Object-Oriented Analysis and Design

Chapter 2

UNIFIED MODELING LANGUAGE

2016-17
Mr. Taing PengSreng (MSc, RUPP)

Unified Modeling Language

CONTENTS

Class Diagram

CONTENTS

Class Diagram?

- Class diagrams are used to capture the static relationships of your software, in other words, how things are put together.
- When writing software you are constantly making design decisions: what classes hold references to other classes, which class "owns" some other class, and so on.
- Class diagrams provide a way to capture this "physical" structure of a system.

CONTENTS

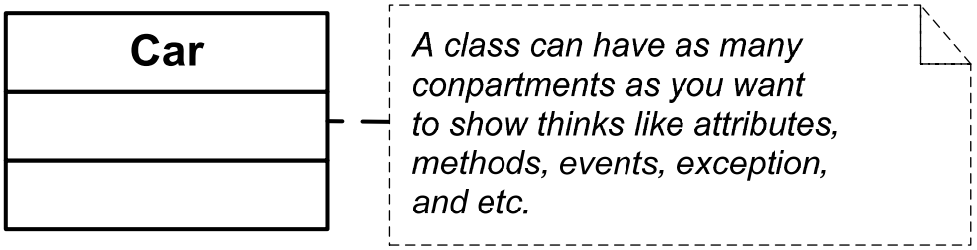
Classes

- A class represents a group of things that have common states and behaviors.
- A class is presented with a rectangular box divided into *compartments* for writing information.
- The first compartment holds the name of the class, the second holds attributes, and the third is used for operations.
- When reading a diagram, you can make no assumptions about a missing compartment; it doesn't mean it is empty.
- You may add compartments to a class to show additional information, such as exceptions or events, though this is outside of the typical notation.

CONTENTS

Classes (cont.)

- UML suggests that the class name:
  - Start with a capital letter
  - Be centered in the top compartment
  - Be written in a boldface font
  - Be written in italics if the class is *abstract*
- Simple class representation



CONTENTS

Class Diagram (cont.)

- An object is an *instance* of a class.
- In the object diagram, an object is shown the name followed by its type (i.e., class)
- You show that this is an instance of a class by underlining the name and type.
- An instance of a type class **Car** in object diagram.

CONTENTS

Classes (cont.)

Attributes

- Details of a class (the price of a product, the color of a car, etc.) are represented as *attributes*.
- Attributes can be simple primitive types (integers, floating-point numbers, etc.) or relationships to other, complex objects.
- An attribute can be shown using two different notations: inlined or relationships between classes. In addition, notation is available to show such things as multiplicity, uniqueness, and ordering.

CONTENTS

Classes (cont.)

- A class's attributes listed in rectangle notation; these are typically called *inlined attributes*.
- There is no semantic difference between inlined attributes and attributes by relationship.
- It's simply a matter of how much detail you want to present (or, in the case of primitives like integers, how much detail you can present.)
- To represent an attribute within the body of a class, place the attribute in the second compartment of the class.

CONTENTS

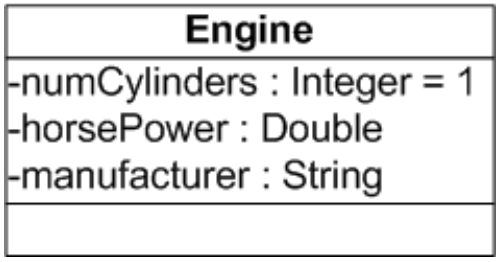
Classes (cont.)

- UML refers to **inlined** attributes as *attribute notation*. **Inlined** attributes use the following notation:

  visibility name multiplicity : type = default
  {property strings and constraints}

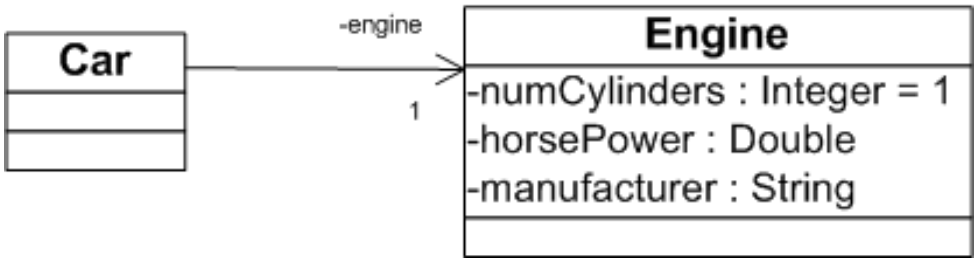
  - *visibility is {+|-|#|~}*
  - *multiplicity forms [lower..upper]*

CONTENTS



Classes (cont.)

- Some attributes are represented using the relationship notations. They results in a larger class diagram, but it can provide greater detail for complex attribute types.
- The relationship notation also conveys exactly how the attribute is contained within a class.



Classes (cont.)

Operations

- Operations are features of classes that specify how to invoke a particular behavior. For example, a class may offer an operation to draw a rectangle on the screen.
- UML makes a clear distinction between the specification of how to invoke a behavior (an operation) and the actual implementation of that behavior (a method.)
- Static operations are indicated with underlining.
- A **method** is an implementation of an operation. UML has no notation for a method.

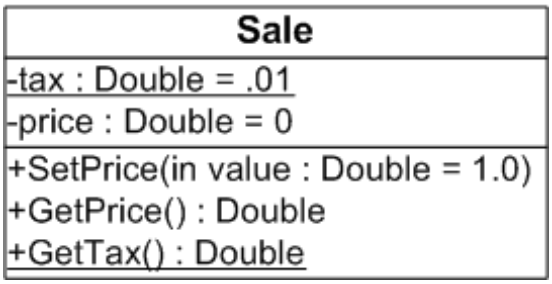
Classes (final)

- Operations are placed in a separate compartment with the following syntax:

  visibility name(parameters):return-type {properties}

  where parameters are written as:

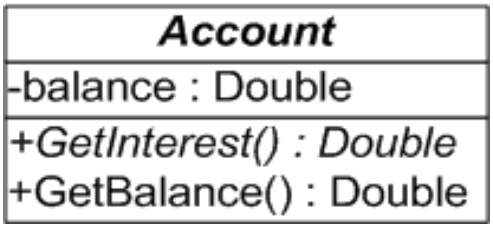
  direction parameter_name : type [multiplicity] = default_value { properties }



CONTENTS

Abstract Classes

- An abstract class is typically a class that provides an operation signature, but no implementation
- An abstract class might have no operations at all.
- An abstract class is useful for identifying common functionality across several types of objects.
- UML shows an abstract class by writing its name in italics, and its abstract operation in italics as well.



CONTENTS

Relationships

- Classes in isolation would not provide much insight into how a system is designed.
- UML provides several ways of representing relationships between classes.
- Each of UML relationship represents a different type of connection between classes and has subtleties that aren't fully captured in the UML specification.
- When modeling in the real world, be sure that your intended viewers understand what you are conveying with your various relationships.

CONTENTS

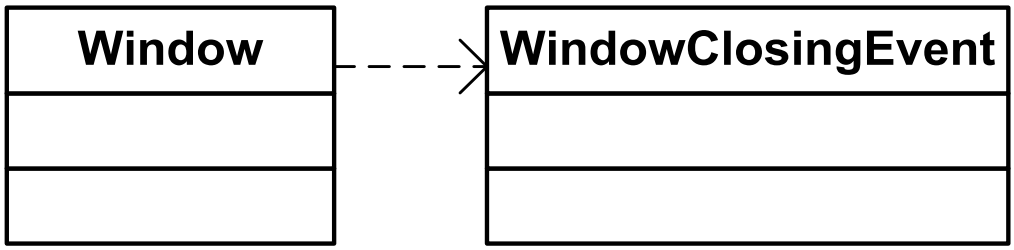
Relationships (cont.)

- The weakest relationship between classes is a **dependency** relationship.
- Dependency between classes means that one class uses, or has knowledge of, another class.
- It is typically a transient relationship, meaning a dependent class briefly interacts with the target class but typically doesn't retain a relationship with it for any real length of time.
- Dependencies are typically read as "...uses a...".

CONTENTS

Relationships (cont.)

- For example, if you have a class named **Window** that sends out a class named **WindowClosingEvent** when it is about to be closed, you would say "*Window uses a WindowClosingEvent.*"
- UML shows a dependency between classes using a dashed line with an arrow pointing from the dependent class to the class that is used.



CONTENTS

Relationships (cont.)

- **Associations** are stronger than dependencies and typically indicate that one class retains a relationship to another class over an extended period of time.
- The lifelines of two objects linked by associations are probably not tied together (meaning one can be destroyed without necessarily destroying the other).
- Associations are typically read as "...has a...".
- For example, if you have a class named **Window** that has a reference to the current mouse cursor, you would say "*Window has a Cursor*".

CONTENTS

Relationships (cont.)

- UML shows an association using a solid line between the classes participating in the relationship.

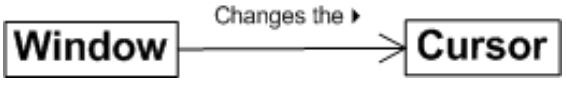
- Associations have explicit notation to express navigability. If you can navigate from one class to another, you show an arrow in the direction of the class you can navigate to.

- If you can navigate in both directions, it is common practice to not show any arrows at all.

CONTENTS

Relationships (cont.)

- Associations may be adorned with several symbols to add information to your model. The simplest is a solid arrowhead showing the direction in which the viewer should read the association.
- It is common to include a short phrase along with the arrowhead to provide some context for the association. The phrase used with the association doesn't typically generate into any form of code representation; it is purely for modeling purposes.



CONTENTS

Relationships (cont.)

- In association, you can express how many instances of a particular class are involved in a relationship.
- If you don't specify a value, a multiplicity of 1 is assumed.
- To show a different value, simply place the multiplicity specification near the owned class.

CONTENTS

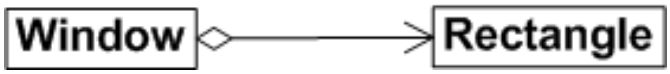
Relationships (cont.)

- **Aggregation** is a stronger version of association. Unlike association, aggregation typically implies ownership and may imply a relationship between lifelines.
- For example, if you had a classed named Window that stored its position and size in a Rectangle class, you would say the "Window owns a Rectangle." The rectangle may be shared with other classes, but the Window has an intimate relationship with the Rectangle.

CONTENTS

Relationships (cont.)

- Aggregations are usually read as "...owns a...".
- You show an aggregation with a diamond shape next to the owning class and a solid line pointing to the owned class.
- An example of aggregation between a class named Window and a class named Rectangle is shown.



CONTENTS

Relationships (cont.)

- **Composition** represents a very strong relationship between classes, to the point of containment.
- Composition is used to capture a whole-part relationship.
- The "part" piece of the relationship can be involved in only one composition relationship at any given time.
- The lifetime of instances involved in composition relationships is almost always linked; if the larger, owning instance is destroyed, it almost always destroys the part piece.
- UML does allow the part to be associated with a different owner before destruction, thus preserving its existence, but this is typically an exception rather than the rule.

CONTENTS

Relationships (cont.)

- A composition relationship is usually read as "...is part of...", which means you need to read the composition from the part to the whole.
- For example, a window in your system must have a title bar, you can represent this with a class named **TitleBar** that "...is part of..." a class named Window.
- You show a composition relationship using a filled diamond next to the owning class and a solid line pointing to the owned class.

CONTENTS

Relationships (cont.)

- A **generalization** relationship conveys that the target of the relationship is a general, or less specific, version of the source class or interface.
- Generalization relationships are often used to pull out commonality between difference classifiers.
- For example, if you had a class named **Cat** and a class named **Dog**, you can create a generalization of both of those classes called Animal.

CONTENTS

Relationships (cont.)

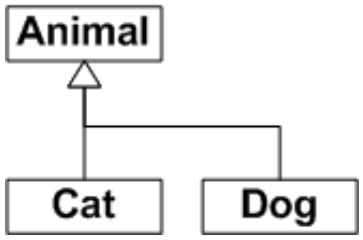
- Generalizations are usually read as "...is a...", starting from the more specific class and reading toward the general class.
- Going back to the Cat and Dog example, you would say "a Cat...is a...Animal“.
- You show a generalization relationship with a solid line with a closed arrow, pointing from the specific class to the general class.
- Unlike associations, generalization relationships are typically not named and don't have any kind of multiplicity.

CONTENTS

Relationships (final)

- UML allows for multiple inheritance, meaning a class can have more than one generalization with each representing an aspect of the decedent class.
- Some modern languages (e.g., Java and C#) don't support multiple inheritance; interfaces and interface realization are used instead.

CONTENTS



Interface

- An **interface** is a classifier that has declarations of properties and methods but no implementations.
- You can use interfaces to group common elements between classifiers and provide a contract a classifier that provides an implementation of an interface must obey.
- For example, you can create an interface named **Sortable** that has one operation named **comesBefore**(...).

CONTENTS

Interface (cont.)

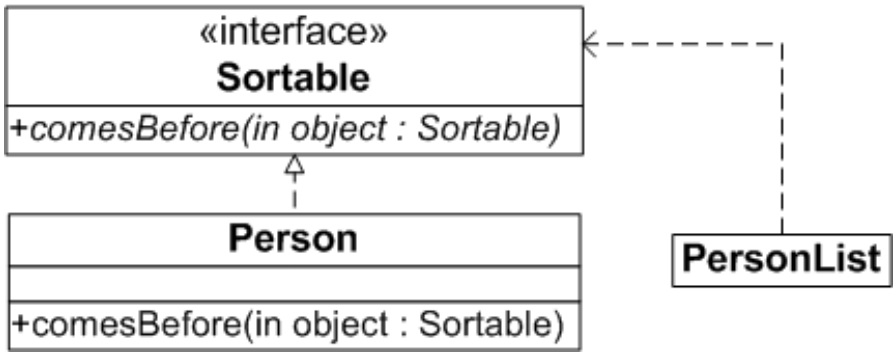
- An interface's representation in the standard UML classifier notation with the stereotype «interface» is

- A class is said to *realize* an interface if it provides an implementation for the operations and properties.
- For example, any class that realizes the **Sortable** interface must provide an implementation of **comesBefore**(...).

CONTENTS

Interface (final)

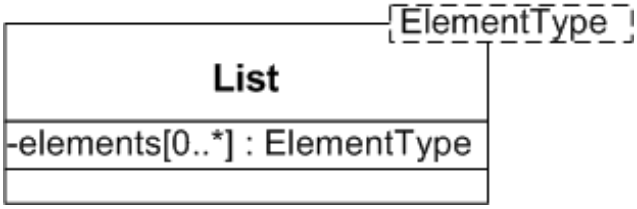
- Realization is shown using a dashed line starting at the realizing classifier and leading to the interface, with a closed arrowhead at the end.
- Classes that are dependent on the interface are shown using a dashed line with an open arrow (dependency).



CONTENTS

Templates

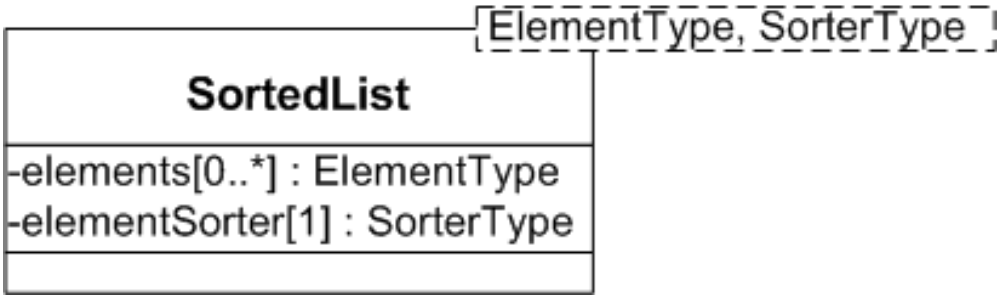
- **Template** is called Parameterized class.
- Template describes a class with one or more unbound formal parameters.
- It defines a family of classes, each of which is specified by binding the parameters to actual values.
- The formal parameters are listed in a dashed rectangle in the upper right corner of the shape.



CONTENTS

Templates (cont.)

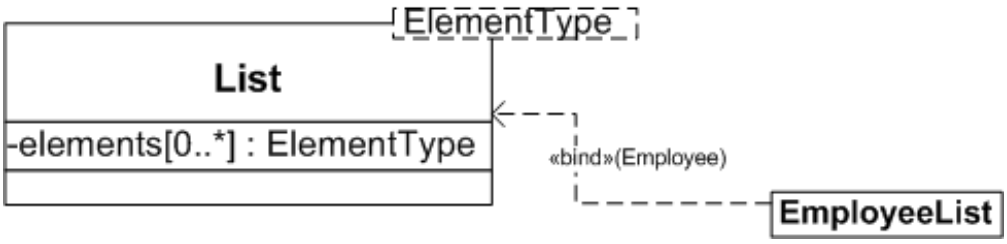
- A template class might have multiple typed parameters separated by a comma (,).
- Example - a more complicated version of the List class that requires a Sorter along with the type of object to store in the list.



CONTENTS

Templates (final)

- An instance of a template class is created with the actual type to use in place of ElementType.
- This is called *binding* a type to a template.
- Creating a bound version of a template class called **explicit binding**.
- For example - a actual class of List, EmployeeList, binds the ElementType of List to a class named Employee.



CONTENTS

Use Case Diagram

CONTENTS

Use Case Diagram?

CONTENTS

- Use case diagram is a way to capture system functionality and requirements in UML.
- They describe how external entities will use the system. These external entities can be either humans or other systems and are referred to as actors in UML terminology.
- The description emphasizes the users' view of the system and the interaction between the users and the system.

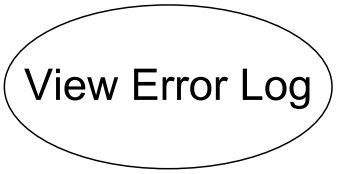
Use Case Diagram? (final)

- Use case diagrams consist of named pieces of functionality (use cases), the persons or systems or things invoking the functionality (actors), and possibly the elements responsible for implementing the use cases (subjects).
- They help to further define system scope and boundaries.
- They are usually in the form of a diagram, along with a textual description of the interaction taking place.

CONTENTS

Use Case

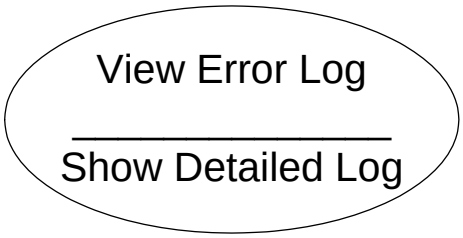
- Use cases represent distinct pieces of functionality for a system, a component, or even a class.
- Each use case must have a name that is typically a few words describing the required functionality, such as **View Error Log**.
- In UML, use case is an oval with the name of the use case in the center.
- Example – Simple use case.



CONTENTS

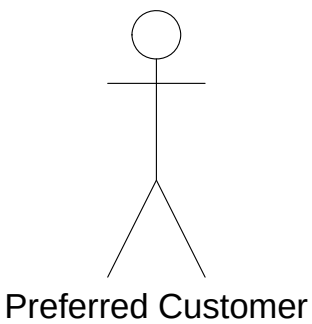
Use Case (final)

- A use case's oval can be divided into compartments to provide more detail about the use case, such as extension points, included use cases, or the modeling of specific constraints.
- Example - a use case oval with a compartment listing extension points.



Actor

- A use case must be initiated by someone or something outside the scope of the use case. This interested party is called an **actor**.
- An actor doesn't need to be a human user. Any external system or element outside may trigger the use case should be modeled as an actor.
- An actor is represented in UML as a stick figure with the name of the actor (usually right below it.)

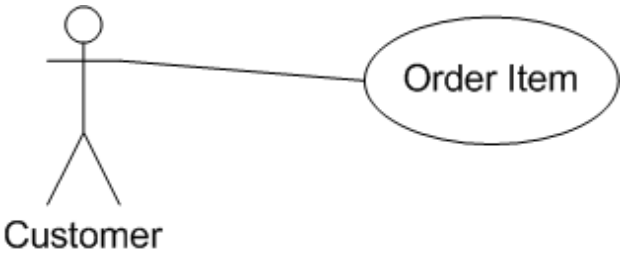


Actor and Use Case Association

- An actor can be associated one or more use cases.
- A relationship between an actor and a use case indicates the actor initiates the use case, the use case provides the actor with results, or both.
- You show an association between an actor and a use case as a solid line.
- Conventionally you read use case diagrams from left to right, with actors initiating use cases on the left and actors that receive use case results on the right. However, depending on the model or level of complexity, it may make sense to group actors differently.

Actor and Use Case Association (cont.)

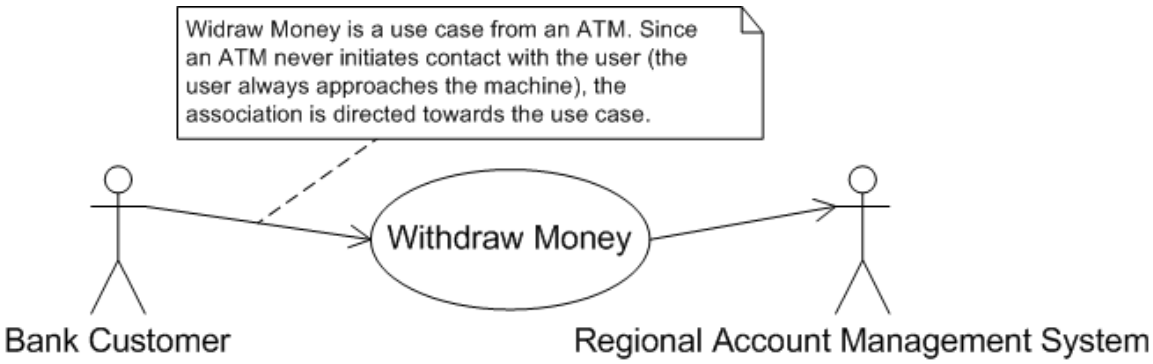
- Example - shows an actor communicating with a use case.



- Though not part of the official UML specification, it is common to see directional arrows on association lines to indicate who initiates communication with whom.

Actor and Use Case Association (final)

- Note that the arrows don't necessarily restrict the direction of information flow; they simply point from the initiator to the receiver of the communication.



CONTENTS

System Boundary

- By definition, use cases capture the functionality of a particular subject.
- Anything not realized by the subject is considered outside the system boundaries and should be modeled as an actor.
- This technique is very useful in determining the scope and assignment of responsibilities when designing a system, subsystem, or component.

CONTENTS

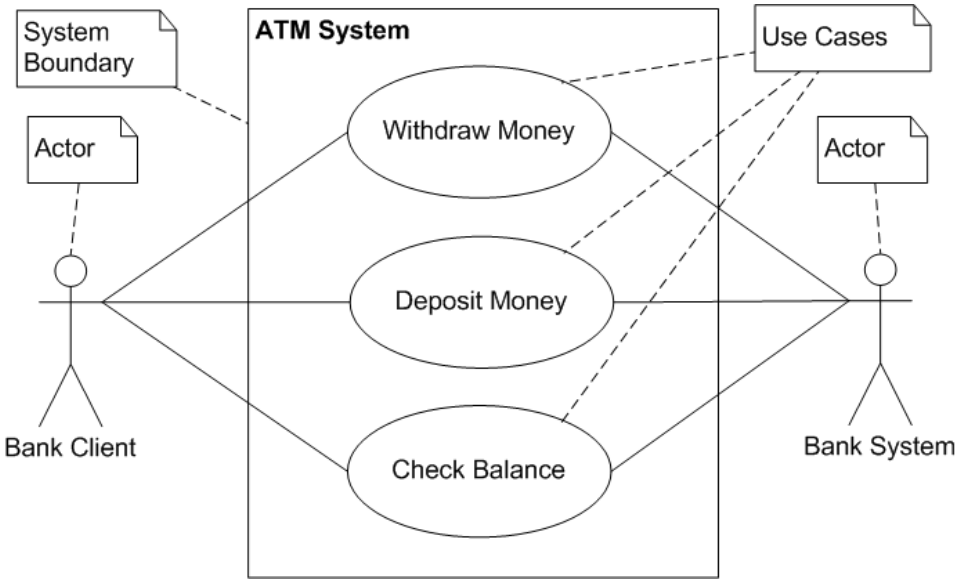
System Boundary (cont.)

- While you are modeling an ATM system, your design discussions digress into discussions of the details of the back-end banking system, a use case model with clearly defined system boundaries would identify the banking system as an actor and therefore outside the scope of the problem.
- System boundaries are represented in a generic sense using a simple rectangle, with the name of the system at the top.

CONTENTS

System Boundary (final)

- Example – A use case diagram showing the system boundaries of an ATM system.



CONTENTS

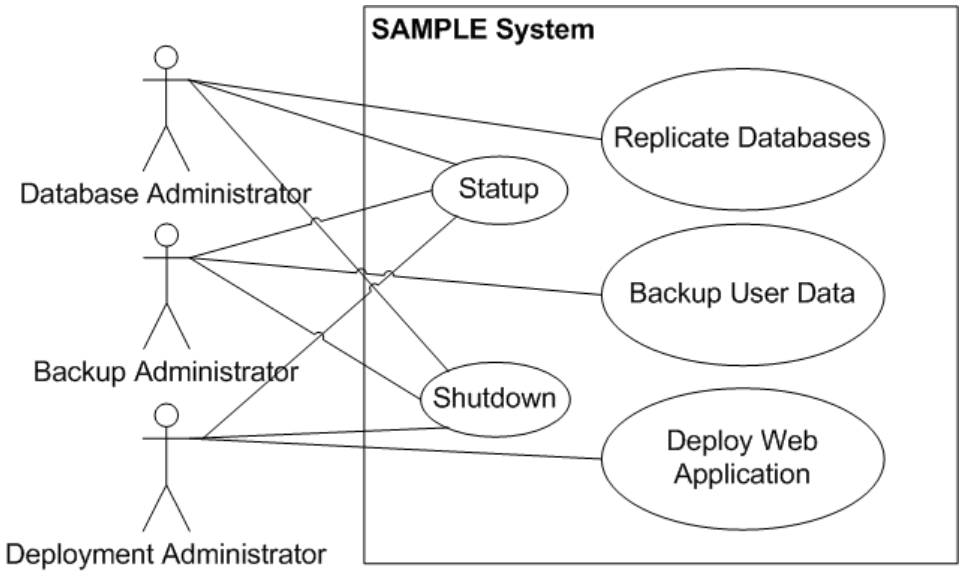
Actor and Functionality Identification

- Actors don't need to have a one-to-one mapping to physical entities; in fact, they don't need to be physical entities at all.
- UML allows for actors to represent roles of potential users of a system.
- For example, the system administrator may be the only physical user of a system, but that administrator may wear many hats. It may be helpful to view the system from the perspective of a database administrator, backup administrator, deployment administrator, and so on.

CONTENTS

Actor and Functionality Identification (final)

- Example – using specialized versions of an actors to help find required functionality.



CONTENTS

Advanced Use Case Modeling

- As it does for other classifiers, UML provides mechanisms for reusing and adding on to use cases and actors.
- You can expand an actor's capabilities or replace entire use cases using **generalization**.
- You can factor out common elements of use cases using **included** use cases, or add on to base use cases using use case **extension**.

CONTENTS

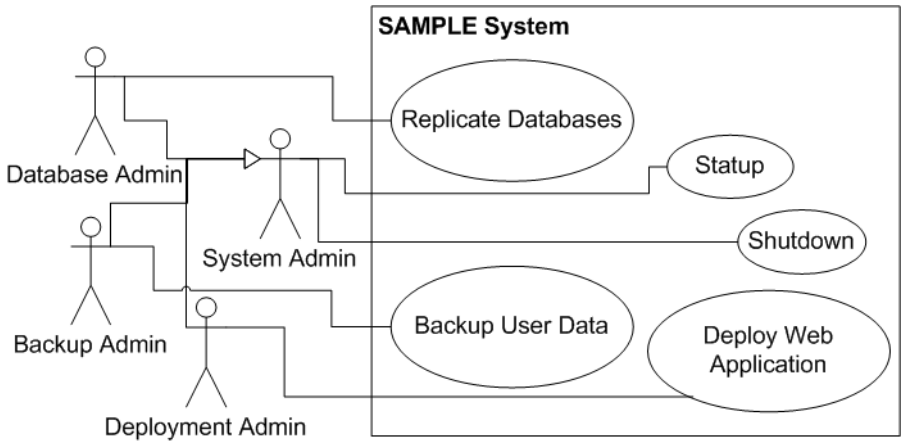
Actor and Use Case Generalization

- Use cases can be generalized like many other classifiers.
- Actor generalization is typically used to pull out common requirements from several different actors to simplify modeling.
- You may have a **Database Administrator**, a **Backup Administrator**, and a **Deployment Administrator**, all with slightly different needs.
- However, the majority of the needs of the individual actors may overlap. You can factor out a generic System Administrator actor to capture the common functionality, and then specialize to identify the unique needs of each actor.

CONTENTS

Actor and Use Case Generalization (final)

- You represent **actor generalization** like any other classifier; draw a solid line, with a closed arrow pointing from the specialized actor to the base actor.
- Example - shows actor generalization for a much easier-to-read diagram.

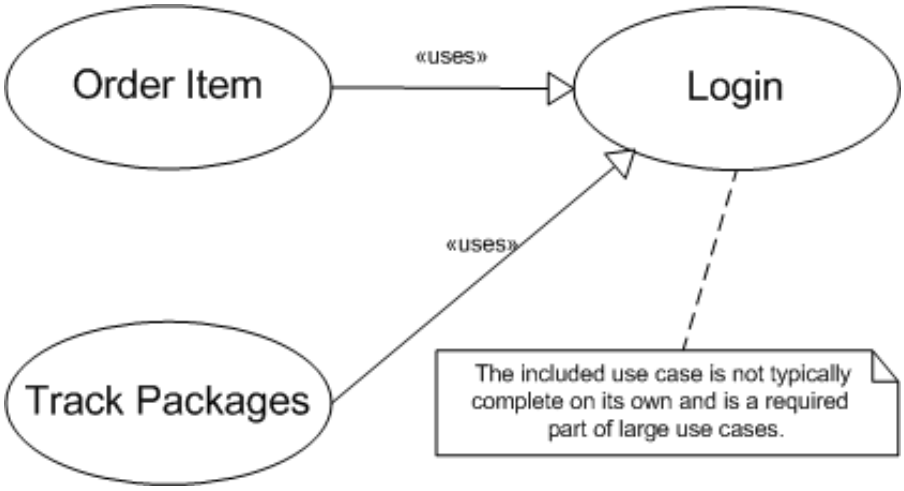


Use Case Inclusion

- Common functionality from several use cases can be factored out by creating a shared, included use case.
- The use case that includes another use case is typically not complete on its own. The included functionality isn't considered optional; it is factored out simply to allow for reuse in other use cases.
- You show use case inclusion using a dashed line, with an open arrow (dependency) pointing from the base use case to the included use case. Label the line with the keyword **include** (in Visio, it is solid line, closed arrow, keyword **Uses**).

Use Case Inclusion (final)

- Example – Use case conclusion



CONTENTS

Use Case Extension

- UML provides the ability to plug in additional functionality to a base use case if specified conditions are met.
- For example, in modeling a banking application, you can have a use case named **Open Account** that specifies how the user can create a new account with the bank. You can offer a joint account that allowed a user to add other people to his account. The joint account functionality can be captured with a different, use case named **Add Joint Member**. In this case the specified condition for the extension is more than one member on the bank application.

CONTENTS

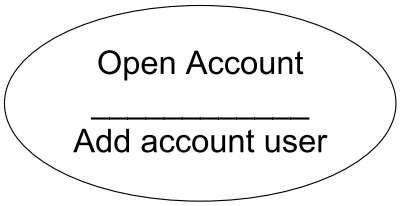
Use Case Extension (cont.)

- UML clearly specifies that a base use case should be a complete use case on its own. The extension use cases are typically smaller in scope and represent additional functionality, so they may not be useful outside the context of the base use case.
- Any use case you want to extend must have clearly defined ***extension points***.
- An extension point is a specification of some point in the use case where an extension use case can plug in and add functionality.
- UML doesn't have a particular syntax for extension points; they are typically freeform text, or step numbers if the use case functionality is represented as a numbered list.

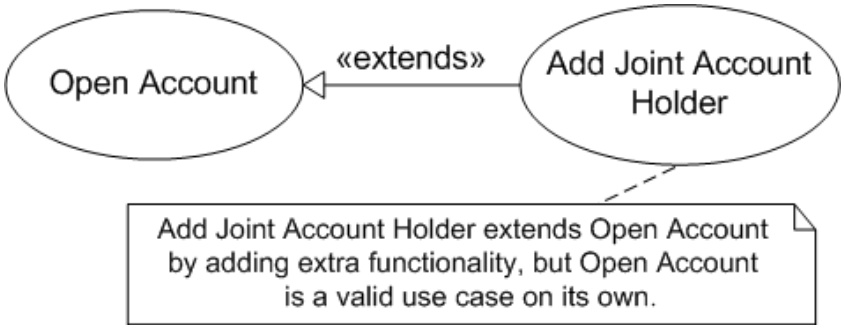
CONTENTS

Use Case Extension (final)

- Extension points are listed in a use case oval.



- A use case extension is represented by showing a dashed line with an open arrow (in Visio, solid line with closed arrow) pointing from the extension use case to the base use case.



CONTENTS