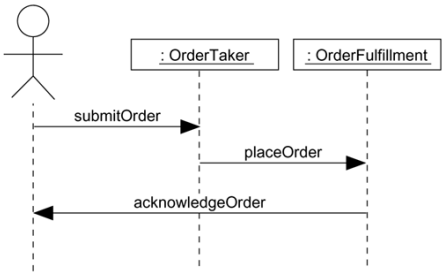
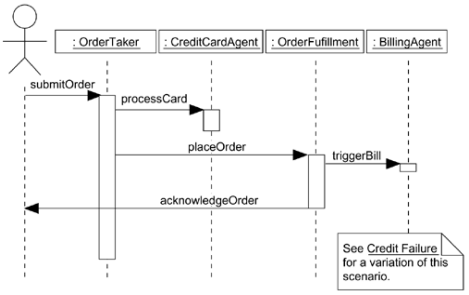


Interaction Diagram at a High Level of Abstraction



Interaction at a Low Level of Abstraction

Modeling Complex Views

- To model complex views,
  - First, convince yourself there's no meaningful way to present this information at a higher level of abstraction, perhaps eliding some parts of the diagram and retaining the detail in other parts.

  - If your diagram is still complex, print it in its entirety and hang it on a convenient large wall. You lose the interactivity an online version of the diagram brings, but you can step back from the diagram and study it for common patterns.

Module-2

Advanced Structural Modeling

Advanced Classes

Advanced classes in UML (Unified Modeling Language) encompass more complex features and relationships than the basic class diagrams. They help in modeling sophisticated systems with intricate structures and interactions. Here's an overview of advanced class modeling concepts in UML:

1. Generalization and Specialization

Generalization:

- **Definition**: Represents an inheritance relationship where a subclass inherits attributes and behaviors from a superclass.

- **Notation**: An arrow with a hollow triangle pointing from the subclass to the superclass.

- **Example**: A Vehicle class might be generalized into Car and Truck subclasses.

Specialization:

- **Definition**: The process of creating a subclass that extends or refines the behavior of a superclass.
- **Notation**: Similar to generalization, where the subclass inherits properties and behaviors from the superclass.

Use Case: Use generalization to define common features in a superclass and specific features in subclasses, reducing redundancy and promoting reuse.

2. Realization

Definition:

- Represents an interface implementation by a class. It shows how a class fulfills the contract specified by an interface.

Notation:

- An arrow with a dashed line and a hollow triangle pointing from the implementing class to the interface.

Example: If you have an interface Drawable with methods like draw(), a class Circle that implements this interface will show a realization relationship to Drawable.

Use Case: Use realization to define how classes implement the behavior specified by interfaces, ensuring a consistent contract across different implementations.

3. Dependency

Definition:

- Indicates a relationship where a class depends on another class to function properly. Changes in the dependent class may affect the class that depends on it.

Notation:

- A dashed arrow pointing from the dependent class to the class it depends on.

Example: A Printer class might depend on a Paper class to perform printing operations.

Use Case: Use dependency to show how classes interact or rely on each other, highlighting areas where changes may have ripple effects.

4. Aggregation and Composition

Aggregation:

- **Definition**: Represents a whole-part relationship where the part can exist independently of the whole. It indicates a weaker relationship than composition.
- **Notation**: A diamond at the end of the association line near the whole class.
- **Example**: A Library class and a Book class, where a library contains books but books can exist independently.

Composition:

- **Definition**: Represents a strong whole-part relationship where the part cannot exist independently of the whole. It indicates a stronger relationship than aggregation.
- **Notation**: A filled diamond at the end of the association line near the whole class.
- **Example**: A House class and a Room class, where rooms cannot exist independently of a house.

Use Case: Use aggregation for loosely coupled whole-part relationships and composition for tightly coupled whole-part relationships where the lifecycle of the part is tied to the whole.

5. Association Classes

Definition:

- A class that represents an association between two or more classes. It allows you to add attributes and methods to the association itself.

Notation:

- A class diagram where an association line is connected to a class representing the association class.

Example: A Student and Course association with an association class Enrollment that holds additional information like grade and enrollmentDate.

Use Case: Use association classes to model complex relationships that require additional information or behavior beyond a simple association.

6. Multiplicity and Role Names

Multiplicity:

- **Definition**: Specifies the number of instances of a class that can be associated with an instance of another class.
- **Notation**: Numbers or ranges at the ends of association lines (e.g., 1, 0..*, 1..5).

Role Names:

- **Definition**: Descriptive names for the roles that classes play in associations, helping to clarify the purpose of the association.
- **Notation**: Names placed near the ends of association lines.

Example: In an association between Author and Book, multiplicity might show that an author can write multiple books (e.g., 1..*), and a book can have multiple authors (e.g., 0..*). Role names like writes and authoredBy clarify the association.

Use Case: Use multiplicity to define the number of allowed relationships and role names to describe the nature of these relationships.

7. Interfaces and Abstract Classes

Interfaces:

- **Definition**: Define a contract that classes must adhere to. Interfaces specify methods without implementing them.

- **Notation**: Represented as a rectangle with the <<interface>> stereotype or as a circle with the interface name.

Abstract Classes:

- **Definition**: Classes that cannot be instantiated and are used to provide common base functionality for subclasses. They can have abstract methods (without implementation) and concrete methods (with implementation).
- **Notation**: Represented as a class with the <<abstract>> stereotype or italicized method names.

Example: An interface Shape with methods draw() and resize(), and an abstract class Polygon with some concrete methods and abstract methods like calculateArea().

Use Case: Use interfaces to define common behavior across disparate classes and abstract classes to provide shared functionality and a base for other classes.

8. Nested Classes

Definition:

- Classes defined within the scope of another class. They can be used to group related classes and manage complexity.

Notation:

- Represented as a class within the boundaries of another class.

Example: A Bank class with a nested Account class that models different types of accounts.

Use Case: Use nested classes to logically group related classes and encapsulate them within the context of another class.

9. Class Stereotypes

Definition:

- Stereotypes provide additional semantics to UML elements. They extend the basic UML model with domain-specific concepts.

Notation:

- Represented with <<stereotype>> above the class name.

Example: A DataAccessObject class might be stereotyped as <<DAO>> to indicate its role in data access.

Use Case: Use stereotypes to add domain-specific meanings to classes, enhancing the expressiveness of your UML model.

10. Dynamic Class Modeling

Dynamic Class Modeling:

- Refers to capturing how classes change and interact over time, including their state changes and interactions during runtime.

Notation:

- Includes dynamic diagrams such as sequence and state diagrams linked with class diagrams.

Example: A User class with methods that change state based on user actions and interactions, illustrated through state diagrams and sequence diagrams.

Use Case: Use dynamic class modeling to represent and understand how classes interact and evolve during system operation.

Advanced Relationships

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

Dependency

- A dependency is a using relationship, specifying that a change in the specification of one thing may affect another thing that uses it, but not necessarily the reverse. Graphically, a dependency is rendered as a dashed line
- First, there are eight stereotypes that apply to dependency relationships among classes and objects in class diagrams.

1	**bind**	Specifies that the source instantiates the target template using the given actual parameters
2	**derive**	Specifies that the source may be computed from the target
3	**friend**	Specifies that the source is given special visibility into the target
4	**instanceOf**	Specifies that the source object is an instance of the target classifier
5	**instantiate**	Specifies that the source creates instances of the target
6	**powertype**	Specifies that the target is a powertype of the source; a powertype is a classifier whose objects are all the children of a given parent
7	**refine**	Specifies that the source is at a finer degree of abstraction than the target
8	**use**	Specifies that the semantics of the source element depends on the semantics of the public part of the target

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

9	**access**	Specifies that the source package is granted the right to reference the elements of the target package
10	**import**	A kind of access that specifies that the public contents of the target package enter the flat namespace of the source, as if they had been declared in the source

* Two stereotypes apply to dependency relationships among use cases:

11	**extend**	Specifies that the target use case extends the behavior of the source
12	**include**	Specifies that the source use case explicitly incorporates the behavior of another use case at a location specified by the source

* There are three stereotypes when modeling interactions among objects.

13	**become**	Specifies that the target is the same object as the source but at a later point in time and with possibly different values, state, or roles
14	**call**	Specifies that the source operation invokes the target operation
15	**copy**	Specifies that the target object is an exact, but independent, copy of the source

* One stereotype you'll encounter in the context of state machines is

16	**send**	Specifies that the source operation sends the target event

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

17	**trace**	Specifies that the target is an historical ancestor of the source

* We'll use trace when you want to model the relationships among elements in different models

Generalization

A generalization is a relationship between a general thing (called the superclass or parent) and a more specific kind of that thing(called the subclass or child).

In a generalization relationship, instances of the child may be used anywhere instances of the parent apply—meaning that the child is substitutable for the parent.

A plain, unadorned generalization relationship is sufficient for most of the inheritance relationships you'll encounter. However, if you want to specify a shade of meaning,

The UML defines one stereotype and four constraints that may be applied to generalization relationships.

Association

An association is a structural relationship, specifying that objects of one thing are connected to objects of another.

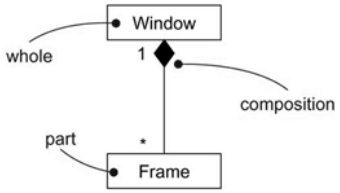
We use associations when you want to show structural relationships.

There are four basic adornments that apply to an association: a name, the role at each end of the association, the multiplicity at each end of the association, and aggregation.

Composition

* Simple aggregation is entirely conceptual and does nothing more than distinguish a "whole" from a "part."

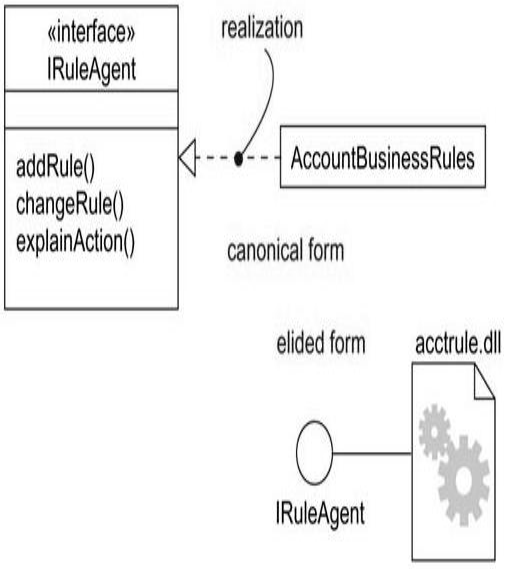
* Composition is a form of aggregation, with strong ownership and coincident lifetime as



Composition

Realization

1. Realization is sufficiently different from dependency, generalization, and association relationships that it is treated as a separate kind of relationship.
2. A realizationis a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out.
3. Graphically, a realization is rendered as a dashed directed line with a large open arrowhead pointing to the classifier that specifies the contract.
4. You'll use realization in two circumstances: in the context of interfaces and in the context of collaborations
5. Most of the time, you'll use realization to specify the relationship between an interface and the class or component that provides an operation or service for it
6. You'll also use realization to specify the relationship between a use case and the collaboration that realizes that use case



Common Modeling Techniques

Modeling Webs of Relationships

1. When you model the vocabulary of a complex system, you may encounter dozens, if not hundreds or thousands, of classes, interfaces, components, nodes, and use cases.
2. Establishing a crisp boundary around each of these abstractions is hard

3. This requires you to form a balanced distribution of responsibilities in the system as a whole, with individual abstractions that are tightly cohesive and with relationships that are expressive, yet loosely coupled

Interfaces, type and roles

Interface

- An interface is a collection of operations that are used to specify a service of a class or a component

type

- A type is a stereotype of a class used to specify a domain of objects, together with the operations (but not the methods) applicable to the object.

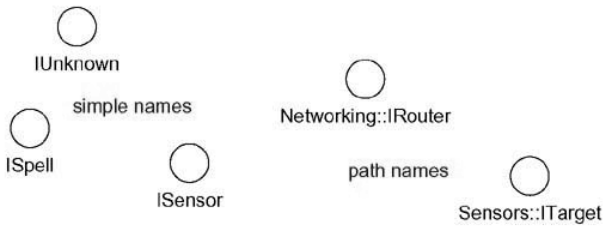
role

- A role is the behavior of an entity participating in a particular context.

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

Names

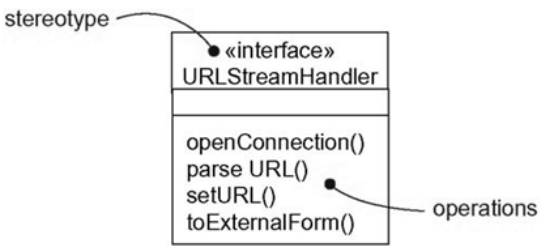
- Every interface must have a name that distinguishes it from other interfaces.
- A name is a textual string. That name alone is known as a simple name;
- A path name is the interface name prefixed by the name of the package



Simple and Path Names

Operations

- An interface is a named collection of operations used to specify a service of a class or of a component.
- Unlike classes or types, interfaces do not specify any structure (so they may not include any attributes), nor do they specify any implementation
- These operations may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.
- you can render an interface as a stereotyped class, listing its operations in the appropriate compartment. Operations may be drawn showing only their name, or they may be augmented to show their full signature and other properties



Operations

Realizations

- In the UML, you can supply much more information to an interface in order to make it understandable and approachable.

- First, you may attach pre- and postconditions to each operation and invariants to the class or component as a whole. By doing this, a client who needs to use an interface will be able to understand what the interface does and how to use it, without having to dive into an implementation.

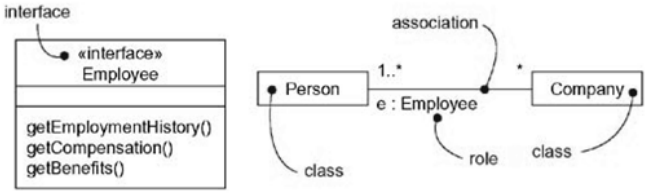
Types and Roles

A role names a behavior of an entity participating in a particular context. Stated another way, a role is the face that an abstraction presents to the world.

For example, consider an instance of the class Person. Depending on the context, that Person instance may play the role of Mother, Comforter, PayerOfBills, Employee, Customer, Manager, Pilot, Singer, and so on. When an object plays a particular role, it presents a face to the world, and clients that interact with it expect a certain behavior depending on the role that it plays at the time.

an instance of Person in the role of Manager would present a different set of properties than if the instance were playing the role of Mother.

In the UML, you can specify a role an abstraction presents to another abstraction by adorning the name of an association end with a specific interface.

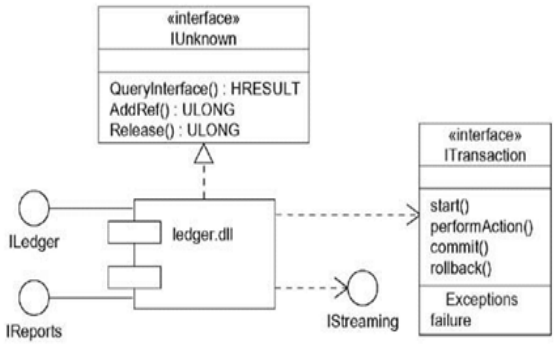


Roles

Common Modeling Techniques

➢ Modeling the Seams in a Systemeling the Seams in a System

- The most common purpose for which you'll use interfaces is to model the seams in a system composed of software components, such as COM+ or Java Beans.
- Identifying the seams in a system involves identifying clear lines of demarcation in your architecture. On either side of those lines, you'll find components that may change independently, without affecting the components on the other side,

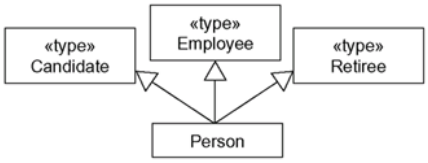


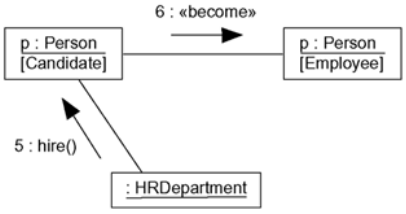
Modeling the Seams in a System

- The above Figure shows the seams surrounding a component (the library ledger.dll) drawn from a financial system. This component realizes three interfaces: IUnknown, ILedger, and IReports. In this diagram, IUnknown is shown in its expanded form; the other two are shown in their simple form, as lollipops. These three interfaces are realized by ledger.dll and are exported to other components for them to build on.
- As this diagram also shows, ledger.dll imports two interfaces, IStreaming and ITransaction, the latter of which is shown in its expanded form. These two interfaces are required by the ledger.dll component for its proper operation. Therefore, in a running system, you must supply components that realize these two interfaces.
- By identifying interfaces such as ITransaction, you've effectively decoupled the components on either side of the interface, permitting you to employ any component that conforms to that interface.

➢ Modeling Static and Dynamic Typeseling Static and Dynamic Types

- Most object-oriented programming languages are statically typed, which means that the type of an object is bound at the time the object is created.
- Even so, that object will likely play different roles over time.
- Modeling the static nature of an object can be visualized in a class diagram. However, when you are modeling things like business objects, which naturally change their roles throughout a workflow,





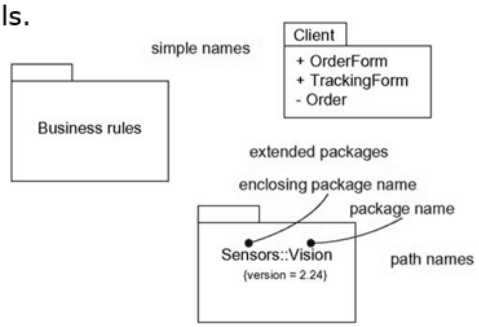
Modeling Static Types **Modeling Dynamic Types**

Package

"A package is a general-purpose mechanism for organizing elements into groups." Graphically, a package is rendered as a tabbed folder.

Names

- Every package must have a name that distinguishes it from other packages. A name is a textual string.
- That name alone is known as a simple name; a path name is the package name prefixed by the name of the package in which that package lives
- We may draw packages adorned with tagged values or with additional compartments to expose their details.



Common Modeling Techniques

Modeling Groups of Elements

Scan the modeling elements in a particular architectural view and look for clumps defined by elements that are conceptually or semantically close to one another.

Surround each of these clumps in a package.

For each package, distinguish which elements should be accessible outside the package. Mark them public, and all others protected or private. When in doubt, hide the element.

Explicitly connect packages that build on others via import dependencies

In the case of families of packages, connect specialized packages to their more general part via generalizations

Modeling Architectural Views

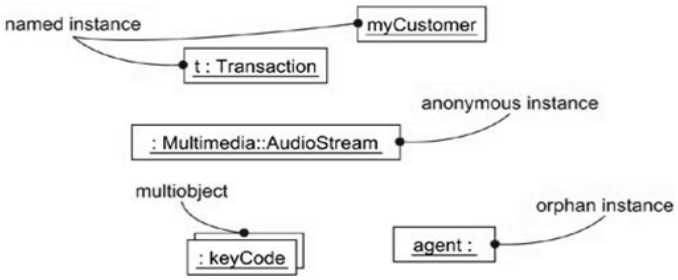
- We can use packages to model the views of an architecture.
- Remember that a view is a projection into the organization and structure of a system, focused on a particular aspect of that system.
- This definition has two implications. First, you can decompose a system into almost orthogonal packages, each of which addresses a set of architecturally significant decisions.(design view, a process view, an implementation view, a deployment view, and a use case view)
- Second, these packages own all the abstractions germane to that view.(Implementation view)

Instances

- An instance is a concrete manifestation of an abstraction to which a set of operations can be applied and which has a state that stores the effects of the operations.
- Graphically, an instance is rendered by underlining its name.
-

Abstractions and Instances

- Most instances you'll model with the UML will be instances of classes although you can have instances of other things, such as components, nodes, use cases, and associations



Class Diagrams

Terms and Concepts:

1. **Class:** A blueprint for creating objects, defining their properties (attributes) and behaviors (methods).
2. **Object:** An instance of a class. It represents a concrete occurrence of a class in the system.
3. **Attributes:** The data stored in an object, typically represented as fields in a class.
4. **Methods:** Functions or procedures defined within a class that describe the behavior of the objects.

5.Relationships:

1. **Association:** A relationship between classes that shows how objects of one class are connected to objects of another class.
2. **Aggregation:** A special form of association indicating a "whole-part" relationship, where a class is a part of another class but can exist independently.
3. **Composition:** A stronger form of aggregation where the part cannot exist independently of the whole.
4. **Inheritance:** A relationship where one class (subclass) inherits attributes and methods from another class (superclass).
5. **Dependency:** A relationship where one class depends on another to function, usually indicated by a dashed arrow.

Modeling Techniques:

1. **Modeling Simple Collaboration:**
    - **Collaboration Diagram:** Shows the interactions between objects in a system and their relationships. It highlights how objects collaborate to achieve a specific goal.
    - **Sequence Diagram:** Often used alongside class diagrams to detail the sequence of messages exchanged between objects.
2. **Logical Database Schema:**
    - **Entity-Relationship Diagram (ERD):** Represents data entities, their attributes, and the relationships between them. Used for designing database schemas.
    - **Class Diagram vs. ERD:** While class diagrams are more about object-oriented design, ERDs focus on database structure. However, both diagrams help in structuring data and relationships.
3. **Forward and Reverse Engineering:**
    - **Forward Engineering:** The process of creating code from models or diagrams. For example, generating class code from a class diagram.
    - **Reverse Engineering:** The process of creating models or diagrams from existing code. It helps in understanding and documenting existing systems.

Object Diagrams

Introduction:

- **Object Diagram:** A snapshot of the instances of the classes at a particular point in time. It represents the state of the system's objects and their relationships at a specific moment.
- **Purpose:** Useful for illustrating the static structure of a system or showing how objects interact at a given point in time.