other neighboring classes. Expose or hide details of the contents of this class diagram as necessary to communicate your intent.
- Manually add design information to the model to express the intent of the design that is missing or hidden in the code.

II. ADVANCED STRUCTURAL MODELING

1. **Advanced Classes**
2. **Advanced Relationships**
3. **Interface, Type and Role**
4. **Packages**

1. Advanced Classes:

Terms and Concepts

A *classifier* is a mechanism that describes structural and behavioral features. Classifiers include classes, associations, interfaces, datatypes, signals, components, nodes, use cases, and subsystems.

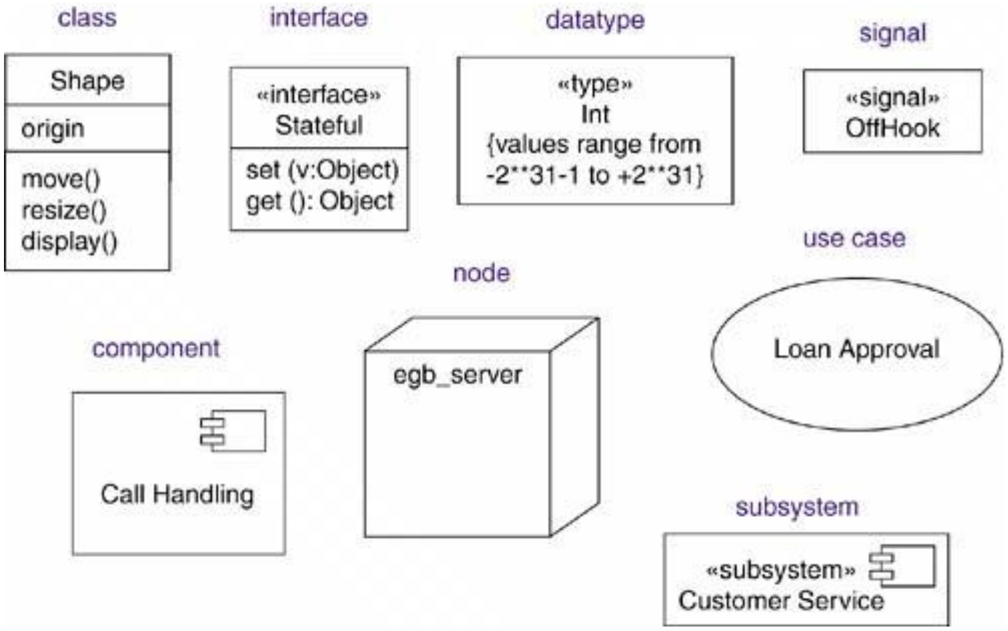
Classifiers

When you model, you'll discover abstractions that represent things in the real world and things in your solution. For example, if you are building a Web-based ordering system, the vocabulary of your project will likely include a Customer class (representing people who order products) and a

TRansaction class (an implementation artifact, representing an atomic action). In the deployed system, you might have a Pricing component, with instances living on every client node. Each of these abstractions will have instances; separating the essence and the instance of the things in your world is an important part of modeling.

The most important kind of classifier in the UML is the class. A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Classes are not the only kind of classifier, however. The UML provides a number of other kinds of classifiers to help you model.

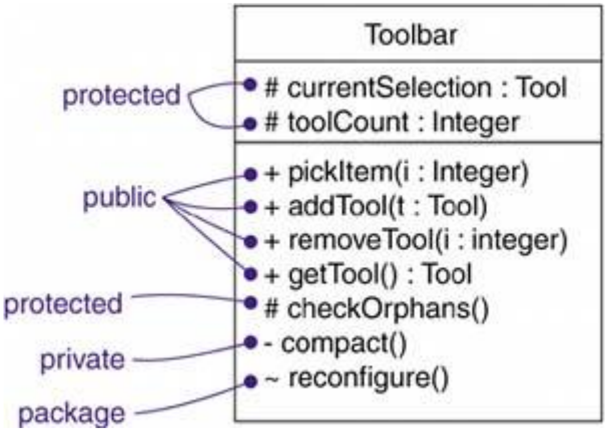
■Interface	A collection of operations that are used to specify a service of a class or a component
■Datatype	A type whose values are immutable, including primitive built-in types (such as numbers and strings) as well as enumeration types (such as Boolean)
■ Association	A description of a set of links, each of which relates two or more objects.
■Signal	The specification of an asynchronous message communicated between instances
■ Component	A modular part of a system that hides its implementation behind a set of external interfaces
■Node	A physical element that exists at run time and that represents a computational resource, generally having at least some memory and often processing capability
■Use case	A description of a set of a sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor
■Subsystem	A component that represents a major part of a system



Visibility

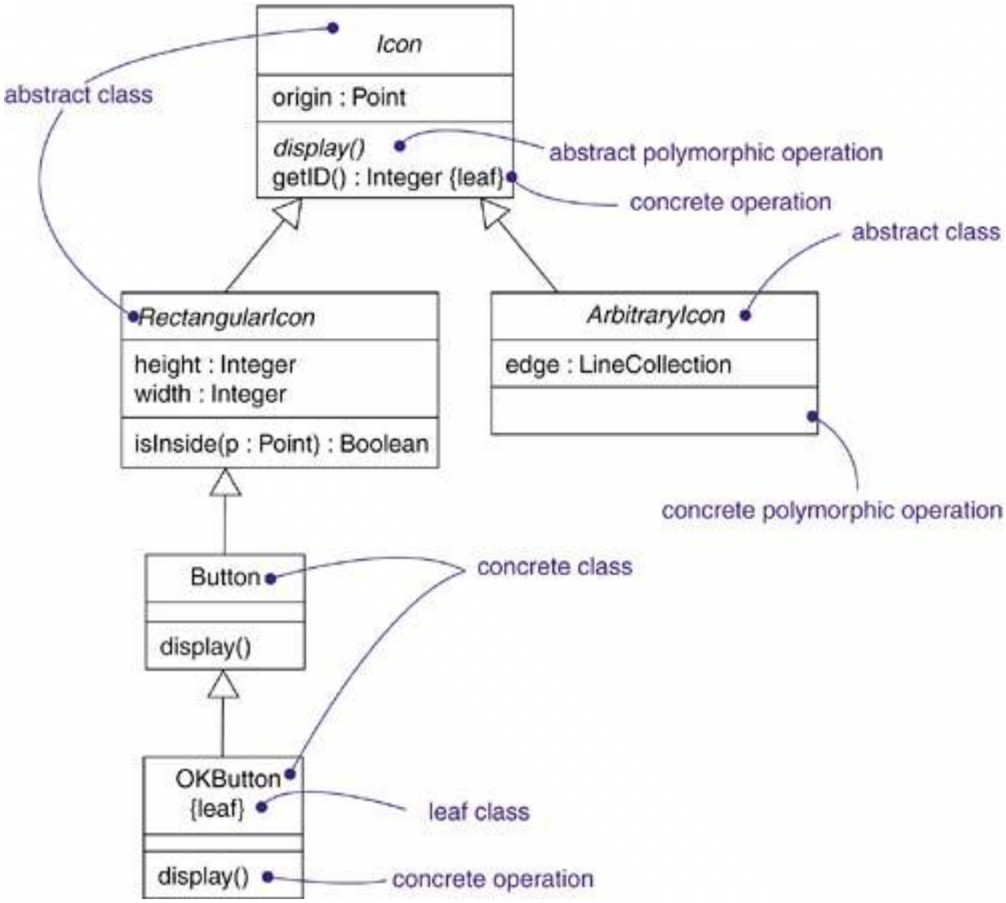
One of the design details you can specify for an attribute or operation is visibility. The visibility of a feature specifies whether it can be used by other classifiers. In the UML, you can specify any of four levels of visibility.

1. public	Any outside classifier with visibility to the given classifier can use the feature; specified by prepending the symbol +.
2. protected	Any descendant of the classifier can use the feature; specified by prepending the symbol #.
3. private	Only the classifier itself can use the feature; specified by prepending the symbol -.
3. package	Only classifiers declared in the same package can use the feature; specified by prepending the symbol ~.



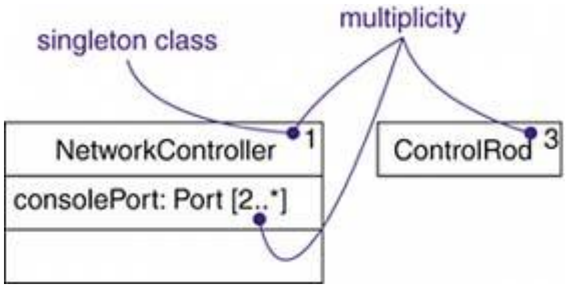
Abstract, Leaf, and Polymorphic Elements

You use generalization relationships to model a lattice of classes, with more-generalized abstractions at the top of the hierarchy and more-specific ones at the bottom. Within these hierarchies, it's common to specify that certain classes are abstractmeaning that they may not have any direct instances. In the UML, you specify that a class is abstract by writing its name in italics. Icon, RectangularIcon, and ArbitraryIcon are all abstract classes. By contrast, a concrete class (such as Button and OKButton) may have direct instances.



Multiplicity

Whenever you use a class, it's reasonable to assume that there may be any number of instances of that class (unless, of course, it is an abstract class and may not have any direct instances, although there may be any number of instances of its concrete children).



Attributes

At the most abstract level, when you model a class's structural features (that is, its attributes), you simply write each attribute's name.

visibility] name

[':' type] ['[' multiplicity] ']']
['=' initial-value]
[property-string {',' property-string}]

For example, the following are all legal attribute declarations:

■origin	Name only
■+ origin	Visibility and name
■origin : Point	Name and type
■name : String[0..1]	Name, type, and multiplicity
■origin : Point = (0,0)	Name, type, and initial value
■id: Integer {readonly}	Name and property

Operations

At the most abstract level, when you model a class's behavioral features. ou can also specify the parameters, return type, concurrency semantics, and other properties of each operation. Collectively, the name of an operation plus its parameters (including its return type, if any) is called the operation's signature.

[visibility] name ['(' parameter-list ')']
[':' return-type]
[property-string {',' property-string}]

For example, the following are all legal operation declarations:

■display	Name only
■+ display	Visibility and name
■set(n : Name, s : String)	Name and parameters
■getID() : Integer	Name and return type
■restart() {guarded}	Name and property

In an operation's signature, you may provide zero or more parameters, each of which follows the syntax

[direction] name : type [= default-value]

Direction may be any of the following values:

in	An input parameter; may not be modified
out	An output parameter; may be modified to communicate information to the caller
inout	An input parameter; may be modified to communicate information to the caller

In addition to the leaf and abstract properties described earlier, there are defined properties that you can use with operations.

1. query	Execution of the operation leaves the state of the system unchanged. In other words, the operation is a pure function that has no side effects.
2. sequential	Callers must coordinate outside the object so that only one flow is in the object at a time. In the presence of multiple flows of control, the semantics and integrity of the object cannot be guaranteed.
3. guarded	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by sequentializing all calls to all of the object's guarded operations. In effect, exactly one operation at a time can be invoked on the object, reducing this to sequential semantics.
4. concurrent	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by treating the operation as atomic. Multiple calls from concurrent flows of control may occur simultaneously to one object on any concurrent operation, and all may proceed concurrently with correct semantics; concurrent operations must be designed so that they perform correctly in case of a concurrent sequential or guarded operation on the same object.
5. static	The operation does not have an implicit parameter for the target object; it behaves like a traditional global procedure.

Template Classes

A template is a parameterized element. In such languages as C++ and Ada, you can write template classes, each of which defines a family of classes.

A template may include slots for classes, objects, and values, and these slots serve as the template's parameters. You can't use a template directly; you have to instantiate it first. Instantiation involves binding these formal template parameters to actual ones. For a template class, the result is a concrete class that can be used just like any ordinary class.

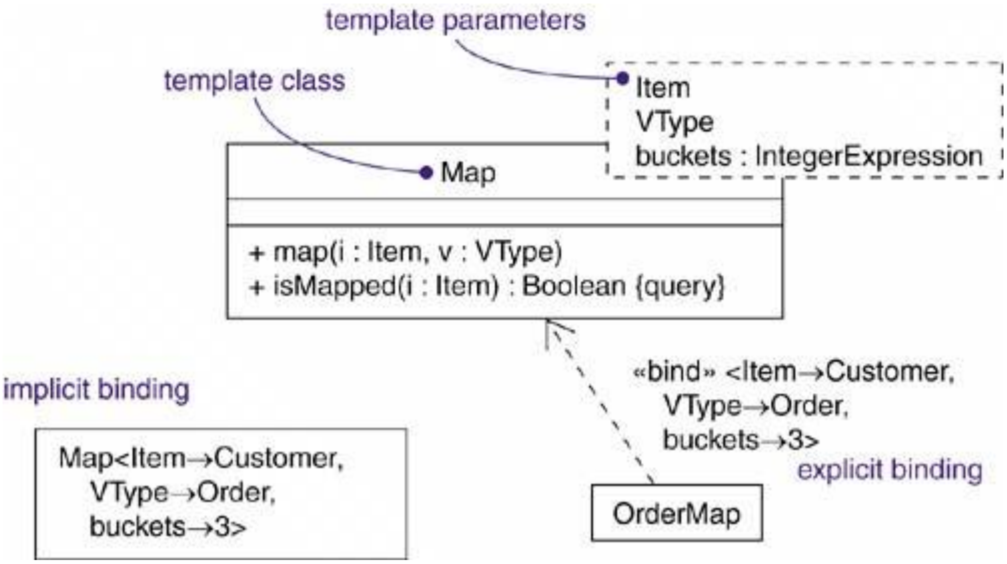
The most common use of template classes is to specify containers that can be instantiated for specific elements, making them type-safe. For example, the following C++ code fragment declares a parameterized Map class.

```
template<class Item, class VType, int Buckets>
```

```
class Map {
public:
 virtual map(const Item&, const VType&);
 virtual Boolean isMappen(const Item&) const;
 ...
};
```

You might then instantiate this template to map Customer objects to Order objects.

```
m : Map<Customer, Order, 3>;
```



Standard Elements

All of the UML's extensibility mechanisms apply to classes.

The UML defines four standard stereotypes that apply to classes.

1. metaclass	Specifies a classifier whose objects are all classes
2. powertype	Specifies a classifier whose objects are classes that are the children of a given parent class
3.stereotype	Specifies that the classifier is a stereotype that may be applied to other elements
4. utility	Specifies a class whose attributes and operations are all static scoped

Common Modeling Techniques

Modeling the Semantics of a Class

To model the semantics of a class, choose among the following possibilities, arranged from informal to formal.

- Specify the responsibilities of the class. A responsibility is a contract or obligation of a type or class and is rendered in a note attached to the class, or in an extra compartment in the class icon.
- Specify the semantics of the class as a whole using structured text, rendered in a note (stereotyped as semantics) attached to the class.
- Specify the body of each method using structured text or a programming language, rendered in a note attached to the operation by a dependency relationship.
- Specify the pre- and postconditions of each operation, plus the invariants of the class as a whole, using structured text. These elements are rendered in notes (stereotyped as precondition, postcondition, and invariant) attached to the operation or class by a dependency relationship.
- Specify a state machine for the class. A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- Specify internal structure of the class.
- Specify a collaboration that represents the class. A collaboration is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. A collaboration has a structural part as well as a dynamic part, so you can use collaborations to specify all dimensions of a class's semantics.
- Specify the pre- and postconditions of each operation, plus the invariants of the class as a whole, using a formal language such as OCL.

Pragmatically, you'll end up doing some combination of these approaches for the different abstractions in your system.

2.Advanced Relationships

Terms and Concepts

A *relationship* is a connection among things. In object-oriented modeling, the four most important relationships are dependencies, generalizations, associations, and realizations. Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the different relationships.

Dependencies

A *dependency* is a using relationship, specifying that a change in the specification of one thing (for example, class SetTopController) may affect another thing that uses it (for example, class ChannelIterator), but not the reverse. Graphically, a dependency is rendered as a dashed line, directed to the thing that is depended on. Apply dependencies when you want to show one thing using another.

A plain, unadorned dependency relationship is sufficient for most of the using relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines a number of stereotypes that may be applied to dependency relationships. There are a number of stereotypes, which can be organized into several groups.

First, there are stereotypes that apply to dependency relationships among classes and objects in class diagrams.

1. bind	Specifies that the source instantiates the target template using the given actual parameters
2.derive	Specifies that the source may be computed from the target
3.permit	Specifies that the source is given special visibility into the target
4.instanceOf	Specifies that the source object is an instance of the target classifier. Ordinarily shown using text notation in the form source : Target
5.instantiate	Specifies that the source creates instances of the target
6.powertype	Specifies that the target is a powertype of the source; a powertype is a classifier whose objects are the children of a given parent
7. refine	Specifies that the source is at a finer degree of
8. use	Specifies that the semantics of the source element depends on the semantics of the public part of the target

There are two stereotypes that apply to dependency relationships among packages.

1.import	Specifies that the public contents of the target package enter the public namespace of the source, as if they had been declared in the source.
2.access	Specifies that the public contents of the target package enter the private namespace of the source. The unqualified names may be used within the source, but they may not be re-exported.

Two stereotypes apply to dependency relationships among use cases:

1. extend	Specifies that the target use case extends the behavior of the source
2.include	Specifies that the source use case explicitly incorporates the behavior of another use case at a location specified by the source

One stereotype you'll encounter in the context of interactions among objects is

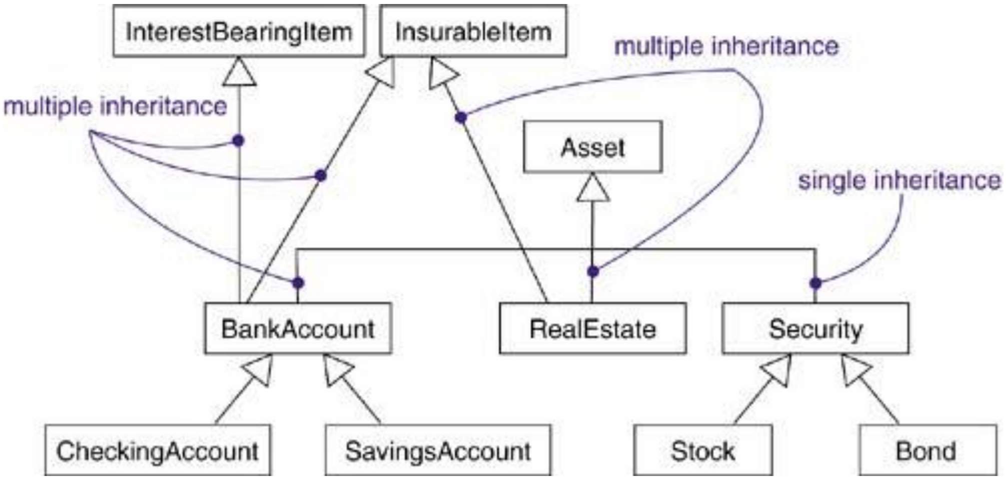
1. send	Specifies that the source class sends the target event

Finally, one stereotype that you'll encounter in the context of organizing the elements of your system into subsystems and models is

1.TRace	Specifies that the target is a historical predecessor of the source from an earlier stage of development

Generalizations

A *generalization* is a relationship between a general classifier (called the superclass or parent) and a more specific classifier (called the subclass or child). For example, you might encounter the general class Window with its more specific subclass, MultiPaneWindow. With a generalization relationship from the child to the parent, the child (MultiPaneWindow) will inherit all the structure and behavior of the parent (Window).



A plain, unadorned generalization relationship is sufficient for most of the inheritance relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines four constraints that may be applied to generalization relationships:

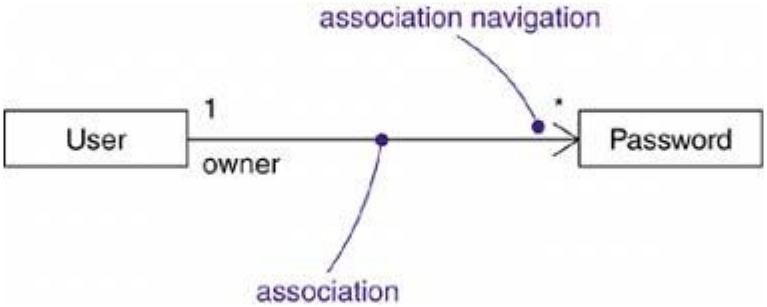
1. complete	Specifies that all children in the generalization have been specified in the model (although some may be elided in the diagram) and that no additional children are permitted
2.incomplete	Specifies that not all children in the generalization have been specified (even if some are elided) and that additional children are permitted
3. disjoint	Specifies that objects of the parent may have no more than one of the children as a type. For example, class Person can be specialized into disjoint classes Man and Woman.
4.overlapping	Specifies that objects of the parent may have more than one of the children as a type. For example, class Vehicle can be specialized into overlapping subclasses LandVehicle and WaterVehicle (an amphibious vehicle is both).

Associations

An *association* is a structural relationship, specifying that objects of one thing are connected to objects of another. For example, a Library class might have a one-to-many association to a Book class, indicating that each Book instance is owned by one Library instance.

Navigation

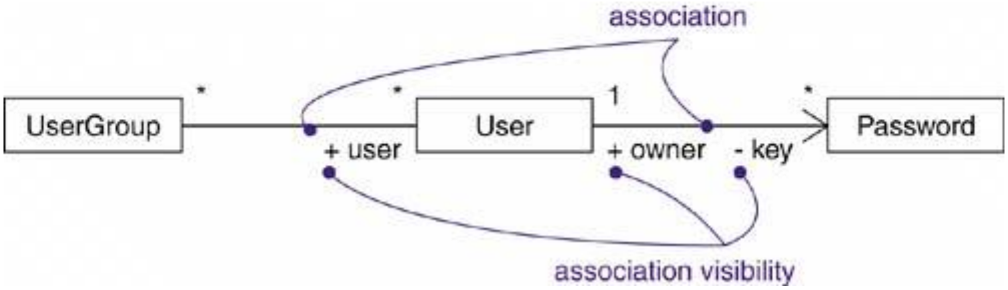
Given a plain, unadorned association between two classes, such as Book and Library, it's possible to navigate from objects of one kind to objects of the other kind. Unless otherwise specified, navigation across an association is bidirectional.



Visibility

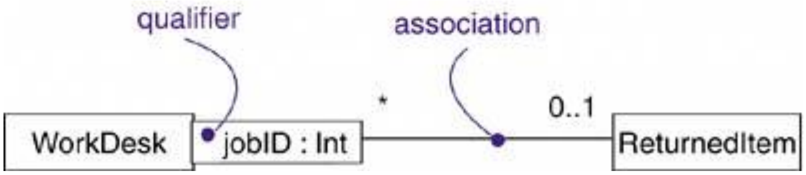
Given an association between two classes, objects of one class can see and navigate to objects of the other unless otherwise restricted by an explicit statement of navigation. However, there are circumstances in which you'll want to limit the visibility across that association relative to objects outside the association.

There is an association between UserGroup and User and another between User and Password. Given a User object, it's possible to identify its corresponding Password objects. However, a Password is private to a User, so it shouldn't be accessible from the outside (unless, of course, the User explicitly exposes access to the Password,



Qualification

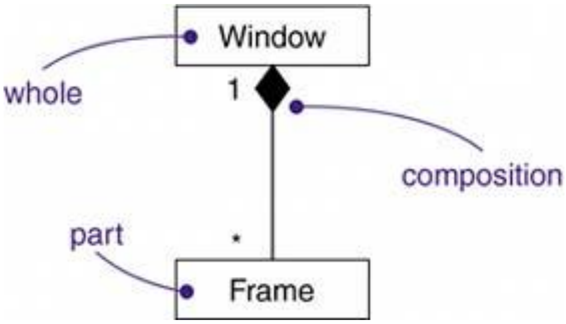
In the context of an association, one of the most common modeling idioms you'll encounter is the problem of lookup. Given an object at one end of an association, how do you identify an object or set of objects at the other end? For example, consider the problem of modeling a work desk at a manufacturing site at which returned items are processed to be fixed



Composition

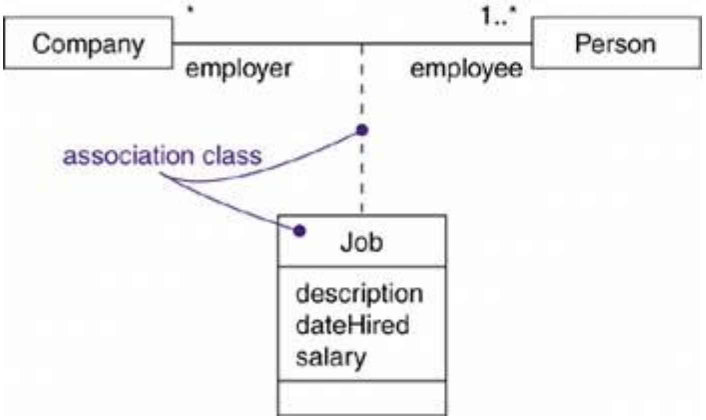
Aggregation turns out to be a simple concept with some fairly deep semantics. Simple aggregation is entirely conceptual and does nothing more than distinguish a "whole" from a "part." Simple aggregation does not change the meaning of navigation across the association between the whole and its parts, nor does it link the lifetimes of the whole and its parts.

in a composite aggregation, an object may be a part of only one composite at a time. For example, in a windowing system, a Frame belongs to exactly one Window. This is in contrast to simple aggregation, in which a part may be shared by several wholes. For example, in the model of a house, a Wall may be a part of one or more Room objects.



Association Classes

In an association between two classes, the association itself might have properties. For example, in an employer/employee relationship between a Company and a Person, there is a Job that represents the properties of that relationship that apply to exactly one pairing of the Person and Company. It wouldn't be appropriate to model this situation with a Company to Job association together with a Job to Person association.



Constraints

These simple and advanced properties of associations are sufficient for most of the structural relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines five constraints that may be applied to association relationships.

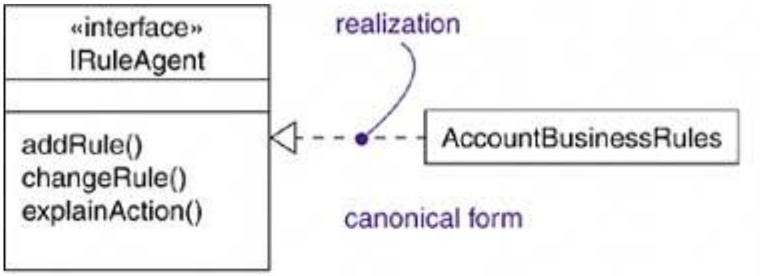
First, you can specify whether the objects at one end of an association (with a multiplicity greater than one) are ordered or unordered.

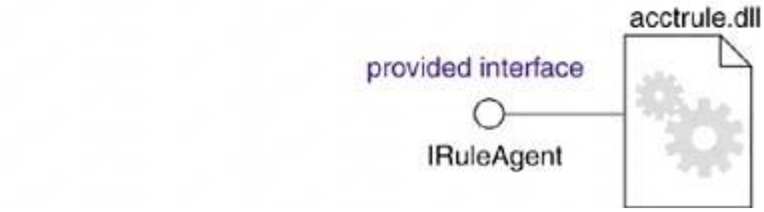
1. ordered	Specifies that the set of objects at one end of an association are in an explicit order.
2. set	The objects are unique with no duplicates.
3. bag	The objects are non-unique, may be duplicates.
4. ordered set	The objects are unique but ordered.
5. list or sequence	The objects are ordered, may be duplicates.
6. readonly	A link, once added from an object on the opposite end of the association, may not be modified or deleted. The default in the absence of this constraint is unlimited changeability.

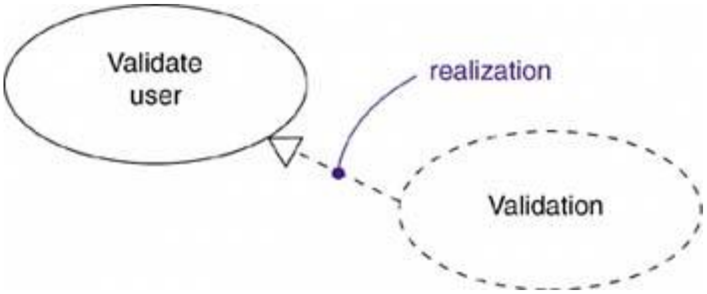
Realizations

A *realization* is a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out. Graphically, a realization is rendered as a dashed directed line with a large open arrowhead pointing to the classifier that specifies the contract.

Realization is different enough from dependency, generalization, and association relationships that it is treated as a separate kind of relationship. Semantically, realization is somewhat of a cross between dependency and generalization, and its notation is a combination of the notation for dependency and generalization.







Common Modeling Techniques

Modeling Webs of Relationships

When you model the vocabulary of a complex system, you may encounter dozens, if not hundreds or thousands, of classes, interfaces, components, nodes, and use cases.

When you model these webs of relationships,

- Don't begin in isolation. Apply use cases and scenarios to drive your discovery of the relationships among a set of abstractions.
- In general, start by modeling the structural relationships that are present. These reflect the static view of the system and are therefore fairly tangible.
- Next, identify opportunities for generalization/specialization relationships; use multiple inheritance sparingly.
- Only after completing the preceding steps should you look for dependencies; they generally represent more-subtle forms of semantic connection.
- For each kind of relationship, start with its basic form and apply advanced features only as absolutely necessary to express your intent.
- Remember that it is both undesirable and unnecessary to model all relationships among a set of abstractions in a single diagram or view. Rather, build up your system's relationships by considering different views on the system. Highlight interesting sets of relationships in individual diagrams.

3.Interface, Types and Roles:

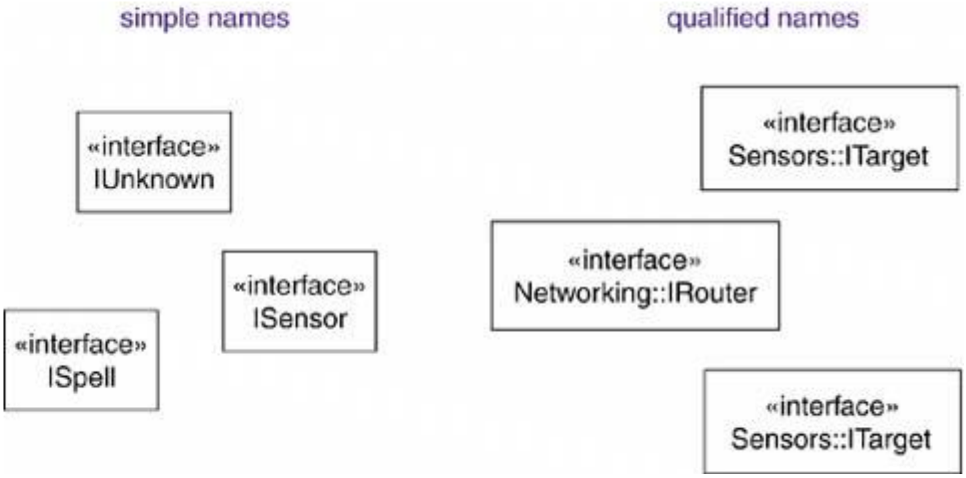
Terms and Concepts

An *interface* is a collection of operations that are used to specify a service of a class or a component. A *type* is a stereotype of a class used to specify a domain of objects, together with the operations (but not the methods) applicable to the object. A *role* is the behavior of an entity participating in a particular context.

Graphically, an interface may be rendered as a stereotyped class in order to expose its operations and other properties.

Names

Every interface must have a name that distinguishes it from other interfaces. A *name* is a textual string. That name alone is known as a simple name; a path name is the interface name prefixed by the name of the package in which that interface lives.

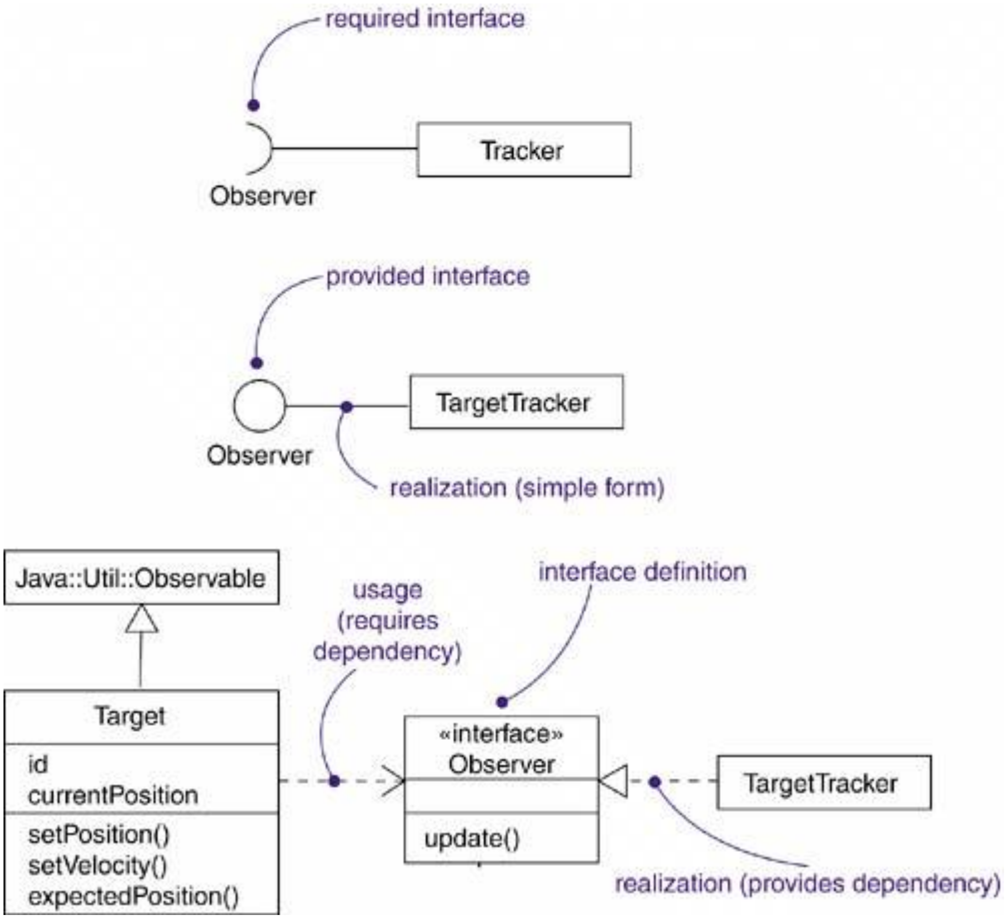

simple names
qualified names
«interface» IUnknown
«interface» Sensors::ITarget
«interface» ISensor
«interface» Networking::IRouter
«interface» ISpell
«interface» Sensors::ITarget

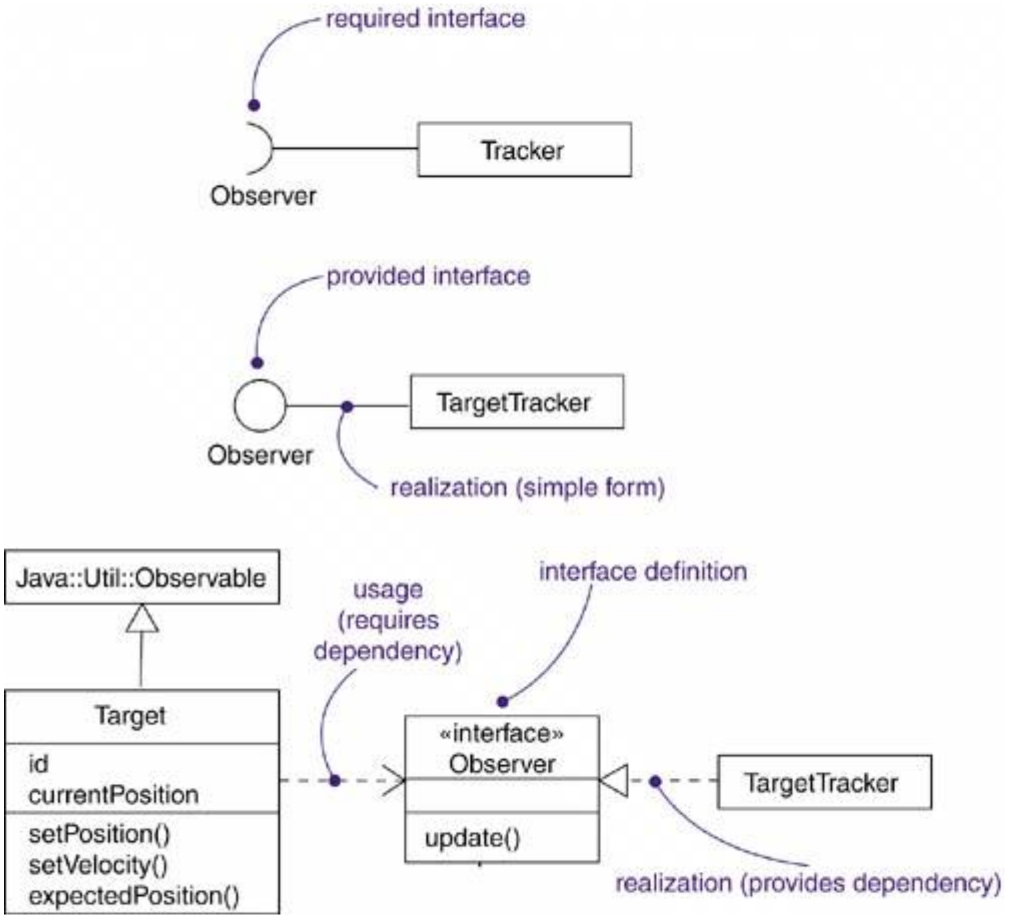
Operations

An interface is a named collection of operations used to specify a service of a class or of a component. Unlike classes or types, interfaces do not specify any implementation (so they may not include any methods, which provide the implementation of an operation). Like a class, an interface may have any number of operations. These operations may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.

Relationships

Like a class, an interface may participate in generalization, association, and dependency relationships. In addition, an interface may participate in realization relationships. Realization is a semantic relationship between two classifiers in which one classifier specifies a contract that another classifier guarantees to carry out.


required interface
Tracker
Observer
provided interface
TargetTracker
Observer
realization (simple form)
Java::Util::Observable
usage
(requires
dependency)
interface definition
Target
id
currentPosition
setPosition()
setVelocity()
expectedPosition()
«interface»
Observer
update()
TargetTracker
realization (provides dependency)



Understanding an Interface

When you are handed an interface, the first thing you'll see is a set of operations that specify a service of a class or a component. Look a little deeper and you'll see the full signature of those operations, along with any of their special properties, such as visibility, scope, and concurrency semantics.

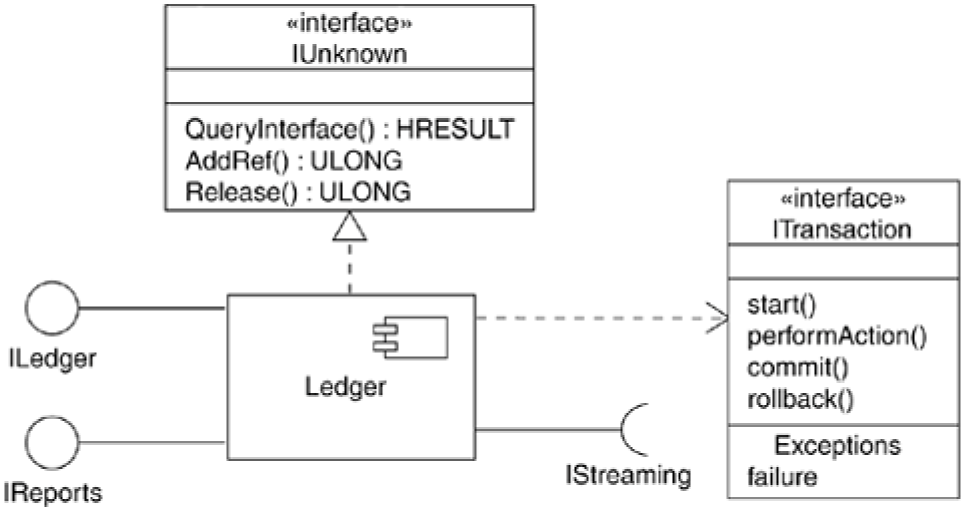
Common Modeling Techniques

Modeling the Seams in a System

o model the seams in a system,

- Within the collection of classes and components in your system, draw a line around those that tend to be tightly coupled relative to other sets of classes and components.
- Refine your grouping by considering the impact of change. Classes or components that tend to change together should be grouped together as collaborations.
- Consider the operations and the signals that cross these boundaries, from instances of one set of classes or components to instances of other sets of classes and components.
- Package logically related sets of these operations and signals as interfaces.

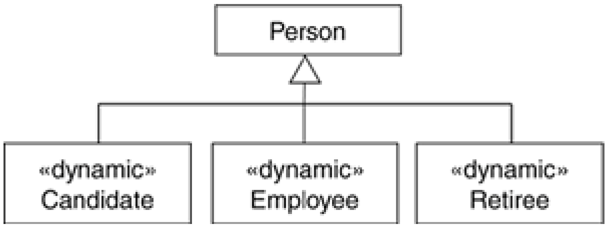
- For each such collaboration in your system, identify the interfaces it requires from (imports) and those it provides to others (exports). You model the importing of interfaces by dependency relationships, and you model the exporting of interfaces by realization relationships.
- For each such interface in your system, document its dynamics by using pre- and postconditions for each operation, and use cases and state machines for the interface as a whole.



Modeling Static and Dynamic Types

o model a dynamic type,

- Specify the different possible types of that object by rendering each type as a class (if the abstraction requires structure and behavior) or as an interface (if the abstraction requires only behavior).
- Model all the roles the class of the object may take on at any point in time. You can mark them with the «dynamic» stereotype. (This is not a predefined UML stereotype, but one that you can add.)
- In an interaction diagram, properly render each instance of the dynamically typed class. Display the type of the instance in brackets below the object's name, just like a state. (We are using UML syntax in a novel way, but one that we feel is consistent with the intent of states.)



4.Packages

Terms and Concepts

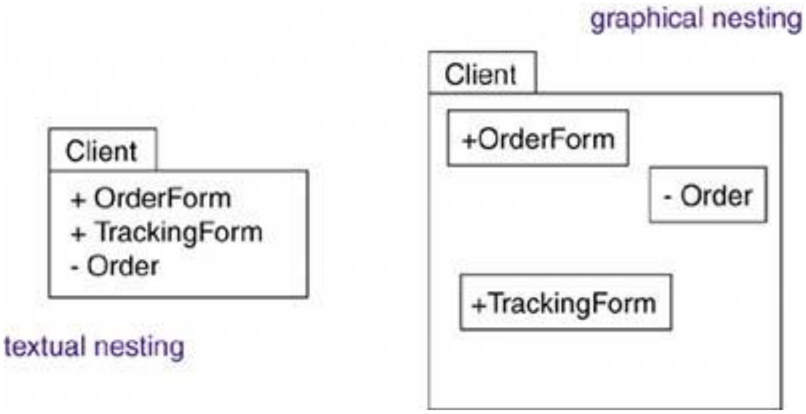
A *package* is a general-purpose mechanism for organizing the model itself into a hierarchy; it has no meaning to the execution. Graphically, a package is rendered as a tabbed folder. The name of the package goes in the folder (if its contents are not shown) or in the tab (if the contents of the folder are shown).

Names

Every package must have a name that distinguishes it from other packages. A *name* is a textual string. That name alone is known as a simple name; a qualified name is the package name prefixed by the name of the package in which that package lives, if any. A double colon (::) separates package names.

Owned Elements

A package may own other elements, including classes, interfaces, components, nodes, collaborations, use cases, diagrams, and even other packages. Ownership is a composite relationship, which means that the element is declared in the package. If the package is destroyed, the element is destroyed. Every element is uniquely owned by exactly one package.

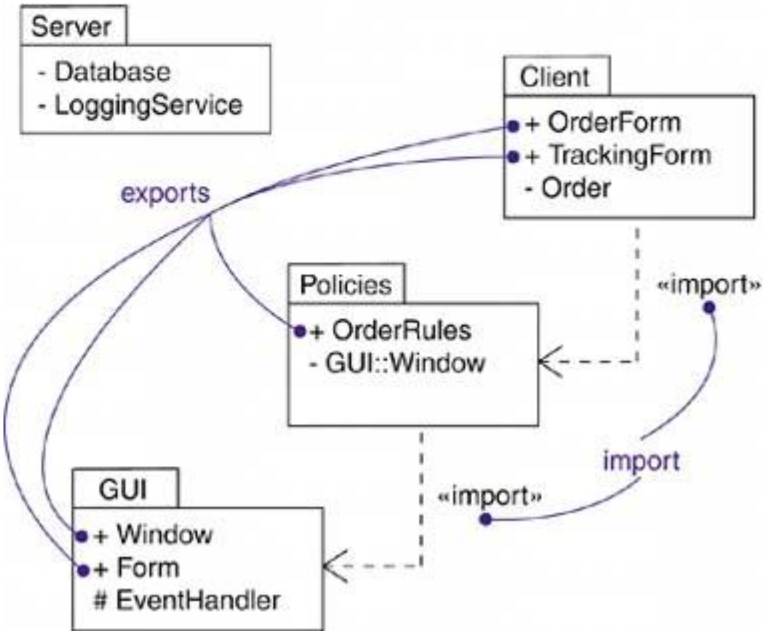


Visibility

You can control the visibility of the elements owned by a package just as you can control the visibility of the attributes and operations owned by a class. Typically, an element owned by a package is public, which means that it is visible to the contents of any package that imports the element's enclosing package. Conversely, protected elements can only be seen by children, and private elements cannot be seen outside the package in which they are declared.

Importing and Exporting

suppose that instead you put A in one package and B in another package, both packages sitting side by side. Suppose also that A and B are both declared as public parts of their respective packages. This is a very different situation. Although A and B are both public, accessing one of the classes from within the other package requires a qualified name. However, if A's package imports B's package, A can now see B directly, although still B cannot see A without a qualified name. Importing adds the public elements from the target package to the public namespace of the importing package. In the UML, you model an import relationship as a dependency adorned with the stereotype import. By packaging your abstractions into meaningful chunks and then controlling their access by importing, you can control the complexity of large numbers of abstractions.

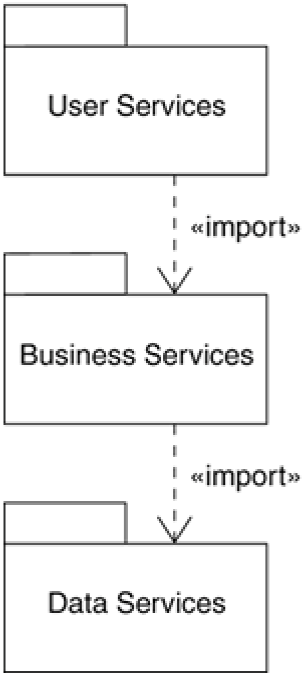


Common Modeling Techniques

Modeling Groups of Elements

To model groups of elements,

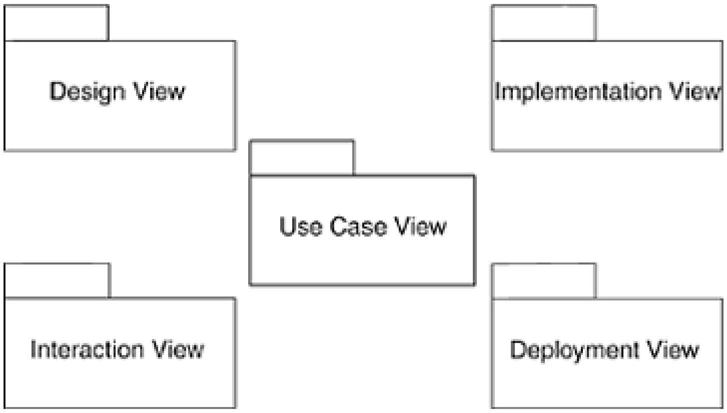
- Scan the modeling elements in a particular architectural view and look for clumps defined by elements that are conceptually or semantically close to one another.
- Surround each of these clumps in a package.
- For each package, distinguish which elements should be accessible outside the package. Mark them public, and all others protected or private. When in doubt, hide the element.
- Explicitly connect packages that build on others via import dependencies.
- In the case of families of packages, connect specialized packages to their more general part via generalizations.



Modeling Architectural Views

To model architectural views,

- Identify the set of architectural views that are significant in the context of your problem. In practice, this typically includes a design view, an interaction view, an implementation view, a deployment view, and a use case view.
- Place the elements (and diagrams) that are necessary and sufficient to visualize, specify, construct, and document the semantics of each view into the appropriate package.
- As necessary, further group these elements into their own packages.


Design View
Implementation View
Use Case View
Interaction View
Deployment View