


Guidelines: Analysis Class

 <p>Analysis Class</p>	<p>Analysis classes represent an early conceptual model for 'things in the system, which have responsibilities and behavior'. They eventually evolve into classes and subsystems in the Design Model.</p>
--	---

Topics

- [Analysis Class Stereotypes](#)
 - [Boundary Class](#)
 - [Control Class](#)
 - [Entity Class](#)
- [Association Stereotype Usage Restrictions](#)
 - [Restrictions for Boundary Classes](#)
 - [Restrictions for Control Classes](#)
 - [Restrictions for Entity Classes](#)
 - [Summary of Restrictions](#)
- [Enforcing Consistency](#)

Analysis Class Stereotypes



Analysis classes may be stereotyped as one of the following:

- Boundary classes
- Control classes
- Entity classes

Apart from giving you more specific process guidance when finding classes, this stereotyping results in a robust object model because changes to the model tend to affect only a specific area. Changes in the user interface, for example, will affect only boundary classes. Changes in the control flow will affect only control classes. Changes in long-term information will affect only entity classes. However, these stereotypes are specially useful in helping you to identify classes in analysis and early design. You should probably consider using a slightly different set of stereotypes in later phases of design to better correlate to the implementation environment, the application type, and so on.

Boundary Class



A **boundary class** is a class used to model interaction between the system's surroundings and its inner workings. Such interaction involves transforming and translating events and noting changes in the system presentation (such as the interface). Boundary classes model the parts of the system that depend on its surroundings. Entity classes and control classes model the parts that are independent of the system's surroundings. Thus, changing the GUI or communication protocol should mean changing only the boundary classes, not the entity and control classes. Boundary classes also make it easier to understand the system because they clarify the system's boundaries. They aid design by providing a good point of departure for identifying related services. For example, if you identify a printer interface early in the design, you will soon see that you must also model the formatting of printouts.

Guidelines

Common boundary classes include windows, communication protocols, printer interfaces, sensors, and terminals. You do not have to model routine interface parts, such as buttons, as separate boundary classes if you are using a GUI builder. Generally the entire window is the finest grained boundary object. Boundary classes are also useful for capturing interfaces to possibly nonobject oriented API's, such as legacy code.

You should model boundary classes according to what kind of boundary they represent. Communication with another system and communication with a human actor (through a user interface) have very different objectives. During user-interface modeling, the most important concern is how the interface will be presented to the user. During system-communication modeling, the most important concern is the communication protocol.

A boundary object (an instance of a boundary class) can outlive a use case instance if, for example, it must appear on a screen between the performance of two use cases. Normally, however, boundary objects live only as long as the use case instance.

Finding Boundary Classes

A boundary class intermediates the interface to something outside the system. Boundary objects insulate the system from changes in the surroundings (changes in interfaces to other systems, changes in user requirements, etc.), keeping these changes from affecting the rest of the system. A system may have several types of boundary classes:

- **User interface classes** - classes which intermediate communication with human users of the system
- **System interface classes** - classes which intermediate communication with other system
- **Device interface classes** - classes which provide the interface to devices (such as sensors), which detect external events

Find User-Interface Classes

Boundary classes representing the user interface may exist from user-interface modeling activities; where appropriate, re-use these classes in this activity. In the event that user-interface modeling has not been done, the following discussion will aid in finding these classes.

There is at least one boundary object for each use-case actor-pair. This object can be viewed as having responsibility for coordinating the interaction with the actor. This boundary object may have **subsidiary** objects to which it delegates some of its responsibilities. This is particularly true for window-based GUI applications, where there is typically one boundary object for each window, or one for each form.

Make sketches, or use screen dumps from a user-interface prototype, that illustrate the behavior and appearance of the boundary objects.

Only model the key abstractions of the system; do not model every button, list and widget in the GUI. The goal of analysis is to form a good picture of how the system is composed, not to design every last detail. In other words, identify boundary classes only for phenomena in the system or for things mentioned in the **flow of events** of the use-case realization. See also [Guidelines: Boundary Class \(Modeling the User Interface\)](#).

Find System-Interface Classes

A boundary class which communicates with an external system is responsible for managing the dialogue with the external system; it provides the interface to that system for the system being built.

Example

In an Automated Teller Machine, withdrawal of funds must be verified through the ATM Network, an actor (which in turn verifies the withdrawal with the bank

Guidelines

accounting system). An object called ATM Network Interface can be identified to provide communication with the ATM Network.

The interface to an existing system may already be well-defined; if it is, the responsibilities should be derived directly from the interface definition. If a formal interface definition exists, it may be reverse engineered and we need not formally define it here; simply make note of the fact that the existing interface will be reused during design.

Find Device Interface Classes

The system may contain elements that act as if they were external (change value spontaneously without any object in the system affecting them), such as sensor equipment. Although it is possible to represent this type of external device using actors, users of the system may find doing so "confusing", as it tends to put devices and human actors on the same "level". Once we move away from gathering requirements, however, we need to consider the source for all external events and make sure we have a way for the system to detect these events.

If the device is represented as an actor in the use-case model, it is easy to justify using a boundary class to intermediate communication between the device and the system. If the use-case model does not include these "device-actors", now is the appropriate time to add them, updating the Supplementary Descriptions of the Use Cases where appropriate.

For each "device-actor", create a boundary class to capture the responsibilities of the device or sensor. If there is a well-defined interface already existing for the device, make note of it for later reference during design.

Control Class



A **control class** is a class used to model control behavior specific to one or a few use cases. Control objects (instances of control classes) often control other objects, so their behavior is of the coordinating type. Control classes encapsulate use-case specific behavior.

The behavior of a control object is closely related to the realization of a specific use case. In many scenarios, you might even say that the control objects "run" the use-case realizations. However, some control objects can participate in more than one use-case realization if the use-case tasks are strongly related. Furthermore, several control objects of different control classes can participate in one use case. Not all use cases require a control object. For example, if the flow of events in a use case is related to one entity object, a boundary object may realize the use case in cooperation with the entity object. You can start by identifying one control class per use-case realization, and then refine this as more use-case realizations are identified and commonality is discovered.

Control classes can contribute to understanding the system because they represent the dynamics of the system, handling the main tasks and control flows.

When the system performs the use case, a control object is created. Control objects usually die when their corresponding use case has been performed.

Note that a control class does not handle **everything** required in a use case. Instead, it coordinates the activities of other objects that implement the functionality. The control class delegates work to the objects that have been assigned the responsibility for the functionality.

Finding Control Classes

Control classes provide coordinating behavior in the system. The system can perform some use cases without control objects (just using entity and boundary objects)—particularly use cases that involve only the simple manipulation of stored information.

More complex use cases generally require one or more control classes to coordinate the behavior of other objects in the system. Examples of control objects include programs such as transaction managers, resource coordinators, and error handlers.

Guidelines

Control classes effectively de-couple boundary and entity objects from one another, making the system more tolerant of changes in the system boundary. They also de-couple the use-case specific behavior from the entity objects, making them more reusable across use cases and systems.

Control classes provide behavior that:

- Is surroundings-independent (does not change when the surroundings change),
- Defines control logic (order between events) and transactions within a use case.
- Changes little if the internal structure or behavior of the entity classes changes,
- Uses or sets the contents of several entity classes, and therefore needs to coordinate the behavior of these entity classes.
- Is not performed in the same way every time it is activated (flow of events features several states).

Determine whether a Control Class is Needed

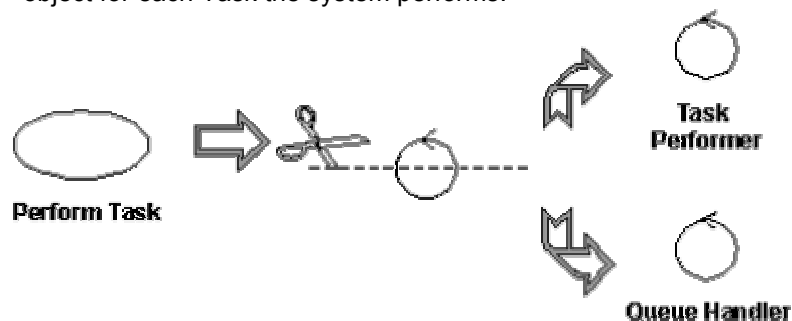
The flow of events of a use case defines the order in which different tasks are performed. Start by investigating if the flow can be handled by the already identified boundary and entity classes. For simple **flows of events** which primarily enter, retrieve and display, or modify information, a separate control class is not usually justified; the boundary classes will be responsible for coordinating the use case.

The **flows of events** should be encapsulated in a separate control class when it is complex and consists of dynamic behavior that may change independently from the interfaces (boundary classes) or information stores (entity classes) of the system. By encapsulating the **flows of events**, the same control class can potentially be re-used for a variety of systems which may have different interfaces and different information stores (or at least the underlying data structures).

Example: Managing a Queue of Tasks

You can identify a control class from the use case Perform Task in the Depot-Handling System. This control class handles a queue of Tasks, ensuring that Tasks are performed in the right order. It performs the next Task in the queue as soon as suitable transportation equipment is allocated. The system can therefore perform several Tasks at the same time.

The behavior defined by the corresponding control object is easier to describe if you split it into two control classes, Task Performer and Queue Handler. A Queue Handler object will handle only the queue order and the allocation of transportation equipment. One Queue Handler object is needed for the whole queue. As soon as the system performs a Task, it will create a new Task Performer object, which will perform the Task. We thus need one Task Performer object for each Task the system performs.



Complex classes should be divided along lines of similar responsibilities

The principal benefit of this split is that we have separated queue handling responsibilities (something generic to many use cases) from the specific activities

Guidelines

of task management, which are specific to this use case. This makes the classes easier to understand and easier to adapt as the design matures. It also has benefits in balancing the load of the system, as many Task Performers can be created as necessary to handle the workload.

Encapsulating the Main Flow of Events and Alternate/Exceptional Flows of Events in separate Control Classes

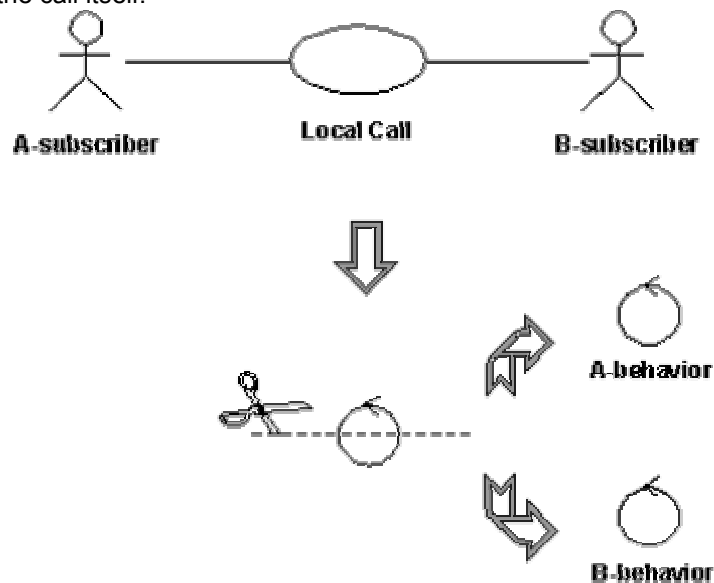
To simplify changes, encapsulate the main flow of events and alternate flows of events in different control classes. If alternate and exception flows are completely independent, separate them as well. This will make the system easier to extend and maintain over time.

Divide Control Classes where two Actors share the same Control Class

Control classes may also need to be divided when several actors use the same control class. By doing this, we isolate changes in the requirements of one actor from the rest of the system. In cases where the cost of change is high or the consequences dire, you should identify all control classes which are related to more than one actor and divide them. In the ideal case, each control class should interact (via some boundary object) with one actor or none at all.

Example: Call Management

Consider the use case **Local Call**. Initially, we can identify a control class to manage the call itself.



The control class handling local phone calls in a telephone system can quickly be divided into two control classes, **A-behavior** and **B-behavior**, one for each actor involved.

In a local phone call, there are two actors: **A-subscriber** who initiates the call, and **B-subscriber** who receives the call. The **A-subscriber** lifts the receiver, hears the dial tone, and then dials a number of digits, which the system stores and analyzes. When the system has received all the digits, it sends a ringing tone to **A-subscriber**, and a ringing signal to **B-subscriber**. When **B-subscriber** answers, the tone and the signal stop, and the conversation between the subscribers can begin. The call is finished when both subscribers hang up.

Two behaviors must be controlled: What happens at A-subscriber's place and what happens at B-subscriber's place. For this reason, the original control object was split into two control objects, **A-behavior** and **B-behavior**.

You do not have to divide a control class if:

Summit

Guidelines

- You can be reasonably sure that the behavior of the actors related to the objects of the control class will never change, or change very little.
- The behavior of an object of the control class toward one actor is very insignificant compared with its behavior toward another actor, a single object can hold all the behavior. Combining behavior in this way will have a negligible effect on changeability.

Entity Class

An **entity class** is a class used to model information and associated behavior that must be stored. Entity objects (instances of entity classes) are used to hold and update information about some phenomenon, such as an event, a person, or some real-life object. They are usually persistent, having attributes and relationships needed for a long period, sometimes for the life of the system.

An entity object is usually not specific to one use-case realization; sometimes, an entity object is not even specific to the system itself. The values of its attributes and relationships are often given by an actor. An entity object may also be needed to help perform internal system tasks. Entity objects can have behavior as complicated as that of other object stereotypes. However, unlike other objects, this behavior is strongly related to the phenomenon the entity object represents. Entity objects are independent of the environment (the actors).

Entity objects represent the key concepts of the system being developed. Typical examples of entity classes in a banking system are **Account** and **Customer**. In a network-handling system, examples are **Node** and **Link**.

If the phenomenon you wish to model is not used by any other class, you can model it as an attribute of an entity class, or even as a relationship between entity classes. On the other hand, if the phenomenon is used by any other class in the design model, you must model it as a class. Entity classes provide another point of view from which to understand the system because they show the logical data structure, which can help you understand what the system is supposed to offer its users.

Finding Entity Classes

Entity classes represent stores of information in the system; they are typically used to represent the key concepts the system manages. Entity objects are frequently passive and persistent. Their main responsibilities are to store and manage information in the system.

A frequent source of inspiration for entity classes are the Glossary (developed during requirements) and a business-domain model (developed during business modeling, if business modeling has been performed).

Association Stereotype Usage Restrictions

Restrictions for Boundary Classes

The following are allowable:

- Communicate associations between two Boundary classes, for instance, to describe how a specific window is related to other boundary objects.
- Communicate or subscribe associations from a Boundary class to an Entity class, because boundary objects might need to keep track of certain entity objects between actions in the boundary object, or be informed of state changes in the entity object.
- Communicate associations from a Boundary class to a Control class, so that a boundary object may trigger particular behavior.

Restrictions for Control Classes

The following are allowable:

- Communicate or subscribe associations between Control classes and Entity classes, because control objects might need to keep track of certain entity objects between actions in the control object, or be informed of state changes in the entity object.
- Communicate associations between Control and Boundary classes, allowing the results of invoked behavior to be communicated to the environment.
- Communicate associations between Control classes, allowing the construction of more complex behavioral patterns.

Restrictions for Entity Classes

Entity classes should only be the source of associations (communicate or subscribe) to other entity classes. Entity class objects tend to be long-lived; control and boundary class objects tend to be short-lived. It is sensible from an architectural viewpoint to limit the visibility that an entity object has of its surroundings, that way, the system is more amenable to change.

Summary of Restrictions


From\To (navigability)	Boundary	Entity	Control
Boundary	communicate	communicate subscribe	communicate
Entity		communicate subscribe	
Control	communicate	communicate subscribe	communicate

Valid Association Stereotype Combinations

Enforcing Consistency

- When a new behavior is identified, check to see if there is an existing class that has similar responsibilities, reusing classes where possible. Only when sure that there is not an existing object that can perform the behavior should you create new classes.
- As classes are identified, examine them to ensure they have consistent responsibilities. When an classes responsibilities are disjoint, split the object into two or more classes. Update the collaboration diagrams accordingly.
- If the a class is split because disjoint responsibilities are discovered, examine the collaborations in which the class plays a role to see if the collaboration needs to be updated. Update the collaboration if needed.
- A class with only one responsibility is not a problem, per se, but it should raise questions on why it is needed. Be prepared to challenge and justify the existence of all classes.

Guidelines: Design Class

 Design Class	A design class is a description of a set of objects that share the same responsibilities, relationships, operations, attributes, and semantics.
--	--

Topics

- [Definition](#)
- [Operations](#)
 - [Parameters](#)
 - [Class Operations](#)
 - [Operation Visibility](#)
- [States](#)
- [Collaborations](#)
- [Attributes](#)
 - [Class Attributes](#)
 - [Modeling External Units with Attributes](#)
 - [Attribute Visibility](#)

Definition

A **design class** represents an abstraction of one or several classes in the system's implementation; exactly what it corresponds to depends on the implementation language. For example, in an object-oriented language such as C++, a class can correspond to a plain class. Or in Ada, a class can correspond to a tagged type defined in the visible part of a package. Classes define **objects**, which in turn realize (implement) the use cases. A class originates from the requirements the use-case realizations make on the objects needed in the system, as well as from any previously developed object model.

Whether or not a class is good depends heavily on the implementation environment. The proper size of the class and its objects depends on the programming language, for example. What is considered right when using Ada might be wrong when using Smalltalk. Classes should map to a particular phenomenon in the implementation language, and the classes should be structured so that the mapping results in good code.

Even though the peculiarities of the implementation language influence the design model, you must keep the class structure easy to understand and modify. You should design as if you had classes and encapsulation even if the implementation language does not support this.

Operations

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. What behavior you assign to an object depends on what role it has in the use-case realizations.

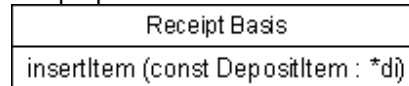
Parameters

Guidelines

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.

Example:

In the **Recycling Machine System**, the objects of a **Receipt Basis** class keep track of how many deposit items of a certain type a customer has handed in. The behavior of a **Receipt Basis** object includes incrementing the number of objects returned. The operation **insertItem**, which receives a reference to the item handed in, fills this purpose.



Use the implementation language syntax and semantics when specifying operations.

Class Operations

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 type-scoping the operation.

Operation Visibility

The following visibilities are possible on an operation:

- **Public**: the operation is visible to model elements other than the class itself.
- **Protected**: the operation is visible only to the class itself, to its subclasses, or to **friends** of the class (language dependent)
- **Private**: the operation is only visible to the class itself and to **friends** of the class
- **Implementation**: the operation is visible only within to the class itself.

Public visibility should be used **very sparingly**, only when an operation is needed by another class.

Protected visibility should be the **default**; it protects the operation from use by external classes, which promotes loose coupling and encapsulation of behavior.

Private visibility should be used in cases where you want to prevent **subclasses** from inheriting the operation. This provides a way to de-couple subclasses from the super-class and to reduce the need to remove or exclude unused inherited operations.

Implementation visibility is the **most restrictive**; it is used in cases where only the class itself is able to use the operation. It is a **variant of Private visibility**, which for most cases is suitable.

States

An object can react differently to a specific message depending on what state it is in; the state-dependent behavior of an object is defined by an associated statechart diagram. For each state the object can enter, the statechart diagram describes what messages it can receive, what operations will be carried out, and what state the object will be in thereafter. Refer to [Guidelines: Statechart Diagram](#) for more information.

Collaborations

Guidelines

A collaboration is a dynamic set of object interactions in which a set of objects communicate by sending **messages** to each other. Sending a message is straightforward in Smalltalk; in Ada it is done as a subprogram call. A message is sent to a receiving object that invokes an operation within the object. The message indicates the name of the operation to perform, along with the required parameters. When messages are sent, **actual parameters** (values for the formal **parameters**) are supplied for all the **parameters**.

The message transmissions among objects in a use-case realization and the focus of control the objects follow as the operations are invoked are described in interaction diagrams. See [Guidelines: Sequence Diagram](#) and [Guidelines: Collaboration Diagram](#) for information about these diagrams.

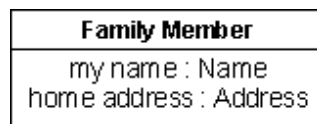
Attributes

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 have an initial value when the object is created.

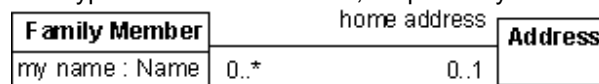
You should model attributes only if doing so makes an object more understandable. You should model the property of an object as an attribute only if it is a property of **that object alone**.

Otherwise, you should model the property with an association or aggregation relationship to a class whose objects represent the property.

Example:



An example of how an attribute is modeled. Each member of a family has a name and an address. Here, we have identified the attributes **my name** and **home address** of type **Name** and **Address**, respectively:



In this example, an association is used instead of an attribute. The **my name** property is probably unique to each member of a family. Therefore we can model it as an attribute of the attribute type **Name**. An address, though, is shared by all family members, so it is best modeled by an association between the **Family Member** class and the **Address** class.

It is not always easy to decide immediately whether to model some concept as a separate object or as an attribute of another object. Having unnecessary objects in the object model leads to unnecessary documentation and development overhead. You must therefore establish certain criteria to determine how important a concept is to the system.

- **Accessibility.** What governs your choice of object versus attribute is not the importance of the concept in real life, but the need to access it during the use case. If the unit is accessed frequently, model it as an object.
- **Separateness during execution.** Model concepts handled separately during the execution of use cases as objects.
- **Ties to other concepts.** Model concepts strictly tied to certain other concepts and never used separately, but always via an object, as an attribute of the object.
- **Demands from relationships.** If, for some reason, you must relate a unit from two directions, re-examine the unit to see if it should be a separate object. Two objects cannot associate the same instance of an attribute type.
- **Frequency of occurrence.** If a unit exists only during a use case, do not model it as an object. Instead model it as an attribute to the object that performs the behavior in question, or simply mention it in the description of the affected object.

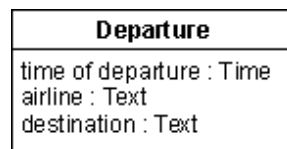
Guidelines

- **Complexity.** If an object becomes too complicated because of its attributes, you may be able to extract some of the attributes into separate objects. Do this in moderation, however, so that you do not have too many objects. On the other hand, the units may be very straightforward. For example, classified as attributes are (1) units that are simple enough to be supported directly by primitive types in the implementation language, such as, integers in C++, and (2) units that are simple enough to be implemented by using the application-independent components of the implementation environment, such as, **String** in C++ and Smalltalk-80.

You will probably model a concept differently for different systems. In one system, the concept may be so vital that you will model it as an object. In another, it may be of minor importance, and you will model it as an attribute of an object.

Example:

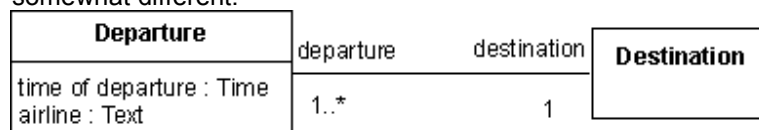
For example, for an airline company you would develop a system that supports departures.



A system that supports departures. Suppose the personnel at an airport want a system that supports departures. For each departure, you must define the time of departure, the airline, and the destination.

You can model this as an object of a class **Departure**, with the attributes **time of departure**, **airline**, and **destination**.

If, instead, the system is developed for a travel agency, the situation might be somewhat different.



Flight destinations forms its own object, **Destination**.

The time of departure, airline, and destination will, of course, still be needed. Yet there are other requirements, because a travel agency is interested in finding a departure with a specific destination. You must therefore create a separate object for **Destination**. The objects of **Departure** and **Destination** must, of course, be aware of each other, which is enabled by an association between their classes.

The argument for the importance of certain concepts is also valid for determining what attributes should be defined in a class. The class **Car** will no doubt define different attributes if its objects are part of a motor-vehicle registration system than if its objects are part of an automobile manufacturing system.

Finally, the rules for what to represent as objects and what to represent as attributes are not absolute. Theoretically, you can model everything as objects, but this is cumbersome. A simple rule of thumb is to view an object as something that at some stage is used irrespective of other objects. In addition, you do not have to model every object property using an attribute, only properties necessary to understand the object. You should not model details that are so implementation-specific that they are better handled by the implementer.

Class Attributes

An attribute nearly always denotes object properties. An attribute can also denote properties of a class, in which case it is a **class attribute**. This can be modeled in the UML by type-scoping the attribute.

Modeling External Units with Attributes

An object can encapsulate something whose value can change without the object performing any behavior. It might be something that is really an external unit, but that was not modeled as an actor. For example, system boundaries may have been chosen so that some form of sensor equipment lies within them. The sensor can then be encapsulated within an object, so that the value it measures constitutes an attribute. This value can then change continually, or at certain intervals without the object being influenced by any other object in the system.

Example:

You can model a thermometer as an object; the object has an attribute that represents temperature, and changes value in response to changes in the temperature of the environment. Other objects may ask for the current temperature by performing an operation on the thermometer object.

Thermometer
temperature : Kelvin

The value of the attribute **temperature** changes spontaneously in the **Thermometer** object.

You can still model an encapsulated value that changes in this way as an ordinary attribute, but you should describe in the object's class that it changes **spontaneously**.

Attribute Visibility

Attribute visibility assumes one of the following values:

- **Public:** the attribute is visible both inside and outside the package containing the class.
- **Protected:** the attribute is visible only to the class itself, to its subclasses, or to **friends** of the class (language dependent)
- **Private:** the attribute is only visible to the class itself and to **friends** of the class
- **Implementation:** the attribute is visible to the class itself.

Public visibility should be used **very sparingly**, only when an attribute is directly accessible by another class. Defining public visibility is effectively a short-hand notation for defining the attribute visibility as protected, private or implementation, with associated public operations to get and set the attribute value. Public attribute visibility can be used as a declaration to a code generator that these get/set operations should be automatically generated, saving time during class definition.

Protected visibility should be the **default**; it protects the attribute from use by external classes, which promotes loose coupling and encapsulation of behavior.

Private visibility should be used in cases where you want to prevent **subclasses** from inheriting the attribute. This provides a way to de-couple subclasses from the super-class and to reduce the need to remove or exclude unused inherited attributes.

Implementation visibility is the **most restrictive**; it is used in cases where only the class itself is able to use the attribute. It is a **variant of Private visibility**, which for most cases is suitable.

Topics

- [References](#)
- [Architectural Goals and Constraints](#)

Guidelines

- [The Use-Case View](#)
- [The Logical View](#)
- [The Process View](#)
- [The Deployment View](#)
- [The Implementation View](#)
- [The Data View](#)
- [Size and Performance](#)
- [Quality](#)

References



The references section presents external documents which provide background information important to an understanding of the architecture of the system. If there are a larger number of references, structure the section in subsections:

1. external documents
2. internal documents
3. government documents
4. non-government documents
5. etc.

Architectural Goals and Constraints



The architecture will be formed by considering:

- functional requirements, captured in the Use-Case Model, and
- non-functional requirements, captured in the Supplementary Specifications

However, these are not the only influences that will shape the architecture: there will be constraints imposed by the environment in which the software must operate; by the need to reuse existing assets; by the imposition of various standards; by the need for compatibility with existing systems, and so on. There may also be a preexisting set of **architectural principles and policies** which will guide the development, and which need to be elaborated and reified for the project. This section of the Software Architecture document is the place to describe these goals and constraints, and any **architectural decisions** flowing from them which do not find a ready home (as requirements) elsewhere. The enforcement of these decisions is achieved by framing a set of **architecture evaluation criteria** which will be used as part of the iteration assessment. Evaluation criteria are also derived from **Change Cases** which document likely future changes to:

- the system's capabilities and properties
- the way the system is used
- the system's operating and support environments

Change Cases clarify those properties of the system described by subjective phrases such as, "easy to extend", "easy to port", "easy to maintain", "robust in the face of change", and "quick to develop". Change Cases focus on what is important and likely rather than just what is possible. Change Cases try to predict changes: such predictions rarely turn out to be exactly true. The properties of a system are determined by users, sponsors, suppliers, developers, and other stakeholders. Changes can arise from many sources, for example:

- Business drivers: new and modified business processes and goals

Guidelines

- Technology drivers: adaptation of the system to new platforms, integration with new components
- Changes in the profiles of the average user
- Changes in the integration needs with other systems
- Scope changes arising from the migration of functionality from external systems

The Use-Case View

The Use-Case View presents a subset of the [Artifact: Use-Case Model](#), presenting the architecturally significant use-cases of the system. It describes the set of scenarios and/or use cases that represent some significant, central functionality. It also describes the set of scenarios and/or use cases that have a substantial architectural coverage (that exercise many architectural elements) or that stress or illustrate a specific, delicate point of the architecture. If the model is larger, it will typically be organized in packages; for ease of understanding the use-case view should similarly be organized by package, if they are packaged. For each significant use case, include a subsection with the following information:

1. The name of the use case.
2. A brief description of the use case.
3. Significant descriptions of the **Flow of Events** of the use case. This can be the whole **Flow of Events** description, or subsections of it that describe significant flows or scenarios of the use case.
4. Significant descriptions of relationships involving the use case, such as include- and extend-relationships, or communicates-associations.
5. An enumeration of the significant use-case diagrams related to the use case.
6. Significant descriptions of **Special Requirements** of the use case. This can be the whole **Special Requirements** description, or subsections of it that describe significant requirements.
7. Significant **Pictures of the User Interface**, clarifying the use case.
8. The realizations of these use cases should be found in the logical view.

The Logical View

The Logical View is a subset of the [Artifact: Design Model](#) which presents architecturally significant design elements. It describes the most important classes, their organization in packages and subsystems, and the organization of these packages and subsystems into layers. It also describes the most important use-case realizations, for example, the dynamic aspects of the architecture.

A complex system may require a number of sections to describe the Logical View:

1. Overview

This subsection describes the overall decomposition of the design model in terms of its package hierarchy and layers. If the system has several levels of packages, you should first describe those that are significant at the top level. Include any diagrams showing significant top-level packages, as well as their interdependencies and layering. Next present any significant packages within these, and so on all the way down to the significant packages at the bottom of the hierarchy.

2. Architecturally Significant Design Packages

For each significant package, include a subsection with the following information

Guidelines

1. Its name.
2. A brief description.
3. A diagram with all significant classes and packages contained within the package. For a better understanding this diagram may show some classes from other packages if necessary.
4. For each significant class in the package, include its name, brief description, and, optionally a description of some of its major responsibilities, operations and attributes. Also describe its important relationships if necessary to understand the included diagrams.

3. Use-Case Realizations

This section illustrates how the software works by giving a few selected use-case (or scenario) realizations, and explains how the various design model elements contribute to their functionality. The realizations given here are chosen because they represent some significant, central functionality of the final system; or for their architectural coverage - they exercise many architectural elements - or stress or illustrate a specific, delicate point of the architecture. The corresponding use cases and scenarios of these realizations should be found in the use-case view.

For each significant use-case realization, include a subsection with the following information

1. The name of the realized use case.
2. A brief description of the realized use case.
3. Significant descriptions of the **Flow of Events - Design** of the use-case realization. This can be the whole **Flow of Events - Design** description, or subsections of it that describe the realization of significant flows or scenarios of the use case.
4. An enumeration of the significant interaction or class diagrams related to the use-case realization.
5. Significant descriptions of **Derived Requirements** of the use-case realization. This can be the whole **Derived Requirements** description, or subsections of it that describe significant requirements.

Architecturally Significant Design Elements

To assist in deciding what is architecturally significant, some examples of qualifying elements and their characteristics are presented:

- A model element that encapsulates a major abstraction of the problem domain, such as:
 - A flight plan in an air-traffic control system.
 - An employee in a payroll system.
 - A subscriber in a telephone system.

Sub-types of these should not necessarily be included, e.g. Distinguishing an **ICAO Standard Flight Plan** from a **US Domestic Flight Plan** is not important; they are all flight plans and share a substantial amount of attributes and operations.

Distinguishing a subscriber with a data line, or with a voice line, does not matter as long as the call handling proceeds in roughly the same way.

- A model element that is used by many other model elements.
- A model element that encapsulates a major mechanism (service) of the system
- Design Mechanisms
 - Persistency mechanism (repository, database, memory management).

Guidelines

- Communication mechanism (RPC, broadcast, broker service).
- Error handling or recovery mechanism.
- Display mechanism, and other common interfaces (windowing, data capture, signal conditioning, and so on).
- Parameterization mechanisms.

In general, any mechanism likely to be used in many different packages (as opposed to completely internal to a package), and for which it is wise to have one single common implementation throughout the system, or at least a single interface that hides several alternative implementations.

- A model element that participates in a major interface in the system with, for example:
 - An operating system.
 - An off-the-shelf product (windowing system, RDBMS, geographic information system).
 - A class that implements or supports an architectural pattern (such as patterns for de-coupling model elements, including the model-view-controller pattern, or the broker pattern).
- A model element that is of localized visibility, but may have some huge impact on the overall performance of the system, for example:
 - A polling mechanism to scan sensors at a very high rate.
 - A tracing mechanism for troubleshooting.
 - A check-pointing mechanism for high-availability system (check-point and restart).
 - A start-up sequence.
 - An online update of code.
 - A class that encapsulates a novel and technically risky algorithm, or some algorithm that is safety-critical or security-critical, for example: computation of irradiation level; airplane collision-avoidance criteria for congested airspace; Password encryption.

The criteria as to what is architecturally significant will evolve in the early iterations of the project, as you discover technical difficulties and begin to better understand the system. As a rule however, you should label at most 10% of the model elements as "**architecturally significant**." Otherwise you risk diluting the concept of architecture, and "everything is architecture." When you define and include the architecturally significant model elements in the logical view, you should also take the following aspects into consideration

- Identify potential for commonality and reuse. Which classes could be subclasses of a common class, or instances of the same parameterized class?
- Identify potential for parameterization. What part of the design can be made more reusable or flexible by using static and run-time parameters (such as table-driven behavior, or resource data loaded at start-up time)?
- Identify potential for using off-the-shelf products.

The Process View

The process view describes the process structure of the system. Since the process structure has great architectural impact, all processes should be presented. Within processes, only architecturally significant lightweight threads need be presented. The process view describes the tasks (processes and threads) involved in the system's execution, their interactions and configurations, as well as the allocation of objects and classes to tasks.

For each network of processes, include a subsection with the following information:

Guidelines

1. Its name.
2. The processes involved.
3. The interactions between processes in the form of collaboration diagrams, in which the objects are actual processes that encompass their own threads of control. For each process, briefly describe its behavior, lifetime and communication characteristics.

The Deployment View

This section describes one or more physical network (hardware) configurations on which the software is deployed and run. It also describes the allocation of tasks (from the **Process View**) to the physical nodes. For each physical network configuration, include a subsection with the following information:

1. Its name.
2. A deployment diagram illustrating the configuration, followed by a mapping of processes to each processor.
3. If there are many possible physical configurations, just describe a typical one and then explain the general mapping rules to follow in defining others. You should also include, in most cases, descriptions of network configurations for performing software tests and simulations.

This view is generated from the [Artifact: Deployment Model](#).

The Implementation View

This section describes the decomposition of the software into layers and subsystems in the implementation model. It describes an overview of the implementation model and its organization in terms of the components in implementation subsystems and layers, as well as the allocation of packages and classes (from the Logical View) to the implementation subsystems and components of the Implementation View. It contains two subsections:

1. Overview

This subsection names and defines the various layers and their contents, the rules that govern the inclusion to a given layer, and the boundaries between layers. Include a component diagram that shows the relations between layers.

2. Layers

For each layer, include a subsection with the following information:

1. Its name.
2. An enumeration of the subsystems located in the layer. For each subsystem, give its name, abbreviation or nickname, and a brief description.
3. A component diagram shows the subsystems and their import dependencies.
4. If appropriate, indicate its relationship to elements in the logical or process view.

The Data View

This view describes the architecturally significant persistent elements in the data model. It describes an overview of the data model and its organization in terms of the tables, views,

Guidelines

indexes, triggers and stored procedures used to provide persistence to the system. It also describes the mapping of persistent classes (from the Logical View) to the data structure of the database

It typically includes:

- The mapping from key persistent design classes, especially where the mapping is non-trivial.
- The architecturally significant parts of the system which have been implemented in the database, in the form of stored procedures and triggers.
- Important decisions in other views which have data implications, such as choice of transaction strategy, distribution, concurrency, fault tolerance. For example, the choice to use database-based transaction management (relying on the database to commit or abort transactions) requires that the error handling mechanism used in the architecture include a strategy for recovering from a failed transaction by refreshing the state of persistence objects cached in memory in the application.

You should present architecturally significant data model elements, describe their responsibilities, as well as a few very important relationships and behaviors (triggers, stored procedures, etc.).

Size and Performance

This section describes architecturally-defining volumetric and responsiveness characteristics of the system. The information presented may include:

- The number of key elements the system will have to handle (such as the number of concurrent flights for an air traffic control system, the number of concurrent phone calls for a telecom switch, the number of concurrent online users for an airline reservation system, etc.).
- The key performance measures of the system, such as average response time for key events; average, maximum and minimum throughput rates, etc.
- The footprint (in terms of disk and memory) of the executables - essential if the system is an embedded system which must live within extremely confining constraints.

Most of these qualities are captured as requirements; they are presented here because they shape the architecture in significant ways and warrant special focus. For each requirement, discuss how the architecture supports this requirement.


Quality

In this section, list the key quality dimensions of the system that shape the architecture. The information presented may include:

- Operating performance requirements, such as **mean-time between failure** (MTBF).
- Quality targets, such as "no unscheduled down-time"
- Extensibility targets, such as "the software will be upgradeable while the system is running".
- Portability targets, such as hardware platforms, operating systems, languages.

For each dimension, discuss how the architecture supports this requirement. You can organize the section by the different views (logical, implementation, and so on), or by quality. When particular characteristics are important in the system, for example, safety, security or privacy, the architectural support for these should be carefully delineated in this section

Guidelines: Association

 Association	<p>An association models a bi-directional semantic connection among instances.</p>
---	---

Topics

- [Associations](#)
- [Association Names](#)
- [Roles](#)
- [Multiplicity](#)
- [Navigability](#)
- [Self-Associations](#)
- [Multiple Associations](#)
- [Ordering Roles](#)
- [Links](#)
- [Association Classes](#)
- [Qualified Associations](#)
- [N-ary Associations](#)

Associations

Associations represent structural relationships between objects of different classes; they represent connections between instances of two or more classes that exist for some duration. Contrast this with transient links that, for example, exist only for the duration of an operation. These latter situations can instead be modeled using collaborations, in which the links exist only in particular limited contexts.

You can use associations to show that objects know about another objects. Sometimes, objects must hold references to each other to be able to interact, for example send messages to each other; thus, in some cases associations may follow from interaction patterns in sequence diagrams or collaboration diagrams.

Association Names

Most associations are binary (exist between exactly two classes), and are drawn as solid paths connecting pairs of class symbols. An association may have either a name or the association [roles](#) may have names. Role names are preferable, as they convey more information. In cases where only one of the roles can be named, roles are still preferable to association names so long as the association is expected to be uni-directional, starting from the object to which the role name is associated.

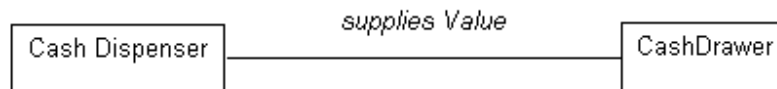
Associations are most often named during analysis, before sufficient information exists to properly name the roles. Where used, association names should reflect the purpose of the relationship and be a verb phrase. The name of the association is placed on, or adjacent to the association path.

Example

In an ATM, the **Cash Drawer** provides the money that the **Cash Dispenser** dispenses. In order for the **Cash Dispenser** to be able to dispense funds, it must

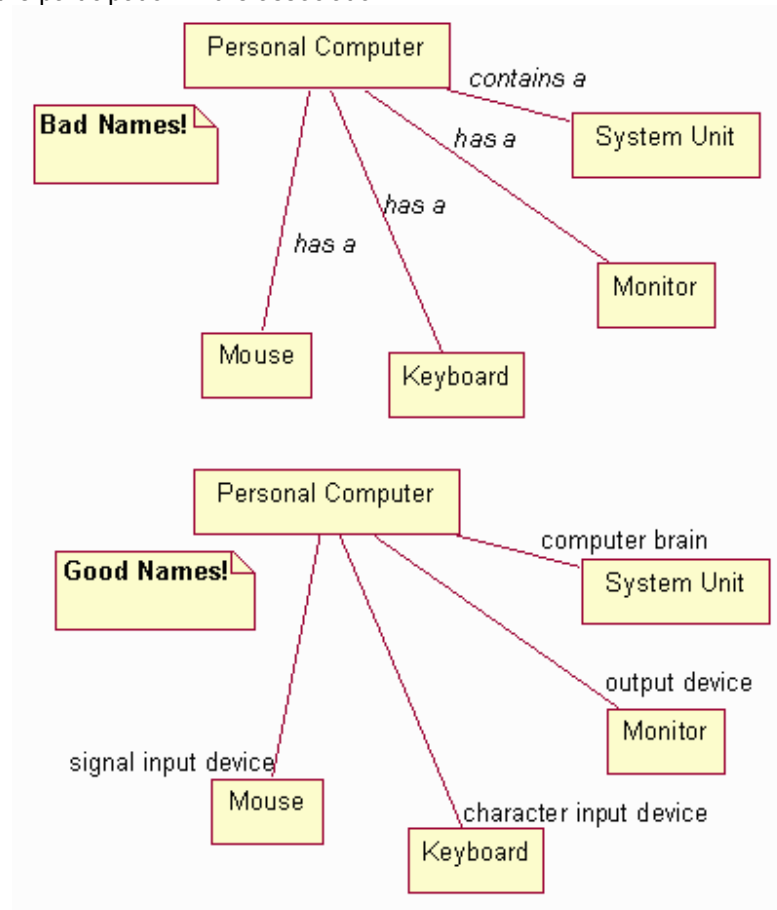
Guidelines

keep a reference to the **Cash Drawer** object; similarly, if the **Cash Drawer** runs out of funds, the **Cash Dispenser** object must be notified, so the **Cash Drawer** must keep a reference to the **Cash Dispenser**. An association models this reference.



An association between the **Cash Dispenser** and the **Cash Drawer**, named **supplies Value**.

Association names, if poorly chosen, can be confusing and misleading. The following example illustrates good and bad naming. In the first diagram, association names are used, and while they are syntactically correct (using verb phrases), they do not convey much information about the relationship. In the second diagram, role names are used, and these convey much more about the nature of the participation in the association.



Examples of good and bad usage of association and role names

Roles

Each end of an association is a **role** specifying the face that a class plays in the association. Each role must have a name, and the role names opposite a class must be unique. The role name should be a noun indicating the associated object's role in relation to the associating object.

Summit

Guidelines

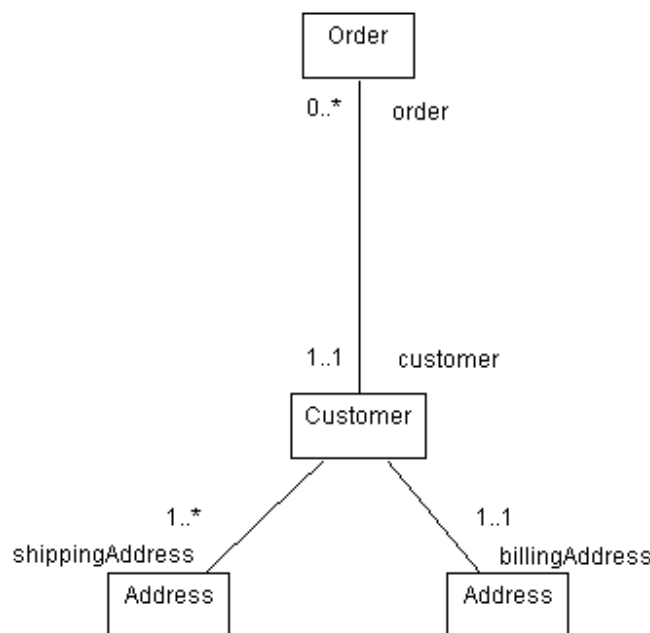
A suitable role name for a **Teacher** in an association with a **Course Section** would, for instance, be **lecturer**; avoid names like **"has"** and **"contains"**, as they add no information about what the relationships are between the classes.

Note that the use of association names and role names is mutually exclusive: one would not use both an association name **and** a role name. Role names are preferable to association names except in cases where insufficient information exists to name the role appropriately (as is often the case in analysis; in design role names should always be used). Lack of a good role name suggests an incomplete or ill-formed model.

The role name is placed next to the end of the association line.

Example

Consider the relationships between classes in an order entry system. A Customer can have two different kinds of Addresses: an address to which bills are sent, and a number of addresses to which orders may be sent. As a result, we have **two** associations between Customer and Address, as shown below. The associations are labeled with the **role** the associated address plays for the Customer.



Associations between **Customer**, **Address**, and **Order**, showing both role names and multiplicities

Multiplicity

For each role you can specify the **multiplicity** of its class, how many objects of the class can be associated with one object of the other class. Multiplicity is indicated by a text expression on the role. The expression is a comma-separated list of integer ranges. A range is indicated by an integer (the lower value), two dots, and an integer (the upper value); a single integer is a valid range, and the symbol **"**"** indicates "many", that is, an unlimited number of objects. The symbol **"**"** by itself is equivalent to **"0..*"**, that is, any number including none; this is the default value. An optional scalar role has the multiplicity **0..1**.

Example

In the previous example, multiplicities were shown for the associations between Order and Customer, and between Customer and Address. Interpreting the diagram, it says that an Order must have an associated Customer (the multiplicity is **1..1** at the Customer end), but a Customer may not have any Orders (the

Guidelines

multiplicity is 0..* at the Order end). Furthermore, a Customer has one billing address, but has one or more shipping address. To reduce notational clutter, if multiplicities are omitted, they may be assumed to be 1..1.

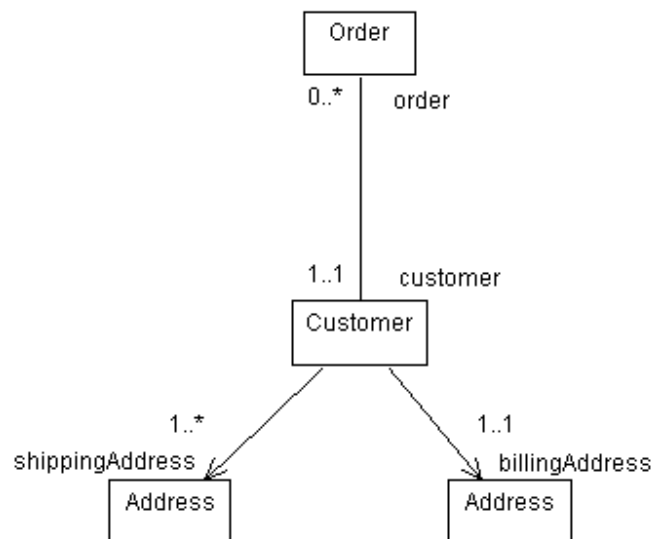
Navigability

The **navigability** property on a role 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**.

Example

In the order entry example, the association between the **Order** and the **Customer** is navigable in **both** directions: an **Order** must know which **Customer** placed the **Order**, and the **Customer** must know which **Orders** it has placed. When no arrowheads are shown, the association is assumed to be navigable in both directions.

In the case of the associations between **Customer** and **Address**, the **Customer** must know its **Addresses**, but the **Addresses** have no knowledge of which **Customers** (or other classes, since many things have addresses) are associated with the address. As a result, the navigability property of the **Customer** end of the association is turned off, resulting in the following diagram:



Updated Order Entry System classes, showing navigability of associations.

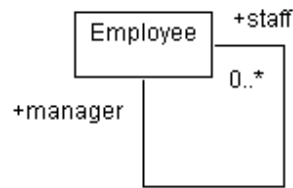
Self-Associations

Sometimes, a class has an association to itself. This does not necessarily mean that an instance of that class has an association to itself; more often, it means that one instance of the class has associations to other instances of the same class. In the case of self-associations, role names are essential to distinguish the purpose for the association.

Example

Consider the following self-association involving the class **Employee**:

Guidelines



In this case, an employee may have an association to other employees; if they do, they are a manager, and the other employees are members of their staff. The association is navigable in both directions since employees would know their manager, and a manager knows her staff.

Multiple Associations

Drawing two associations between classes means objects are related twice; a given object can be linked to different objects through each association. Each association is independent, and is distinguished by the role name. As shown above, a Customer can have associations to different instances of the same class, each with different role names.

Ordering Roles

When the multiplicity of an association is greater than one, the associated instances may be **ordered**. The **ordered** property on a role indicates that the instances participating in the association are ordered; by default they are an unordered set. The model does not specify **how** the ordering is maintained; the operations that update an ordered association must specify where the updated elements are inserted.

Links

The individual instances of an association are called **links**; a link is thus a relationship among instances. Messages may be sent on links, and links may denote references and aggregations between objects. See [Guidelines: Collaboration Diagram](#) for more information.

Association Classes

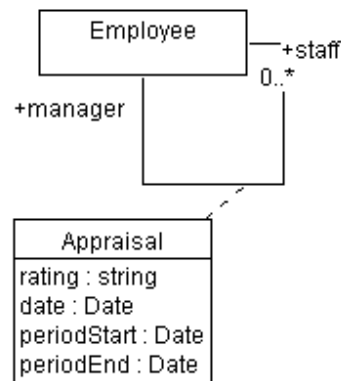
An **association class** is an association that also has class properties (such as attributes, operations, and associations). It is shown by drawing a dashed line from the association path to a class symbol that holds the attributes, operations, and associations for the association. The attributes, operations, 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.

Example

Expanding the Employee example from before, consider the case where an Employee (a staff-person) works for another Employee (a manager). The manager performs a periodic assessment of the staff member, reflecting their performance over a specific time period.

Guidelines

The appraisal cannot be an attribute of either the manager or the staff member alone, but we can associate the information with the association itself, as shown below:



The association class **Appraisal** captures information relating to the association itself

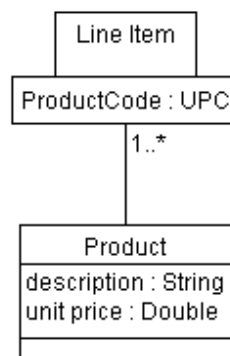
Qualified Associations



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 role; 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.

Example

Consider the following refinement of the association between **Line Item** and **Product**: a **Line Item** has an association to the **Product** which is ordered. Each **Line Item** refers to one and only one **Product**, while a **Product** may be ordered on many **Line Items**. By qualifying the association with the qualifier **ProductCode** we additionally indicate that each product has a unique product code, and that **Line Items** are associated with **Products** using this product code.



The association between **Line Item** and **Product** has the qualifier **ProductCode**.

N-ary Associations




Summit

Guidelines

An n-ary association is an association among three or more classes, where a single class can appear more than once. N-ary associations are drawn as large diamonds with one association path to each participating class. This is the traditional entity-relationship model symbol for an association. The binary form is drawn without the diamond for greater compactness, since they are the bulk of associations in a real model. N-ary associations are fairly rare and can also be modeled by promoting them to classes. N-ary associations can also have an association class; this is shown by drawing a dashed line from the diamond to the class symbol. Roles may have role names but multiplicity is more complicated and best specified by listing candidate keys. If given, the multiplicity represents the number of instances corresponding to a given tuple of the other N-1 objects. Most uses of n-ary associations can be eliminated using qualified associations or association classes. They can also be replaced by ordinary classes, although this loses the constraint that only one link can occur for a given tuple of participating objects.

Guidelines: Aggregation

	An aggregation is a special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.
---	--

Topics

- [Aggregation](#)
- [Shared Aggregation](#)
- [Composition](#)
- [Using Composition to Model Class Properties](#)
- [Aggregation or Association?](#)
- [Self-Aggregations](#)

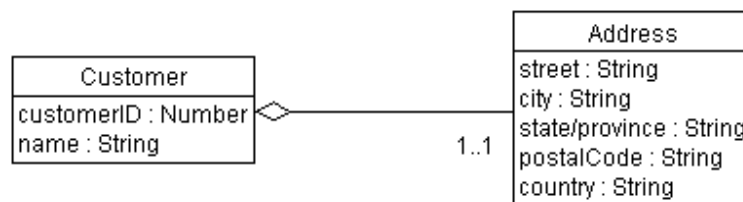
Aggregation

Aggregation is used to model a compositional relationship between model elements. There are many examples of compositional relationships: a **Library** contains **Books**, within a company **Departments** are made-up of **Employees**, a **Computer** is composed of a number of **Devices**. To model this, the aggregate (**Department**) has an **aggregation** association to the its constituent parts (**Employee**).

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

Example

In this example an **Customer** has an **Address**. We use aggregation because the two classes represent part of a larger whole. We have also chosen to model **Address** as a separate class, since many other kinds of things have addresses as well.



An aggregate object can hold other objects together.

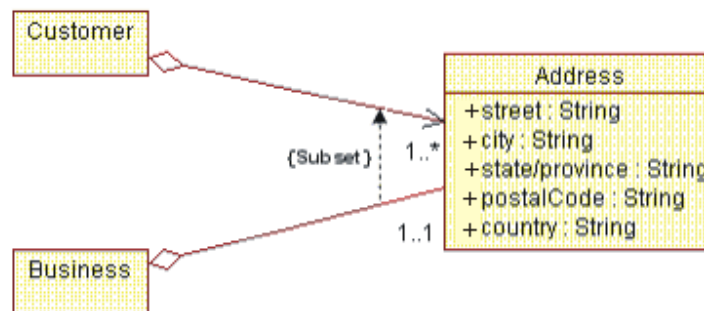
Shared Aggregation

An aggregation relationship that has a multiplicity greater than one established for the aggregate is called **shared**, and destroying the aggregate does not necessarily destroy the parts. By implication, a shared aggregation forms a graph, or a tree with many roots. Shared aggregations are used in cases where there is a strong relationship between two classes, so that the same instance can participate in two different aggregations.

Example

Consider the case where a person has a home-based business. Both the Person and the Business have an address; in fact it is the same address. The Address is an integral part of both the Person and the Business. Yet the Business may cease to exist, leaving the Person hopefully at the same address.

Note also that it is possible in this case to start off with shared aggregation, then convert to non-shared aggregation at a later date. The home-based business may grow and prosper, eventually moving into separate quarters. At this point, the Person and the Business no longer share the same address. As a result, the aggregation is no longer shared.



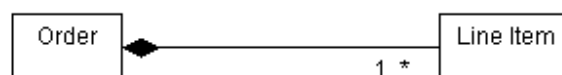
An example of shared aggregation.

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 (in the example, the **Order**) 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 compositional aggregation 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. In the example presented below, it does make sense to even have an **Order** if there is nothing being ordered (i.e. **Line Items**). In some cases, this inter-dependency can be identified as early as analysis (as in the case with this example), but more often it is not until design that such decisions can be made confidently.

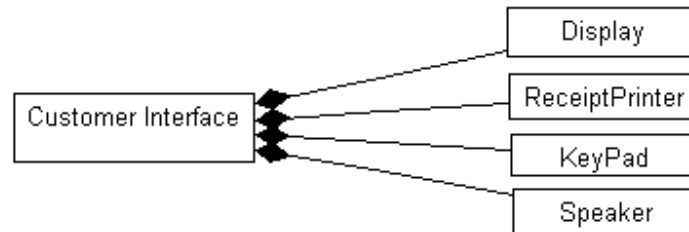
A solid filled diamond is attached to the end of an association path to indicate composition, as shown below:



An example of compositional aggregation

Example

In this example, the **Customer Interface** is composed of several other classes. In this example the multiplicities of the aggregations are not yet specified.



A **Customer Interface** object knows which **Display**, **Receipt Printer**, **KeyPad**, and **Speaker** objects belong to it.

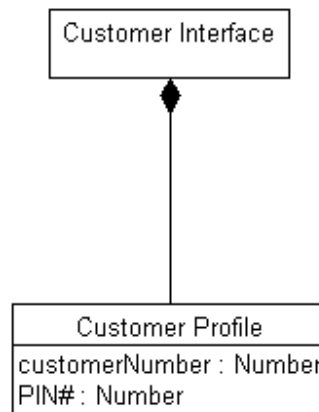
Using Composition to Model Class Properties

A property of a class is something that the class knows about. As in the case of the **Customer** class shown above, one could choose to model the **Address** of the Customer as either a class, as we have shown it, or as a set of attributes of the class. The decision whether to use a class and the aggregation relation, or a set of attributes, depends on the following:

- Do the 'properties' need to have independent identity, such that they can be referenced from a number of objects? If so, use a class and aggregation.
- Do a number of classes need to have the same 'properties'? If so, use a class and aggregation.
- Do the 'properties' have a complex structure and properties of their own? If so, use a class (or classes) and aggregation.
- Otherwise, use attributes.

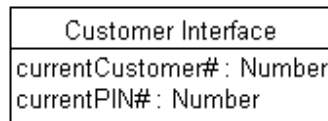
Example

In an **Automated Teller Machine**, the system must keep track of the current customer and their PIN, let us assume that the **Customer Interface** is responsible for this. This information may be thought of as "properties" of the class. This may be done using a separate class, shown as follows:



Object properties modeled using Aggregation
The alternative, having the **Customer Interface** keep track of the current Customer and their PIN using attributes, is modeled as follows:

Guidelines



Object properties modeled using Attributes

The decision of whether to use attributes or an aggregation association to a separate class is determined based the degree of coupling between the concepts being represented: when the concepts being modeled are tightly connected, use attributes. When the concepts are likely to change independently, use aggregation.

Aggregation or Association?

Aggregation should be used only in cases where there is a compositional 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.

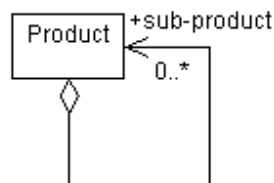
If the classes can have independent identity outside the context provided by other classes, if they are not parts of some greater whole, then the association relationship should be used. In addition, when in doubt, an association more appropriate; aggregations are generally obvious, and choosing aggregation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

Self-Aggregations

Sometimes, a class may be aggregated with itself. This does not mean that an instance of that class is composed of itself (this would be silly), it means that one instance if the class is an aggregate composed of other instances of the same class. In the case of self-aggregations, role names are essential to distinguish the purpose for the association.

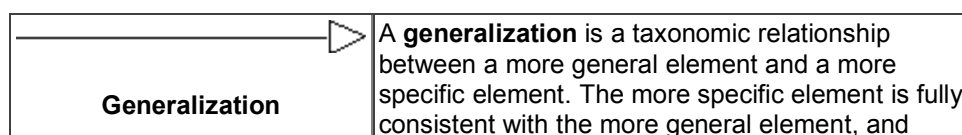
Example

Consider the following self-aggregation involving the class **Product**:



In this case, a product may be composed of other products; if they are, the aggregated products are called sub-products. The association is navigable only from the aggregate to the sub-product; i.e. sub-products would not know what products they are part of (since they may be part of many products).

Guidelines: Generalization



Guidelines

	contains additional information. An instance of the more specific element may be used where the more general element is allowed.
--	--

Topics

- [Generalization](#)
- [Multiple Inheritance](#)
- [Abstract and Concrete Classes](#)
- [Use](#)
- [Inheritance to Support Polymorphism](#)
- [Inheritance to Support Implementation Reuse](#)
- [Inheritance in Programming Languages](#)

Generalization

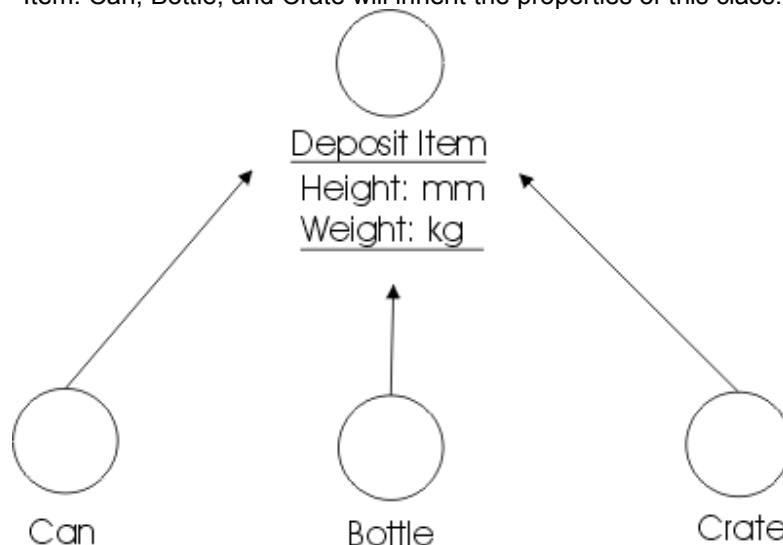
Many things in real life have common properties. Both dogs and cats are animals, for example. Objects can have common properties as well, which you can clarify using a generalization between their classes. By extracting common properties into classes of their own, you will be able to change and maintain the system more easily in the future.

A generalization shows that one class inherits from another. The inheriting class is called a descendant. The class inherited from is called the ancestor. Inheritance means that the definition of the ancestor - including any properties such as attributes, relationships, or operations on its objects - is also valid for objects of the descendant. The generalization is drawn from the descendant class to its ancestor class.

Generalization can take place in several stages, which lets you model complex, multilevel inheritance hierarchies. General properties are placed in the upper part of the inheritance hierarchy, and special properties lower down. In other words, you can use generalization to model specializations of a more general concept.

Example

In the Recycling Machine System all the classes - Can, Bottle, and Crate - describe different types of deposit items. They have two common properties, besides being of the same type: each has a height and a weight. You can model these properties through attributes and operations in a separate class, Deposit Item. Can, Bottle, and Crate will inherit the properties of this class.



Guidelines

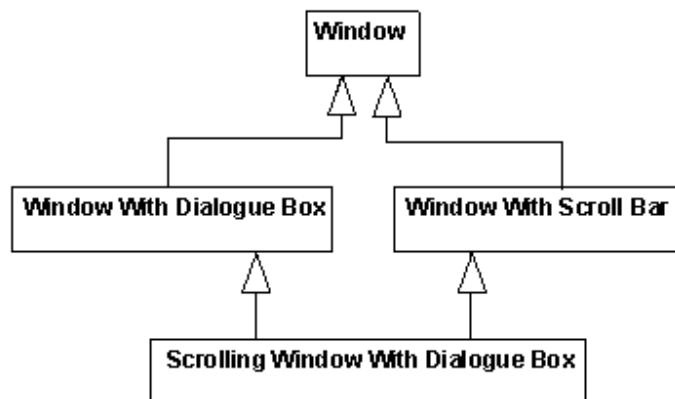
The classes Can, Bottle, and Crate have common properties height and weight. Each is a specialization of the general concept Deposit Item.

Multiple Inheritance

A class can inherit from several other classes through multiple inheritance, although generally it will inherit from only one.

There are a couple of potential problems you must be aware of if you use multiple inheritance:

- If the class inherits from several classes, you must check how the relationships, operations, and attributes are named in the ancestors. If the same name appears in several ancestors, you must describe what this means to the specific inheriting class, for example, by qualifying the name to indicate its source of declaration.
- If repeated inheritance is used; in this case, the same ancestor is being inherited by a descendant more than once. When this occurs, the inheritance hierarchy will have a "diamond shape" as shown below.



Multiple and repeated inheritance. The Scrolling Window With Dialog Box class is inheriting the Window class more than once.

A question that might arise in this context is "How many copies of the attributes of Window are included in instances of Scrolling Window With Dialog Box?" So, if you are using repeated inheritance, you must have a clear definition of its semantics; in most cases this is defined by the programming language supporting the multiple inheritance.

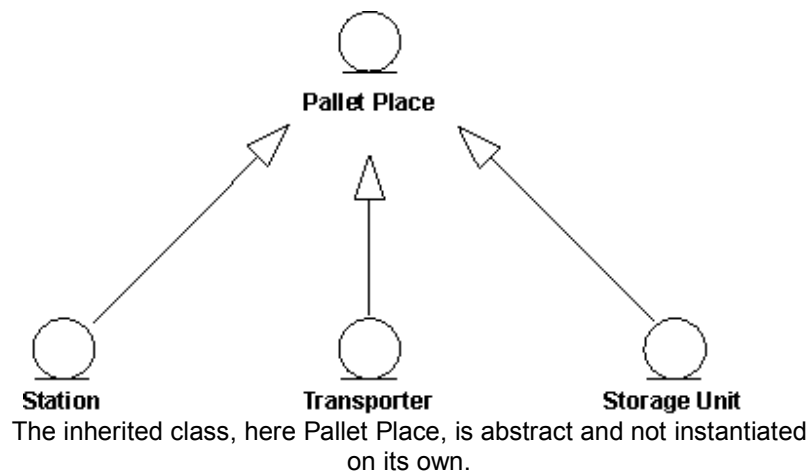
In general, the programming language rules governing multiple inheritance are complex, and often difficult to use correctly. Therefore using multiple inheritance only when needed, and always with caution is recommended.

Abstract and Concrete Classes

A class that is not instantiated and exists only for other classes to inherit it, is an abstract class. Classes that are actually instantiated are concrete classes. Note that an abstract class must have at least one descendant to be useful.

Example

A Pallet Place in the Depot-Handling System is an abstract entity class that represents properties common to different types of pallet places. The class is inherited by the concrete classes Station, Transporter, and Storage Unit, all of which can act as pallet places in the depot. All these objects have one common property: they can hold one or more Pallets.



Use

Because class stereotypes have different purposes, inheritance from one class stereotype to another does not make sense. Letting a boundary class inherit an entity class, for example, would make the boundary class into some kind of hybrid. Therefore, you should use generalizations only between classes of the same stereotype.

You can use generalization to express two relationships between classes:

- Subtyping, specifying that the descendant is a subtype of the ancestor. Subtyping means that the descendant inherits the structure and behavior of the ancestor, and that the descendant is a type of the ancestor (that is, the descendant is a subtype that can fill in for all its ancestors in any situation).
- Subclassing, specifying that the descendant is a subclass (but not a subtype) of the ancestor. Subclassing means that the descendant inherits the structure and behavior of the ancestor, and that the descendant is not a type of the ancestor.

You can create relationships such as these by breaking out properties common to several classes and placing them in a separate classes that the others inherit; or by creating new classes that specialize more general ones and letting them inherit from the general classes.

If the two variants coincide, you should have no difficulty setting up the right inheritance between classes. In some cases, however, they do not coincide, and you must take care to keep the use of inheritance understandable. At the very least you should know the purpose of each inheritance relationship in the model.

Inheritance to Support Polymorphism

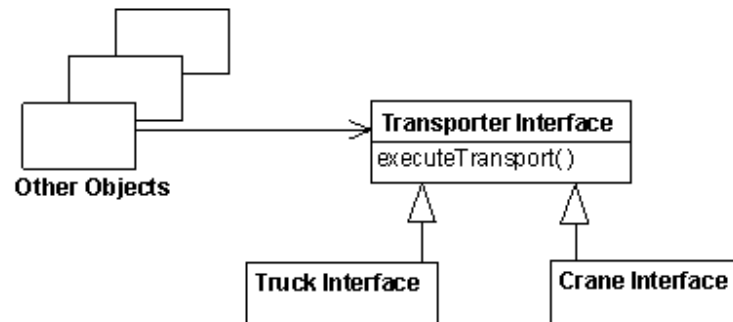
Subtyping means that the descendant is a subtype that can fill in for all its ancestors in any situation. Subtyping is a special case of polymorphism, and is an important property because it lets you design all the clients (objects that use the ancestor) without taking the ancestor's potential descendants into consideration. This makes the client objects more general and reusable. When the client uses the actual object, it will work in a specific way, and it will always find that the object does its task. Subtyping ensures that the system will tolerate changes in the set of subtypes.

Example

In a Depot-Handling System, the Transporter Interface class defines basic functionality for communication with all types of transport equipment, such as

Guidelines

cranes and trucks. The class defines the operation `executeTransport`, among other things.



Both the **Truck Interface** and **Crane Interface** classes inherit from the **Transporter Interface**; that is, objects of both classes will respond to the message `executeTransport`. The objects may stand in for **Transporter Interface** at any time and will offer all its behavior. Thus, other objects (client objects) can send a message to a **Transporter Interface** object, without knowing if a **Truck Interface** or **Crane Interface** object will respond to the message.

The **Transporter Interface** class can even be abstract, never instantiated on its own. In which case, the **Transporter Interface** might define only the signature of the `executeTransport` operation, whereas the descendant classes implement it.

Some object-oriented languages, such as C++, use the class hierarchy as a type hierarchy, forcing the designer to use inheritance to subtype in the design model. Others, such as Smalltalk-80, have no type checking at compile time. If the objects cannot respond to a received message they will generate an error message.

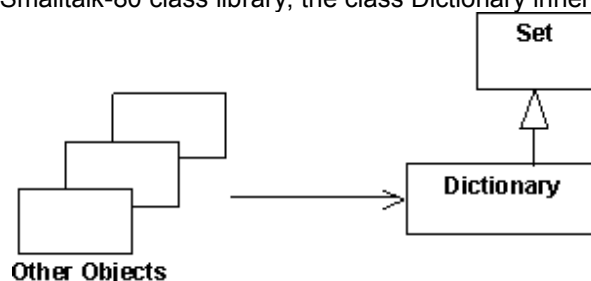
It may be a good idea to use generalization to indicate subtype relationships even in languages without type checking. In some cases, you should use generalization to make the object model and source code easier to understand and maintain, regardless of whether the language allows it. Whether or not this use of inheritance is good style depends heavily on the conventions of the programming language.

Inheritance to Support Implementation Reuse

Subclassing constitutes the reuse aspect of generalization. When subclassing, you consider what parts of an implementation you can reuse by inheriting properties defined by other classes. Subclassing saves labor and lets you reuse code when implementing a particular class.

Example

In the Smalltalk-80 class library, the class **Dictionary** inherits properties from **Set**.



The reason for this generalization is that **Dictionary** can then reuse some general methods and storage strategies from the implementation of **Set**. Even though a **Dictionary** can be seen as a **Set**

Guidelines

(containing key-value pairs), Dictionary is not a subtype of Set because you cannot add just any kind of object to a Dictionary (only key-value pairs). Objects that use Dictionary are not aware that it actually is a Set.

Subclassing often leads to illogical inheritance hierarchies that are difficult to understand and to maintain. Therefore, it is not recommended that you use inheritance only for reuse, unless something else is recommended in using your programming language. Maintenance of this kind of reuse is usually quite tricky. Any change in the class Set can imply large changes of all classes inheriting the class Set. Be aware of this and inherit only stable classes. Inheritance will actually freeze the implementation of the class Set, because changes to it are too expensive.

Inheritance in Programming Languages

The use of generalization relationships in design should depend heavily on the semantics and proposed use of inheritance in the programming language. Object-oriented languages support inheritance between classes, but nonobject-oriented languages do not. You should handle language characteristics in the design model. If you are using a language that does not support inheritance, or multiple inheritance, you must simulate inheritance in the implementation. In which case, it is better to model the simulation in the design model and not use generalizations to describe inheritance structures. Modeling inheritance structures with generalizations, and then simulating inheritance in the implementation, can ruin the design.

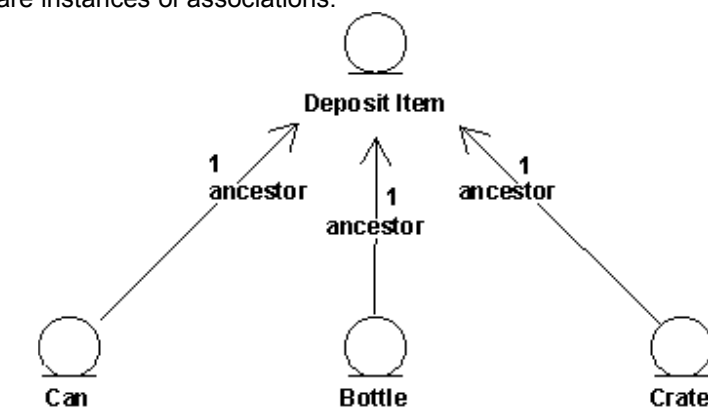
If you are using a language that does not support inheritance, or multiple inheritance, you must simulate inheritance in the implementation. In this case, it is best to model the simulation in the design model and not use generalizations to describe inheritance structures. Modeling inheritance structures with generalizations, and then only simulating inheritance in the implementation can ruin the design.

You will probably have to change the interfaces and other object properties during simulation. It is recommended that you simulate inheritance in one of the following ways:

1. By letting the descendant forward messages to the ancestor.
2. By duplicating the code of the ancestor in each descendant. In this case, no ancestor class is created.

Example

In this example the descendants forward messages to the ancestor via links that are instances of associations.



Behavior common to Can, Bottle, and Crate objects is assigned to a special class. Objects for which this behavior is common send a message to a Deposit Item object to perform the behavior when necessary.

Moving From Analysis Classes to Design

Topics

- Mapping from the Analysis Model
- Mapping to the Implementation Model
- Characteristics of a good Design Model

Mapping from the Analysis Model

Artifact: Analysis Classes represent roles played by instances of design elements; these roles may be fulfilled by one or more design model elements. In addition, a single design element may fulfill multiple roles. The following observations discuss the ways the analysis roles may be fulfilled:

- An analysis class can become one single class in the design model.
- An analysis class can become a part of a class in the design model.
- An analysis class can become an aggregate class in the design model. (Meaning that the parts in this aggregate may not be explicitly modeled in the analysis model.)
- An analysis class can become a group of classes that inherits from the same class in the design model.
- An analysis class can become a group of functionally related classes in the design model.
- An analysis class can become a package in the design model (meaning that it can become a component.)
- An analysis class can become a relationship in the design model.
- A relationship between analysis classes can become a class in the design model.
- Analysis classes handle primarily functional requirements, and model objects from the "problem" domain; design classes handle non-functional requirements, and model objects from the "solution" domain.
- Analysis classes can be used to represent "the objects we want the system to support," without taking a decision on how much of them to support with hardware and how much with software. Thus, part of an analysis class can be realized by hardware, and not modeled in the design model at all.

Any combination of the above are also possible.

Mapping to the Implementation Model

You should decide before the design starts how classes in the design model should relate to implementation classes; this should be described in the Design Guidelines specific to the project. The design model can be more or less close to the implementation model, depending on how you map its classes, packages and subsystems to components, packages and subsystems in the implementation model. During implementation, you will often address small tactical issues related to the implementation environment that shouldn't have impact on the design model. For example, classes and subsystems can be added during implementation to handle parallel development, or to adjust import dependencies. For more information, refer to Activity: Structure the Implementation Model and Concepts: Mapping from Design to Code.

There should be a consistent mapping from the design model to the implementation model. The Artifact: Design Guidelines should define this mapping, and a consistent level of abstraction should be applied across the design model.

Characteristics of a Good Design Model

Guidelines

A good design model has the following characteristics:

- It satisfies the system requirements.
- It is resistant to changes in the implementation environment.
- It is easy to maintain in relation to other possible object models and to system implementation.
- It is clear how to implement.
- It does not include information that is best documented in program code.
- It is easily adapted to changes in requirements.