Software Engineering and Project Management

MVJ22CS51

MODULE 2

INTRODUCTION, MODELLING CONCEPTS AND CLASS MODELLING

WHAT IS OBJECT ORIENTATION?

Object-Oriented (OO) means that we organize software as a collection of discrete objects that incorporate both data structure and behavior.

They include four aspects: ***identity***, ***classification***, ***inheritance***, and ***polymorphism***.

1. Identity

Identity means that data is quantized into discrete, distinguishable entities called objects. Eg. Bicycle, monitor, Binary Tree, a symbol table.

Objects can be concrete, such as a file in a file system, or conceptual, such as a scheduling policy in a multiprocessing operating system.

Each object has its own inherent identity.

In the real world an object simply exists, but within a programming language each object has a unique handle by which it can be referenced.

2. ***Classification***

Classification means that objects with the same data structure (attributes) and behavior (operations) are grouped into a class. Eg. Paragraph, Monitor, and ChessPiece.

A class is an abstraction that describes properties important to an application and ignores the rest.

Each class describes a possibly infinite set of individual objects.

Each object is said to be an instance of its class.

An object contains an implicit reference to its own class.

3. ***Inheritance***

Inheritance is the sharing of attributes and operations (features) among classes based on a hierarchical relationship.

A superclass has general information that subclasses refine and elaborate.

Each subclass incorporates, or inherits, all the features of its superclass and adds its own unique features. E.g. ScrollingWindow and FixedWindow are subclasses of Window.

4. ***Polymorphism***

Polymorphism means that the same operation may behave differently for different classes. Eg. The move operation, behaves differently for a pawn than for the queen in a chess game.

An operation is a procedure or transformation that an object performs or is subject to. E.g. operations such as RightJustify, display, and move.

An implementation of an operation by a specific class is called a **method**.

Because an OO operator is polymorphic, it may have more than one method implementing it, each for a different class of object.

****

WHAT IS OO DEVELOPMENT?

OO development as a way of thinking about software based on abstractions that exist in the real world as well as in the program.

Development refers to the software life cycle: **analysis**, **design**, and **implementation**.

The essence of OO development is the identification and organization of application concepts, rather than their final representation in a programming language.

1. *Modeling Concepts, Not Implementation*

OO community focused on programming languages, with the literature emphasizing implementation rather than analysis and design.

An OO development approach encourages software developers to work and think in terms of the application throughout the software life cycle.

OO development is a conceptual process independent of a programming language until the final stages.

OO development is fundamentally a way of thinking and not a programming technique. Its greatest benefits come from helping developers, and customers express abstract concepts clearly and communicate them to each other.

It can serve as a medium for specification, analysis, documentation, and interfacing, as well as for programming.

2. *OO Methodology*

We use a process for OO development and a graphical notation for representing OO concepts.

The process consists of building a model of an application and then adding details to it during design. The same seamless notation is used from analysis to design to implementation, so that information added in one stage of development need not be lost or translated for the next stage.

The methodology has the following stages:

1. *System conception*. Software development begins with business analysts or users conceiving an application and formulating tentative requirements.
2. *Analysis*. The analyst scrutinizes and rigorously restates the requirements from system conception by constructing models. The analyst must work with the requestor to understand the problem, because problem statements are rarely complete or correct. The analysis model is a concise, precise abstraction of what the desired system must do, not how it will be done. The analysis model should not contain implementation decisions. E.g., a Window class in a workstation windowing system would be described in terms of its visible attributes and operations.

   The analysis model has two parts:
   1. Domain model - description of the real-world objects reflected within the system
   2. Application model- a description of the parts of the application system itself that are visible to the user.
3. *System design:*

   The development team devise a high-level strategy—the system architecture—for solving the application problem.

   The system designer must decide what performance characteristics to optimize, choose a strategy of attacking the problem, and make tentative resource allocations.

4. *Class design:*
   The class designer adds details to the analysis model in accordance with the system design strategy.
   The focus of class design is the data structures and algorithms needed to implement each class. For example, the class designer now determines data structures and algorithms for each of the operations of the Window class.
5. *Implementation:*
   Implementers translate the classes and relationships developed during class design into a particular programming language, database, or hardware.
   During implementation, it is important to follow good software engineering practice so that traceability to the design is apparent and so that the system remains flexible and extensible.

3. *Three Models*

A complete description of a system requires models from all three viewpoints.

1. *Class Model:*
   The ***class model*** describes the static structure of the objects in a system and their relationships.
   The class model defines the context for software development—the universe of discourse.
   The class model contains class diagrams.
   A ***class diagram*** is a graph whose nodes are classes and whose arcs are relationships among classes.
2. *State Model:*
   The **state model** describes the aspects of an object that change over time.
   The state model specifies and implements control with state diagrams.
   A **state diagram** is a graph whose
   nodes are states and whose arcs are transitions between states caused by events.
3. *Interaction Model:*
   The interaction model describes how the objects in a system cooperate to achieve broader results.
   The interaction model starts with use cases that are then elaborated with sequence and activity diagrams.
   A **use case** focuses on the functionality of a system—that is, what a system does for users.
   A **sequence** diagram shows the objects that interact and the time sequence of their interactions.
   An **activity** diagram elaborates important processing steps.

****

OO THEMES

1. *Abstraction*

Abstraction focus on essential aspects of an application while ignoring details.

This means focusing on what an object is and does, before deciding how to implement it.

Use of abstraction preserves the freedom to make decisions as long as possible by avoiding premature commitments to details. Most modern languages provide data abstraction, but inheritance and polymorphism add power.

2. *Encapsulation*

Encapsulation (also information hiding) separates the external aspects of an object, that are accessible to other objects, from the internal implementation details, that are hidden from other objects.

Encapsulation prevents portions of a program from becoming so interdependent that a small change has massive ripple effects.

You can change an object's implementation without affecting the applications that use it.

Encapsulation is not unique to OO languages, but the ability to combine data structure and behavior in a single entity.

3. *Combining Data and Behavior*

The caller of an operation need not consider how many implementations exist.

Operator polymorphism shifts the burden of deciding what implementation to use from the calling code to the class hierarchy.

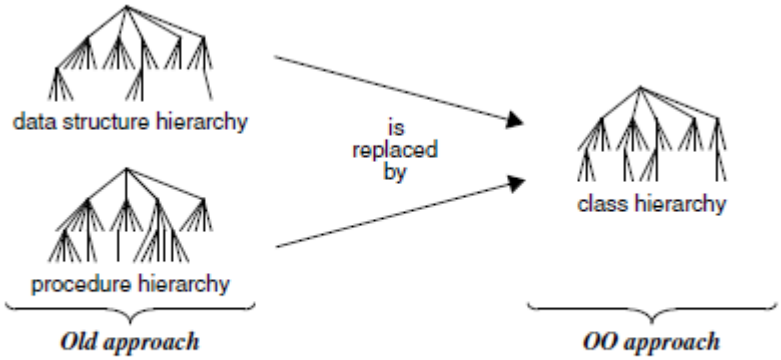


Figure 1.3 OO vs. prior approach. An OO approach has one unified hierarchy for both data and behavior.

4. *Sharing*

Inheritance of both data structure and behavior lets subclasses share common code.

OO development not only lets you share information within an application, but also offers the prospect of reusing designs and code on future projects.

OO development provides the tools, such as abstraction, encapsulation, and inheritance, to build libraries of reusable components.

5. *Emphasis on the Essence of an Object*

OO technology stresses what an object is, rather than how it is used.

The uses of an object depend on the details of the application and often change during development.

OO development places a greater emphasis on data structure and a lesser emphasis on procedure structure than functional-decomposition methodologies.

6. *Synergy*

Identity, classification, polymorphism, and inheritance characterize OO languages.

Each of these concepts can be used in isolation, but together they complement each other synergistically.

****

EVIDENCE FOR USEFULNESS OF OO DEVELOPMENT

We used OO techniques for developing compilers, graphics, user interfaces, databases, an OO language, CAD systems, simulations, metamodels, control systems, and other applications. We used OO models to document programs that are ill-structured and difficult to understand. Our implementation targets ranged from OO languages to non-OO languages to databases.

The annual OOPSLA (Object-Oriented Programming Systems, Languages, and Applications), ECOOP (European Conference on Object-Oriented Programming), and TOOLS (Technology of Object-Oriented Languages and Systems) conferences are important forums for disseminating new OO ideas and application results.

****

OO MODELLING HISTORY

Our work at GE R&D led to the development of the Object Modeling Technique (OMT), which the previous edition of this book introduced in 1991.

In 1994 Jim Rumbaugh joined Rational (now part of IBM) and began working with Grady Booch on unifying the OMT and Booch notations.

In 1995, Ivar Jacobson also joined Rational and added Objectory to the unification work.

In 1996 the Object Management Group (OMG) issued a request for proposals for a standard OO modeling notation. Several companies responded, and eventually the competing proposals were coalesced into a final proposal.

The OMG unanimously accepted the resulting Unified Modeling Language (UML) as a standard in November 1997.

In 2001 OMG members started work on a revision to add features missing from the initial specification and to fix problems that were discovered by experience with UML

UML 2.0 revision approved in 2004

****

MODELLING AS DESIGN TECHNIQUE: MODELLING, ABSTRACTION, THE THREE MODELS

***1*. Modeling**

Designers build many kinds of models for various purposes before constructing things.

E.g. architectural models to show customers, airplane scale models for wind-tunnel tests, pencil sketches for composition of oil paintings, blueprints of machine parts, storyboards of advertisements, and outlines of books.

1. *Testing a physical entity before building it*.

The medieval masons did not know modern physics, but they built scale models of the Gothic cathedrals to test the forces on the structure.

Now, Engineers test scale models of airplanes, cars, and boats in wind tunnels and water tanks to improve their dynamics.

Recent advances in computation permit the simulation of many physical structures without the need to build physical models.

Not only is simulation cheaper, but it provides information that is too fleeting or inaccessible to be measured from a physical model.

2. *Communication with customers*.

Architects and product designers build models to show their customers.

Mock-ups are demonstration products that imitate some or all of the external behavior of a system.

3. *Visualization*.

Storyboards of movies, television shows, and advertisements let writers see how their ideas flow.

They can modify awkward transitions, dangling ends, and unnecessary segments before detailed writing begins.

Artists' sketches let them block out their ideas and make changes before committing them to oil or stone.

4. *Reduction of complexity.*

Perhaps the main reason for modeling, which incorporates all the previous reasons, is to deal with systems that are too complex to understand directly.

Models reduce complexity by separating out a small number of important things to deal with at a time.

2. **Abstraction**

The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant.

Abstraction must always be for some purpose, because the purpose determines what is, and is not, important.

Many different abstractions of the same thing are possible, depending on the purpose for which they are made.

All abstractions are incomplete and inaccurate.

All human words and language are abstractions—incomplete descriptions of the real world. This does not destroy their usefulness.

The purpose of an abstraction is to limit the universe so we can understand.

There is no single "correct" model of a situation, only adequate and inadequate ones.

3. **The Three Models**

We model a system from three related but different viewpoints:

1. The **class** model represents the static, structural, "data" aspects of a system.
2. The **state** model represents the temporal, behavioral, "control" aspects of a system.
3. The **interaction** model represents the collaboration of individual objects, the "interaction" aspects of a system.

A typical software procedure incorporates all three aspects:

- It uses data structures (class model)
- It sequences operations in time (state model)
- It passes data and control among objects (interaction model).

The different models are not completely independent—a system is more than a collection of independent parts—but each model can be examined and understood by itself to a large extent.

1. *Class Model*

The class model describes the structure of objects in a system—their identity, their relationships to other objects, their attributes, and their operations.

The class model provides context for the state and interaction models.

Our goal in constructing a class model is to capture those concepts from the real world that are important to an application.

In modeling an engineering problem, the class model should contain terms familiar to engineers; in modeling a business problem, terms from the business; in modeling a user interface, terms from the application.

Class diagrams express the class model.

Generalization lets classes share structure and behavior, and associations relate the classes.

2. *State Model*

The state model describes those aspects of objects concerned with time and the sequencing of operations—events that mark changes, states that define the context for events, and the organization of events and states.

The state model captures control, the aspect of a system that describes the sequences of operations that occur, without regard for what the operations do, what they operate on, or how they are implemented.

State diagrams express the state model.

Each state diagram shows the state and event sequences permitted in a system for one class of objects.

State diagrams refer to the other models.

Actions and events in a state diagram become operations on objects in the class model.

References between state diagrams become interactions in the interaction model.

3. *Interaction Model*

The interaction model describes interactions between objects—how individual objects collaborate to achieve the behavior of the system as a whole.

The state and interaction models describe different aspects of behavior, and you need both to describe behavior fully.

Use cases, sequence diagrams, and activity diagrams document the interaction model.

- Use cases diagrams document major themes for interaction between the system and outside actors.
- Sequence diagrams show the objects that interact and the time sequence of their interactions.
- Activity diagrams show the flow of control among the processing steps of a computation.

****

CLASS MODELLING:

OBJECT AND CLASS CONCEPT

1. **Objects**

The purpose of class modeling is to describe objects.

An object is a concept, abstraction, or thing with identity that has meaning for an application.

Objects often appear as proper nouns or specific references in problem descriptions and discussions with users.

The choice of objects depends on judgment and the nature of a problem.

All objects have identity and are distinguishable. E.g. Two apples with the same color, shape, and texture are still individual apples; a person can eat one and then eat the other. Similarly, identical twins are two distinct persons, even though they may look the same.

The term identity means that objects are distinguished by their inherent existence and not by descriptive properties that they may have.

2. **Classes**

An object is an instance—or occurrence—of a class.

A class describes a group of objects with the same properties (attributes), behavior (operations), kinds of relationships, and semantics.

Person, company, process, and window are all classes.

Each person has name and birthdate and may work at a job. Each process has an owner, priority, and list of required resources.

Classes often appear as common nouns and noun phrases in problem descriptions and discussions with users.

Objects in a class have the same attributes and forms of behavior.

The choice of classes depends on the nature and scope of an application and is a matter of judgment.

The objects in a class share a common semantic purpose, above and beyond the requirement of common attributes and behavior.

E.g. a barn and a horse may both have a cost and an age. If barn and horse were regarded as purely financial assets, they could belong to the same class. If the developer took into consideration that a person paints a barn and feeds a horse, they would be modeled as distinct classes.

Each object "knows" its class.

An object's class is an implicit property of the object.

3. **Class Diagrams**

We need a means for expressing models that is coherent, precise, and easy to formulate.

There are two kinds of models of structure—class diagrams and object diagrams.

Class diagrams provide a graphic notation for modeling classes and their relationships, thereby describing possible objects.

Class diagrams are useful both for abstract modeling and for designing actual programs. They are concise, easy to understand, and work well in practice.

Object diagrams are helpful for documenting test cases and discussing examples.

A class diagram corresponds to an infinite set of object diagrams.

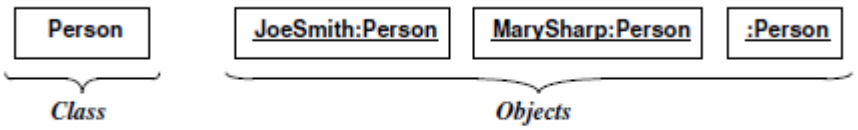


Figure 3.1 A class and objects. Objects and classes are the focus of class modeling.

Steps to represent a class diagram:

- The UML symbol for an object is a box with an object name followed by a colon and the class name.
- The object name and class name are both underlined. Our convention is to list the object name and class name in boldface.
- The UML symbol for a class also is a box. Our convention is to list the class name in boldface, center the name in the box, and capitalize the first letter.
- We use singular nouns for the names of classes.

- Note how we run together multiword names, such as “JoeSmith”, separating the words with intervening capital letters. Alternative conventions would be to use intervening spaces (Joe Smith) or underscores (Joe_Smith).

4. **Values and Attributes**

A **value** is a piece of data.

An **attribute** is a named property of a class that describes a value held by each object of the class.

E.g. Name, birthdate, and weight are attributes of Person objects.

Color, modelYear, and weight are attributes of Car objects.

Each attribute has a value for each object. E.g. attribute birthdate has value “21 October 1983” for object JoeSmith.

Different objects may have the same or different values for a given attribute.

Each attribute name is unique within a class (as opposed to being unique across all classes). Thus class Person and class Car may each have an attribute called weight.

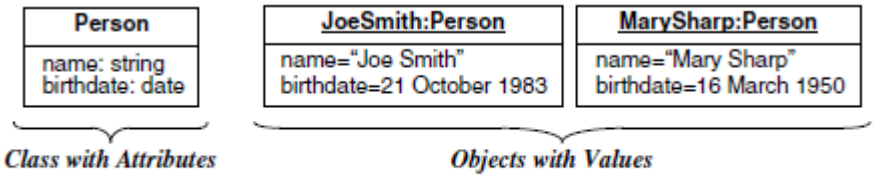


Figure 3.2 Attributes and values. Attributes elaborate classes.

The UML notation lists attributes in the second compartment of the class box.

Optional details, such as type and default value, may follow each attribute. A colon precedes the type.

An equal sign precedes the default value.

Our convention is to show the attribute name in regular face, left align the name in the box, and use a lowercase letter for the first letter.

5. **Operations and Methods**

An operation is a function or procedure that may be applied to or by objects in a class.

Hire,fire, and payDividend are operations on class Company. Open, close, hide, and redisplay are operations on class Window.

All objects in a class share the same operations.

The same operation may apply to many different classes. Such an operation is polymorphic; that is, the same operation takes on different forms in different classes.

A method is the implementation of an operation for a class. For example, the class File may have an operation print.

An operation may have arguments in addition to its target object.

When an operation has methods on several classes, it is important that the methods all have the same signature—the number and types of arguments and the type of result value. E.g., print should not have fileName as an argument for one method and filePointer for another.

The class Person has attributes name and birthdate and operations change- Job and changeAddress.

Name, birthdate, changeJob, and changeAddress are features of Person.

Feature is a generic word for either an attribute or operation.

Similarly, File has a print operation. GeometricObject has move, select, and rotate operations. Move has argument delta, which is a Vector; select has one argument p, which is of type Point and returns a Boolean; and rotate has argument angle, which is an input of type float with a default value of 0.0.

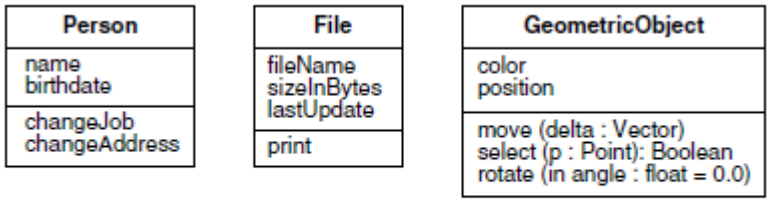


Figure 3.4 Operations. An operation is a function or procedure that may be applied to or by objects in a class.

****

LINK AND ASSOCIATIONS CONCEPTS

1. **Links and Associations**

A **link** is a physical or conceptual connection among objects.

A link is an instance of an association.

An **association** is a description of a group of links with common structure and common semantics. E.g., a person WorksFor a company. The links of an association connect objects from the same classes.

An association describes a set of potential links in the same way that a class describes a set of potential objects.

Links and associations often appear as verbs in problem statements.

The below figure depicts a model for a financial application. Stock brokerage firms need to perform tasks such as recording ownership of various stocks, tracking dividends, alerting customers to changes in the market, and computing margin requirements. The top portion of the figure shows a class diagram and the bottom shows an object diagram.

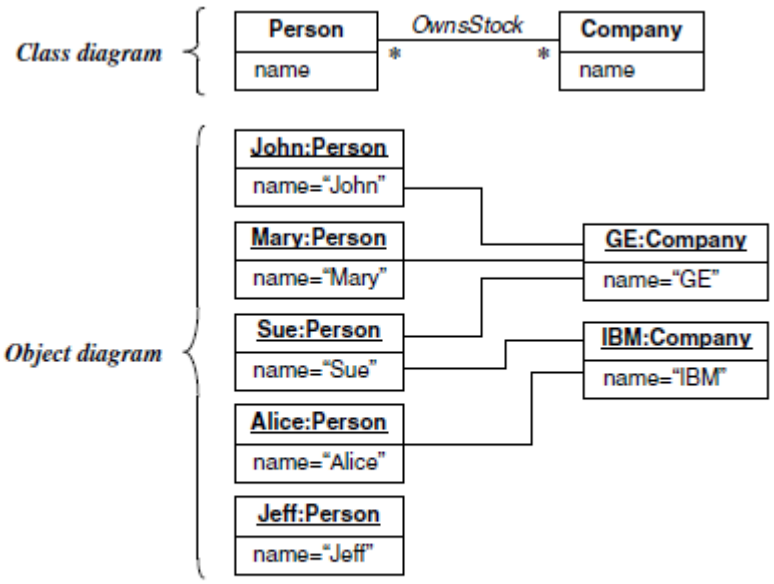


Figure 3.7 Many-to-many association. An association describes a set of potential links in the same way that a class describes a set of potential objects.

A **reference** is an attribute in one object that refers to another object.

E.g., a data structure for Person might contain an attribute employer that refers to a Company object, and a Company object might contain an attribute employees that refers to a set of Person objects.

2. **Multiplicity**

Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class.

Multiplicity constrains the number of related objects.

We describe multiplicity as being “one” or “many,” but more generally it is a (possibly infinite) subset of the nonnegative integers.

UML(Unified Modeling Language) diagrams explicitly list multiplicity at the ends of association lines.

The UML specifies multiplicity with an interval, such as “1” (exactly one), “1..*” (one or more), or “3..5” (three to five, inclusive).

The special symbol “*” is a shorthand notation that denotes “many” (zero or more).

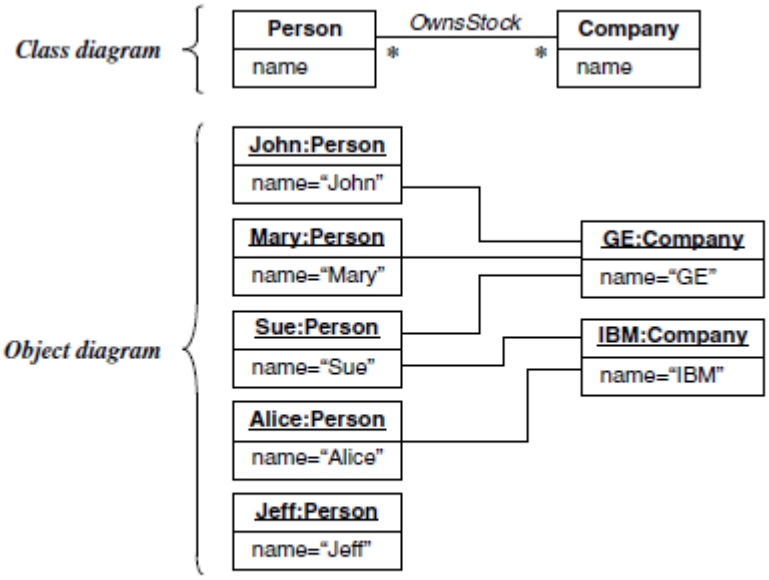


Figure 3.7 Many-to-many association. An association describes a set of potential links in the same way that a class describes a set of potential objects.

The above diagram illustrates many-to-many multiplicity. A person may own stock in many companies. A company may have multiple persons holding its stock.

In this particular case, John and Mary own stock in the GE company; Alice owns stock in the IBM company; Sue owns stock in both companies; Jeff does not own any stock. GE stock is owned by three persons; IBM stock is owned by two persons.

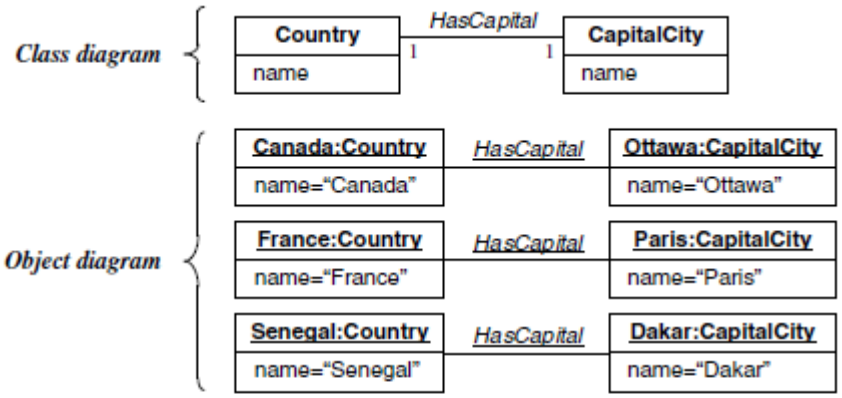


Figure 3.8 One-to-one association. Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class.

Figure 3.9 illustrates zero-or-one multiplicity. A workstation may have one of its windows designated as the console to receive general error messages. It is possible, however, that no console window exists. (The word "console" on the diagram is an association end name, discussed in Section 3.2.3.)

The above diagram shows one-to-one association and some corresponding links.

Each country has one capital city. A capital city administers one country. (In fact, some countries, such as The Netherlands and Switzerland, have more than one capital city for different purposes. If this fact were important, the model could be modified by changing the multiplicity or by providing a separate association for each kind of capital city.)

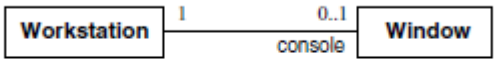


Figure 3.9 Zero-or-one multiplicity. It may be optional whether an object is involved in an association.

The above diagram illustrates zero-or-one multiplicity.

A workstation may have one of its windows designated as the console to receive general error messages. It is possible, however, that no console window exists. (The word "console" on the diagram is an association end name.)

Multiplicity is a constraint on the size of a collection; **cardinality** is the count of elements that are actually in a collection. Therefore, multiplicity is a constraint on the cardinality.

A multiplicity of "many" specifies that an object may be associated with multiple objects. However, for each association there is at most one link between a given pair of objects.

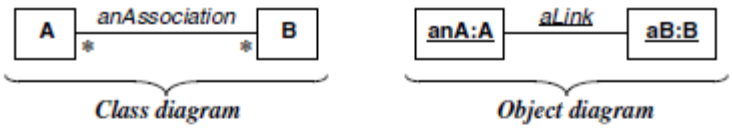


Figure 3.10 Association vs. link. A pair of objects can be instantiated at most once per association (except for bags and sequences).

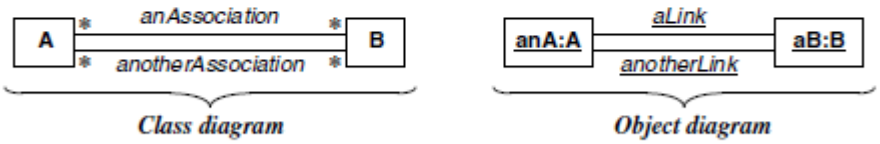


Figure 3.11 Association vs. link. You can use multiple associations to model multiple links between the same objects.

The above diagrams shows two links between the same objects, you must have two associations.

Multiplicity depends on assumptions and how you define the boundaries of a problem.

3. **Association End Names**

Multiplicity implicitly is referred to the ends of associations.

E.g., one-to-many association has two ends—an end with a multiplicity of "one" and an end with a multiplicity of "many."

You can not only assign a multiplicity to an association end, but you can give it a name as well. Association end names often appear as nouns in problem descriptions.

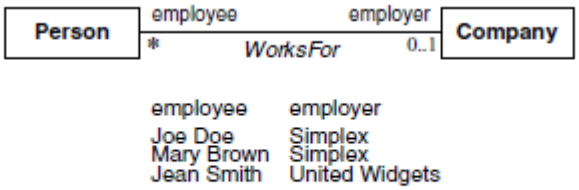


Figure 3.12 Association end names. Each end of an association can have a name.

The above diagram shows, a name appears next to the association end. In the figure Person and Company participate in association WorksFor. A person is an employee with respect to a company; a company is an employer with respect to a person.

Association end names are necessary for associations between two objects of the same class.

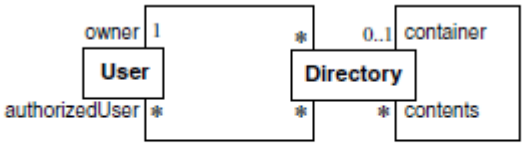


Figure 3.13 Association end names. Association end names are necessary for associations between two objects of the same class. They can also distinguish multiple associations between a pair of classes.

The above diagram represents a **container** and **contents** distinguish the two usages of Directory in the self-association. A directory may contain many lesser directories and may optionally be contained itself.

Each directory has exactly one user who is an owner and many users who are authorized to use the directory.

Association end names let you unify multiple references to the same class.

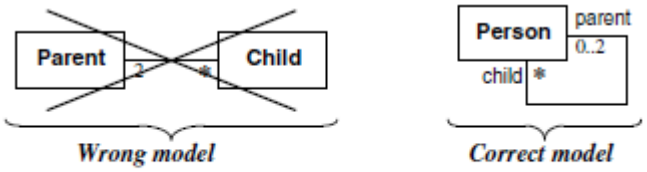


Figure 3.14 Association end names. Use association end names to model multiple references to the same class.

From the above diagram, when constructing class diagrams you should properly use association end names and not introduce a separate class for each reference.

In the wrong model, two instances represent a person with a child, one for the child and one for the parent.

In the correct model, one person instance participates in two or more links, twice as a parent and zero or more times as a child.

Because association end names distinguish objects, all names on the far end of associations attached to a class must be unique. Although the name appears next to the destination object on an association, it is really a pseudo attribute of the source class and must be unique within it.

4. **Ordering**

Often the objects on a "many" association end have no explicit order, and you can regard them as a set.

Sometimes, however, the objects have an explicit order.

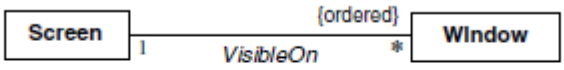


Figure 3.15 Ordering the objects for an association end. Ordering sometimes occurs for "many" multiplicity.

The above diagram shows a workstation screen containing a number of overlapping windows.

Each window on a screen occurs at most once. The windows have an explicit order, so only the topmost window is visible at any point on the screen.

The ordering is an inherent part of the association. You can indicate an ordered set of objects by writing "{ordered}" next to the appropriate association end.

5. **Bags and Sequences**

A binary association has at most one link for a pair of objects.

However, we can permit multiple links for a pair of objects by annotating an association end with {bag} or {sequence}.

A **bag** is a collection of elements with duplicates allowed.

A **sequence** is an ordered collection of elements with duplicates allowed.

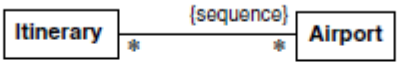


Figure 3.16 An example of a sequence. An itinerary may visit multiple airports, so you should use *{sequence}* and not *{ordered}*.

The above diagram shows an itinerary is a sequence of airports and the same airport can be visited more than once.

Like the {ordered} indication, {bag} and {sequence} are permitted only for binary associations.

Note that the {ordered} and the {sequence} annotations are the same, except that the first disallows duplicates and the other allows them.

A sequence association is an ordered bag, while an ordered association is an ordered set.

6. **Association Classes**

The representation of links of an association with attributes is known as association class.

An association class is an association that is also a class.

Like the links of an association, the instances of an association class derive identity from instances of the constituent classes.

Like a class, an association class can have attributes and operations and participate in associations.

You can find association classes by looking for adverbs in a problem statement or by abstracting known values.

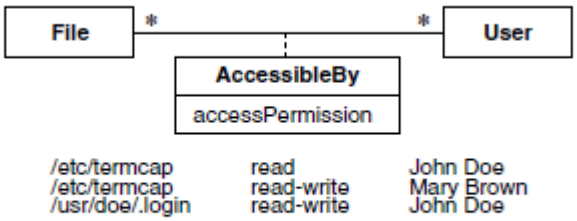


Figure 3.17 An association class. The links of an association can have attributes.

From the above figure, we say accessPermission is an attribute of AccessibleBy.

The sample data at the bottom of the figure shows the value for each link.

The UML notation for an association class is a box (a class box) attached to the association by a dashed line.

Many-to-many associations provide a compelling rationale for association classes. Attributes for such associations unmistakably belong to the link and cannot be ascribed to either object.

From the above figure, accessPermission is a joint property of File and User and cannot be attached to either File or User alone without losing information.

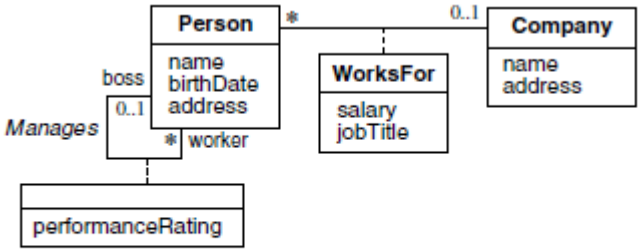


Figure 3.18 Association classes. Attributes may also occur for one-to-many and one-to-one associations.

The above figure presents attributes for two one-to-many associations. Each person working for a company receives a salary and has a job title. The boss evaluates the performance of each worker. Attributes may also occur for one-to-one associations.

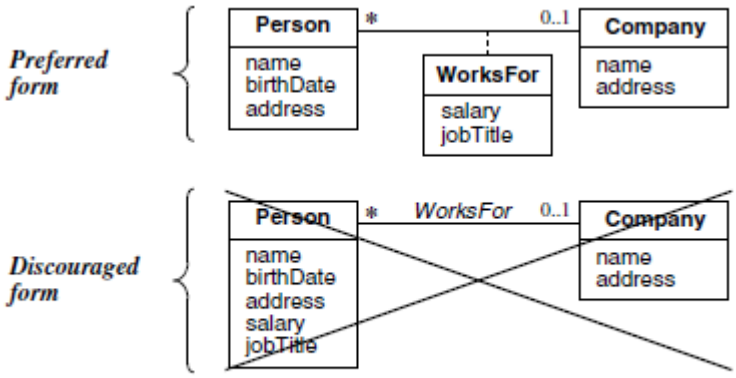


Figure 3.19 Proper use of association classes. Do not fold attributes of an association into a class.

The above figure shows how it is possible to fold attributes for one-to-one and one-to-many associations into the class opposite a “one” end. This is not possible for many-to-many associations.

As a rule, you should not fold such attributes into a class because the multiplicity of the association might change. Only the association class form remains correct if the multiplicity of WorksFor is changed to many-to-many.

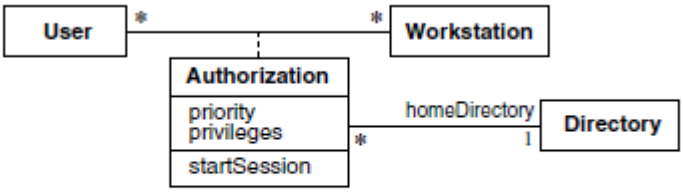


Figure 3.20 An association class participating in an association. Association classes let you specify identity and navigation paths precisely.

The above figure shows an association class participating in an association.

Users may be authorized on many workstations. Each authorization carries a priority and access privileges.

A user has a home directory for each authorized workstation, but several workstations and users can share the same home directory.

Association classes are an important aspect of class modeling because they let you specify identity and navigation paths precisely.

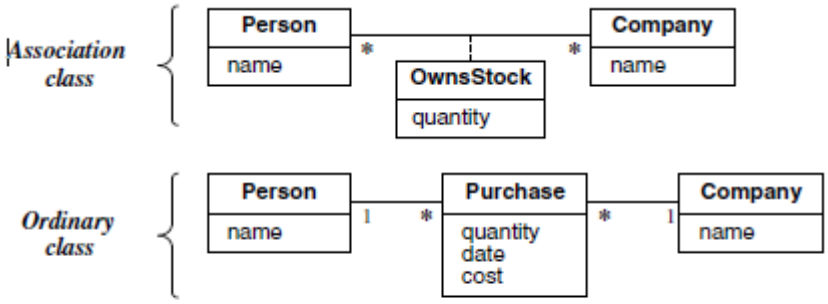


Figure 3.21 Association class vs. ordinary class. An association class is much different than an ordinary class.

The above figure shows the difference between association class and ordinary class.

The association class has only one occurrence for each pairing of Person and Company. In contrast there can be any number of occurrences of a Purchase for each Person and Company.

Each purchase is distinct and has its own quantity, date, and cost.

7. **<u>Qualified Associations</u>**

A qualified association is an association in which an attribute called the **qualifier** disambiguates the objects for a "many" association end.

It is possible to define qualifiers for one-to-many and many-to-many associations.

A qualifier selects among the target objects, reducing the effective multiplicity, from "many" to "one."

Qualified associations with a target multiplicity of "one" or "zero-or-one" specify a precise path for finding the target object from the source object.

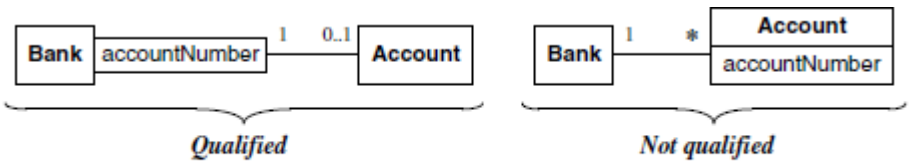


Figure 3.22 Qualified association. Qualification increases the precision of a model.

The above figure illustrates the most common use of a qualifier— for associations with one-to-many multiplicity.

A bank services multiple accounts. An account belongs to a single bank. Within the context of a bank, the account number specifies a unique account. Bank and Account are classes and accountNumber is the qualifier.

Qualification reduces the effective multiplicity of this association from one-to-many to one-to-one.

Both models are acceptable, but the qualified model adds information.

The qualified model adds a multiplicity constraint, that the combination of a bank and an account number yields at most one account.

The qualified model conveys the significance of account number in traversing the model, as methods will reflect.

The notation for a qualifier is a small box on the end of the association line near the source class.

The qualifier box may grow out of any side (top, bottom, left, right) of the source class.

The source class plus the qualifier yields the target class.

From the above figure, Bank + accountNumber yields an Account, therefore accountNumber is listed in a box contiguous to Bank.

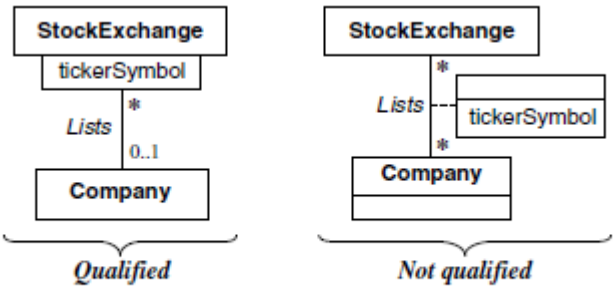


Figure 3.23 Qualified association. Qualification also facilitates traversal of class models.

The above figure is an example of qualification. A stock exchange lists many companies. However, a stock exchange lists only one company with a given ticker symbol.

A company may be listed on many stock exchanges, possibly under different symbols. (We are presuming this is true. If every stock had a single ticker symbol that was invariant across exchanges, we would make tickerSymbol an attribute of Company.)

****

GENERALIZATION AND INHERITANCE

Generalization is the relationship between a class (the **superclass**) and one or more variations of the class (the **subclasses**).

Generalization organizes classes by their similarities and differences, structuring the description of objects.

The superclass holds common attributes, operations, and associations; the subclasses add specific attributes, operations, and associations.

Each subclass is said to inherit the features of its superclass.

Generalization is sometimes called the “is-a” relationship, because each instance of a subclass is an instance of the superclass as well.

Simple generalization organizes classes into a hierarchy; each subclass has a single immediate superclass

There can be multiple levels of generalizations.

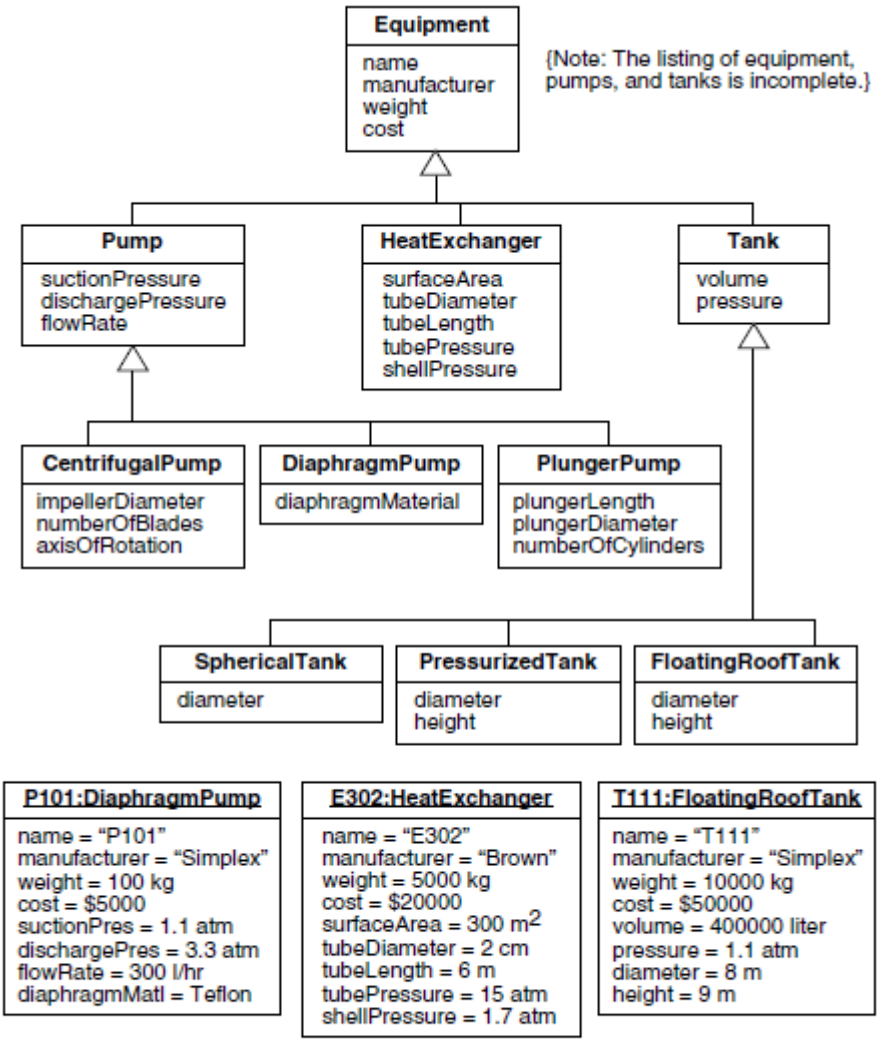


Figure 3.24 A multilevel inheritance hierarchy with instances. Generalization organizes classes by their similarities and differences, structuring the description of objects.

The above figure shows several examples of generalization for equipment.

Each piece of equipment is a pump, heat exchanger, or tank.

There are several kinds of pumps: centrifugal, diaphragm, and plunger.

There are several kinds of tanks: spherical, pressurized, and floating roof.

The fact that the tank generalization symbol is drawn below the pump generalization symbol is not significant.

Several objects are displayed at the bottom of the figure. Each object inherits features from one class at each level of the generalization. Thus P101 embodies the features of equipment, pump, and diaphragm pump. E302 has the properties of equipment and heat exchanger.

A large hollow arrowhead denotes generalization. The arrowhead points to the superclass.

You may directly connect the superclass to each subclass, but we normally prefer to group subclasses as a tree.

For convenience, you can rotate the triangle and place it on any side, but if possible you should draw the superclass on top and the subclasses on the bottom.

The curly braces denote a UML comment, indicating that there are additional subclasses that the diagram does not show.

Generalization is transitive across an arbitrary number of levels. The terms **ancestor** and **descendant** refer to generalization of classes across multiple levels.

An instance of a subclass is simultaneously an instance of all its ancestor classes.

An instance includes a value for every attribute of every ancestor class.

An instance can invoke any operation on any ancestor class.

Each subclass not only inherits all the features of its ancestors but adds its own specific features as well.

E.g., Pump adds attributes suctionPressure, dischargePressure, and flowRate, which other kinds of equipment do not share.

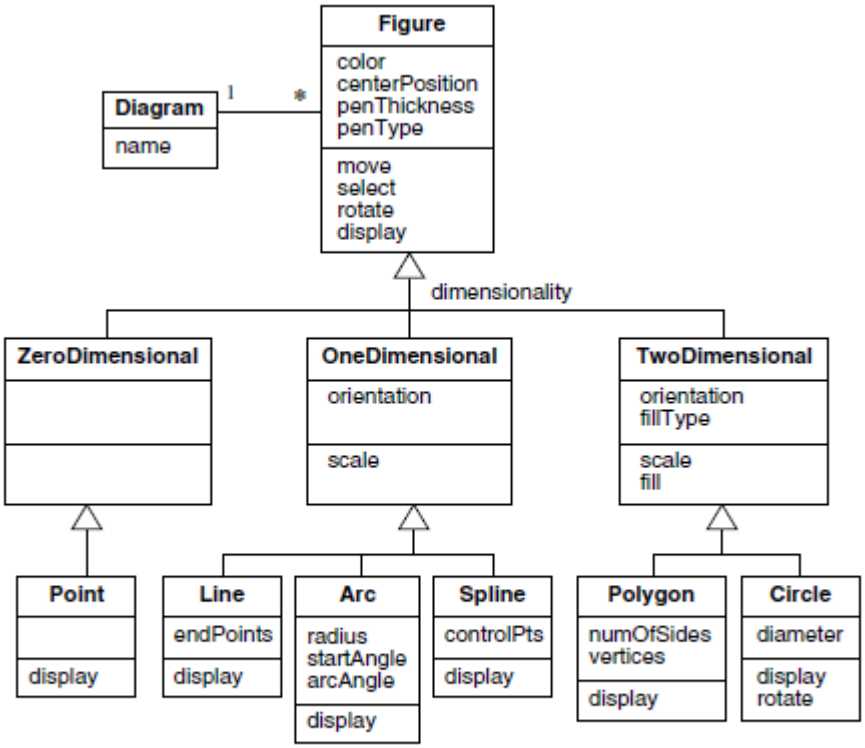


Figure 3.25 Inheritance for graphic figures. Each subclass inherits the attributes, operations, and associations of its superclasses.

The above figure shows classes of geometric figures. This example has more of a programming flavor and emphasizes inheritance of operations.

Move, select, rotate, and display are operations that all subclasses inherit.

Scale applies to one-dimensional and two-dimensional figures. **Fill** applies only to two-dimensional figures.

A **generalization set name** is an enumerated attribute that indicates which aspect of an object is being abstracted by a particular generalization.

You should generalize only one aspect at a time. For example, the means of propulsion (wind, fuel, animal, gravity) and the operating environment (land, air, water, outer space) are two aspects for class Vehicle.

Generalization set values is inherently in one-to-one correspondence with the subclasses of a generalization.

Do not nest subclasses too deeply. Deeply nested subclasses can be difficult to understand, much like deeply nested blocks of code in a procedural language. E.g. An inheritance hierarchy that is two or three levels deep is certainly acceptable; ten levels deep is probably excessive; five or six levels may or may not be proper.

Use of Generalization

Generalization has three purposes,

1. *To support for polymorphism*

We can call an operation at the superclass level, and the OO language compiler automatically resolves the call to the method that matches the calling object's class.

Polymorphism increases the flexibility of software—you add a new subclass and automatically inherit superclass behavior.

Furthermore, the new subclass does not disrupt existing code. Contrast the OO situation with procedural code, where addition of a new type can cause a ripple of changes.

2. *To structure the description of objects.*

   When you use generalization, you are making a conceptual statement—you are forming a taxonomy and organizing objects on the basis of their similarities and differences.

   This is much more profound than modeling each class individually and in isolation from other classes.

3. *To enable reuse of code*

   We can inherit code within your application as well as from past work (such as a class library).

   Reuse is more productive than repeatedly writing code from scratch.

   Generalization also lets you adjust the code, where necessary, to get the precise desired behavior. Reuse is an important motivator for inheritance, but the benefits are often oversold.

The terms generalization, specialization, and inheritance all refer to aspects of the same idea.

The word **generalization** derives from the fact that the superclass generalizes the subclasses.

Specialization refers to the fact that the subclasses refine or specialize the superclass.

Inheritance is the mechanism for sharing attributes, operations, and associations via the generalization/ specialization relationship.

Overriding Features

A subclass may **override** a superclass feature by defining a feature with the same name.

The overriding feature (the subclass feature) refines and replaces the overridden feature (the superclass feature).

There are several reasons why you may wish to override a feature: to specify behavior that depends on the subclass, to tighten the specification of a feature, or to improve performance.

You may override methods and default values of attributes.

You should never override the signature, or form, of a feature.

An override should preserve attribute type, number and type of arguments to an operation, and operation return type.

Tightening the type of an attribute or operation argument to be a subclass of the original type is a form of restriction and must be done with care.

It is common to boost performance by overriding a general method with a special method that takes advantage of specific information but does not alter the operation semantics.

You should never override a feature so that it is inconsistent with the original inherited feature. A subclass is a special case of its superclass and should be compatible with it in every respect.

A common, but unfortunate, practice in OO programming is to “borrow” a class that is similar to a desired class and then modify it by changing and ignoring some of its features, even though the new class is not really a special case of the original class. This practice can lead to conceptual confusion and hidden assumptions built into programs.

****

A SAMPLE CLASS MODEL

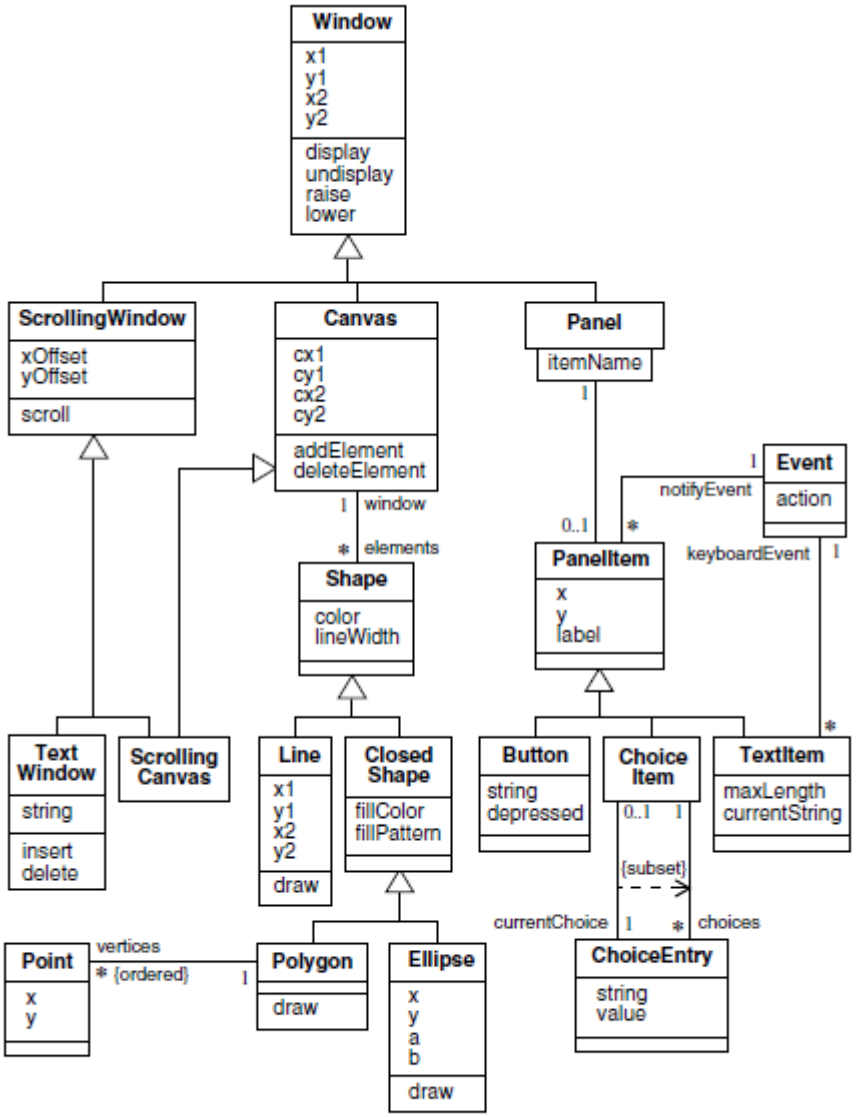


Figure 3.26 Class model of a windowing system

The above figure shows a class model of a **workstation window management system**.

Class **Window** defines common parameters of all kinds of windows, including a rectangular boundary defined by the attributes x1, y1, x2, y2, and operations to display and undisplay a window and to raise it to the top (foreground) or lower it to the bottom (background) of the entire set of windows.

A **canvas** is a region for drawing graphics. It inherits the window boundary from Window and adds the dimensions of the underlying canvas region defined by attributes cx1, cy1, cx2, cy2. Canvas windows have operations to add and delete elements.

A canvas contains a set of elements, shown by the association to class **Shape**. All shapes have color and line width. Shapes can be lines, ellipses, or polygons, each with their own parameters.

A **polygon** consists of a list of vertices. Ellipses and polygons are both closed shapes, which have a fill color and a fill pattern.

Lines are one dimensional and cannot be filled.

TextWindow is a kind of a ScrollingWindow, which has a two-dimensional scrolling offset within its window, as specified by xOffset and yOffset, as well as an operation scroll to change the scroll value. A text window contains a string and has operations to insert and delete characters.

ScrollingCanvas is a special kind of canvas that supports scrolling; it is both a Canvas and a ScrollingWindow. This is an example of multiple inheritance.

A **Panel** contains a set of PanelItem objects, each identified by a unique itemName within a given panel, as shown by the qualified association.

Each **panel item** belongs to a single panel. A panel item is a predefined icon with which a user can interact on the screen. Panel items come in three kinds: buttons, choice items, and text items.

A **button** has a string that appears on the screen; a button can be pushed by the user and has an attribute depressed.

A **choice item** allows the user to select one of a set of predefined choices, each of which is a ChoiceEntry containing a string to be displayed and a value to be returned if the entry is selected.

There are two associations between **ChoiceItem** and **ChoiceEntry**; a one-to-many association defines the set of allowable choices, while a one-to-one association identifies the current choice. The current choice must be one of the allowable choices, so one association is a subset of the other as shown by the arrow between them labeled "{subset}." This is an example of a constraint.

When a panel item is selected by the user, it generates an **Event**, which is a signal that something has happened together with an action to be performed.

All kinds of panel items have notifyEvent associations. Each panel item has a single event, but one event can be shared among many panel items.

Text items have a second kind of event, which is generated when a keyboard character is typed while the text item is selected.

The association with end name keyboardEvent shows these events. Text items also inherit the notifyEvent from superclass PanelItem; the notifyEvent is generated when the entire text item is selected with a mouse.

There are many deficiencies in this model.

E.g., perhaps we should define a type Rectangle, which can then be used for the window and canvas boundaries, rather than having two similar sets of four position attributes. Maybe a line should be a special case of a polyline (a connected series of line segments), in which case both Polyline and Polygon could be subclasses of a new superclass that defines a list of points. Many attributes, operations, and classes are missing from a description of a realistic windowing system. Certainly the windows have associations among themselves, such as overlapping one another. Nevertheless, this simple model gives a flavor of the use of class modeling. We can criticize its details because it says something precise. It would serve as the basis for a fuller model.

NAVIGATION OF CLASS MODELS

Navigation is important because it lets you exercise a model and uncover hidden flaws and omissions so that you can repair them. We can perform navigation manually (an informal technique) or write navigation expressions (as we will explain).

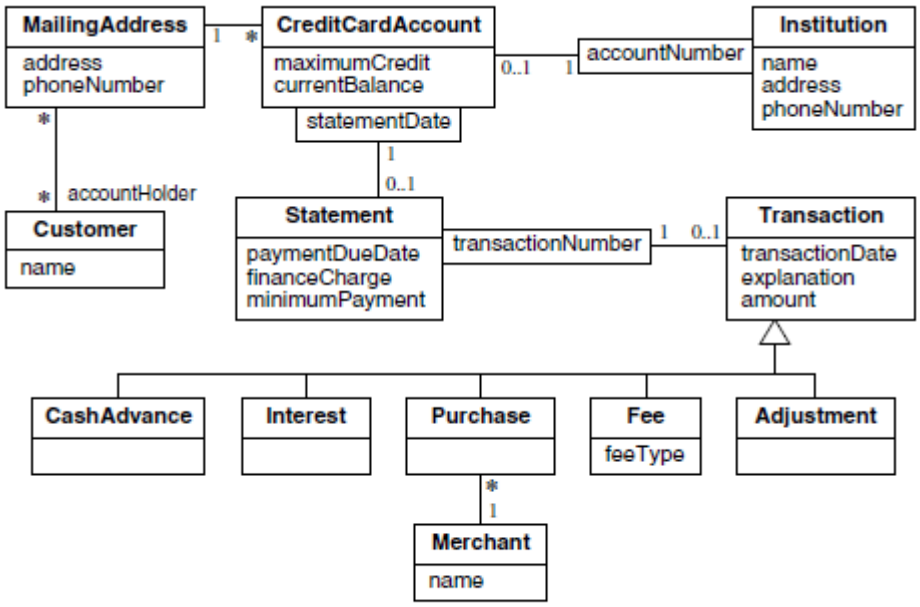


Figure 3.27 Class model for managing credit card accounts

Consider the simple model for **credit card accounts** in the above figure.

An institution may issue many credit card accounts, each identified by an account number.

Each account has a maximum credit limit, a current balance, and a mailing address.

The account serves one or more customers who reside at the mailing address.

The institution periodically issues a statement for each account.

The statement lists a payment due date, finance charge, and minimum payment. The statement itemizes various transactions that have occurred throughout the billing interval: cash advances, interest charges, purchases, fees, and adjustments to the account.

The name of the merchant is printed for each purchase.

We can pose a variety of questions against the model.

- What transactions occurred for a credit card account within a time interval?
- What volume of transactions were handled by an institution in the last year?
- What customers patronized a merchant in the last year by any kind of credit card?
- How many credit card accounts does a customer currently have?
- What is the total maximum credit for a customer, for all accounts?

The UML incorporates a language that can express these kinds of questions—the **Object Constraint Language (OCL)**

1. **OCL Constructs for Traversing Class Models**

The OCL can traverse the constructs in class models.

1. *Attributes*.

You can traverse from an object to an attribute value. The syntax is the source object, followed by a dot, and then the attribute name. For example, the expression:

aCreditCardAccount.maximumCredit takes a CreditCardAccount object and finds the value of maximumCredit. (We use the convention of preceding a class name by “a” to refer to an object.)

Similarly, you can access an attribute for each object in a collection, returning a collection of attribute values. In addition, you can find an attribute value for a link, or a collection of attribute values for a collection of links.

2. *Operations*.

You can also invoke an operation for an object or a collection of objects.

The syntax is the source object or object collection, followed by a dot, and then the operation.

An operation must be followed by parentheses, even if it has no arguments, to avoid confusion with attributes. You may invoke operations from your class model or predefined operations that are built into the OCL.

The OCL has special operations that operate on entire collections (as opposed to operating on each object in a collection).

E.g., you can count the objects in a collection or sum a collection of numeric values. The syntax for a collection operation is the source object collection, followed by "->", and then the operation.

3. *Simple associations*.

A third use of the dot notation is to traverse an association to a target end. The target end may be indicated by an association end name or, where there is no ambiguity, a class name.

In the example, aCustomer.MailingAddress yields a set of addresses for a customer (the target end has "many" multiplicity). In contrast, aCredit-CardAccount.MailingAddress yields a single address (the target end has multiplicity of one).

4. *Qualified associations*.

A qualifier lets you make a more precise traversal.

The expression aCreditCardAccount.Statement[30 November 1999] finds the statement for a credit card account with the statement date of 30 November 1999.

The syntax is to enclose the qualifier value in brackets. Alternatively, you can ignore the qualifier and traverse a qualified association as if it were a simple association.

Thus the expression aCredit-CardAccount.Statement finds the multiple statements for a credit card account. (The multiplicity is "many" when the qualifier is not used.)

5. *Association classes*.

Given a link of an association class, you can find the constituent objects. Alternatively, given a constituent object, you can find the multiple links of an association class.

6. *Generalizations.*

Traversal of a generalization hierarchy is implicit for the OCL notation.

7. *Filters*.

There is often a need to filter the objects in a set.

The OCL has several kinds of filters, the most common of which is the select operation.

The select operation applies a predicate to each element in a collection and returns the elements that satisfy the predicate.

For example, aStatement.Transaction->select(amount>$100) finds the transactions for a statement in excess of $100.

2. **Building OCL Expressions**

The real power of the OCL comes from combining primitive constructs into expressions.

For example, an OCL expression could chain together several association traversals. There could be several qualifiers, filters, and operators as well.

With the OCL, a traversal from an object through a single association yields a singleton or a set (or a bag if the association has the annotation {bag} or {sequence}).

In general, a traversal through multiple associations can yield a bag (depending on the multiplicities), so you must be careful with OCL expressions.

A set is a collection of elements without duplicates. A bag is a collection of elements with duplicates allowed.

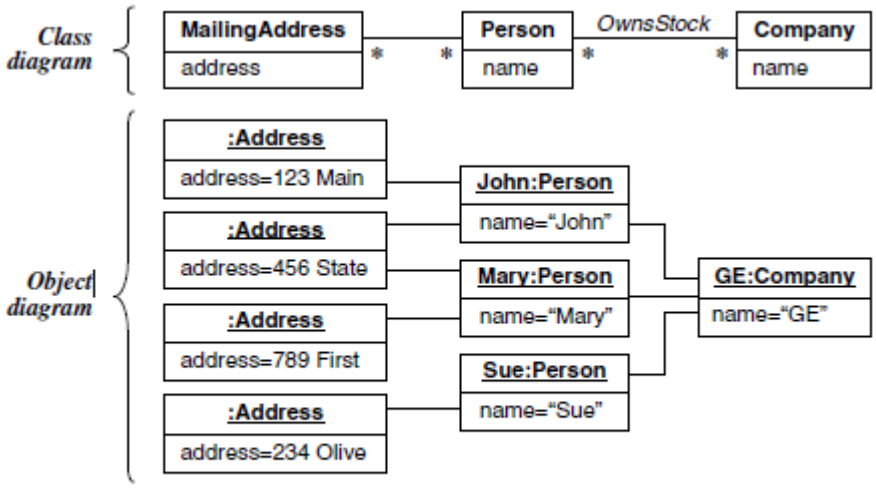


Figure 3.28 A sample model and examples. Traversal of multiple associations can yield a bag.

The above figure illustrates how an OCL expression can yield a bag.

A company might want to send a single mailing to each stockholder address. Starting with the GE company, we traverse the OwnsStock association and get a set of three persons. Starting with these three persons and traversing to mailing address, we get a bag obtaining the address 456 State twice.

Null is a special value denoting that an attribute value is unknown or not applicable.

Nulls do not arise for properly phrased and valid constraints.

But they certainly do arise with model navigation. E.g., a person may lack a mailing address.

We extend the meaning of OCL expressions to accommodate nulls—a traversal may yield a null value, and an OCL expression evaluates to null if the source object is null.

3. **Examples of OCL Expressions**

We can use the OCL to answer the credit card questions.

1. What transactions occurred for a credit card account within a time interval?

aCreditCardAccount.Statement.Transaction->select(aStartDate <= transactionDate and transactionDate <= anEndDate)

The expression traverses from a *CreditCardAccount* object to Statement and then to Transaction, resulting in a set of transactions. (Traversal of the two associations results in a set, rather than a bag, because both associations are

one-to-many.) Then we use the OCL select operator (a collection operator) to find the transactions within the time interval bounded by *aStartDate* and *anEndDate*.

2. What volume of transactions were handled by an institution in the last year?

anInstitution.CreditCardAccount.Statement.Transaction->select(aStartDate <= transactionDate and transactionDate <= anEndDate).amount->sum()

The expression traverses from an Institution object to *CreditCardAccount*, then to Statement, and then to Transaction. (Traversal results in a set, rather than a bag, because all three associations are one-to-many.) The OCL select operator finds the transactions within the time interval bounded by *aStartDate* and *anEndDate*. (We choose to make the time interval more general than last year.) Then we find the amount for each transaction and compute the total with the OCL sum operator (a collection operator).

3. What customers patronized a merchant in the last year by any kind of credit card?

aMerchant.Purchase-> select(aStartDate <= transactionDate and transactionDate <= anEndDate).Statement.CreditCardAccount.MailingAddress.Customer->asSet()

The expression traverses from a Merchant object to Purchase. The OCL select operator finds the transactions within the time interval bounded by *aStartDate* and *anEndDate*. (Traversal across a generalization, from Purchase to Transaction, is implicit in the OCL.) For these transactions, we then traverse to Statement, then to *CreditCardAccount*, then to *MailingAddress*, and finally to Customer. The association from *MailingAddress* to Customer is many-to-many, so traversal to Customer yields a bag. The OCL *asSet* operator converts a bag of customers to a set of customers, resulting in our answer.

4. How many credit card accounts does a customer currently have?

aCustomer.MailingAddress.CreditCardAccount->size()

Given a Customer object, we find a set of *MailingAddress* objects. Then, given the set of *MailingAddress* objects, we find a set of *CreditCardAccount* objects. (This traversal yields a set, and not a bag, because each *CreditCardAccount* pertains to a single *MailingAddress*.)

For the set of *CreditCardAccount* objects we apply the OCL size operator, which returns the cardinality of the set.

5. What is the total maximum credit for a customer, for all accounts?

aCustomer.MailingAddress.CreditCardAccount.maximumCredit->sum()

The expression traverses from a Customer object to *MailingAddress*, and then to *CreditCardAccount*, yielding a set of *CreditCardAccount* objects. For each *CreditCardAccount*, we find the value of *maximumCredit* and compute the total with the OCL sum operator.

Note that these kinds of questions exercise a model and uncover hidden flaws and omissions that can then be repaired. For example, the query on the number of credit card accounts suggests that we may need to differentiate past accounts from current accounts.

OCL was originally intended as a constraint. However, as we explain here, the OCL is also useful for navigating models.

****

INTRODUCTION TO RUP AND UML DIAGRAMS

RATIONAL UNIFIED PROCESS (RUP)

Rational Unified Process is a software development process for object-oriented models. It is also known as Unified Process Model. It was created by Rational Corporation and is designed and documented using UML(Unified Modeling Language)

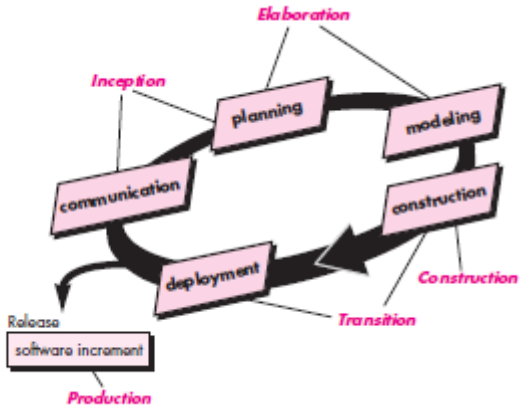
The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system.

It emphasizes the important role of software architecture and helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse.

It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

Phases of Unified Process:

FIGURE 2.9
The Unified Process



1. *Inception Phase*

   The inception phase of the Unified Process encompasses both customer communication and planning activities. By collaborating with stakeholders, business requirements for the software are identified; a rough architecture for the system is proposed; and a plan for the iterative, incremental nature of the ensuing project is developed.

2. *Elaboration phase*

   The elaboration phase encompasses the communication and modeling activities of the generic process model.

   Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—the use case model, the requirements model, the design model, the implementation model, and the deployment model.

3. *Construction phase*

   The construction phase of the Unified Phase is identical to the construction activity defined for the generic software process.

   Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users.

4. *Transition phase*

The transition phase of the Unified Phase encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity.

Software is given to end users for beta testing and user feedback reports both defects and necessary changes.

5. *Production phase*

The production phase of the Unified Process coincides with the deployment activity of the generic process.

During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.

The five Unified Process phases do not occur in a sequence, but rather with staggered concurrency.

****

Introduction

Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of object-oriented software engineering. UML includes a set of graphic notation techniques to create visual models of object-oriented software systems. UML combines techniques from data modeling, business modeling, object modeling, and component modeling and can be used throughout the software development life-cycle and across different implementation technologies.

Modeling

There is a difference between a UML model and the set of diagrams of a system. A diagram is a partial graphic representation of a system's model. The model also contains documentation that drives the model elements and diagrams (such as written use cases).

UML diagrams represent two different views of a system model:

Static (or structural) view

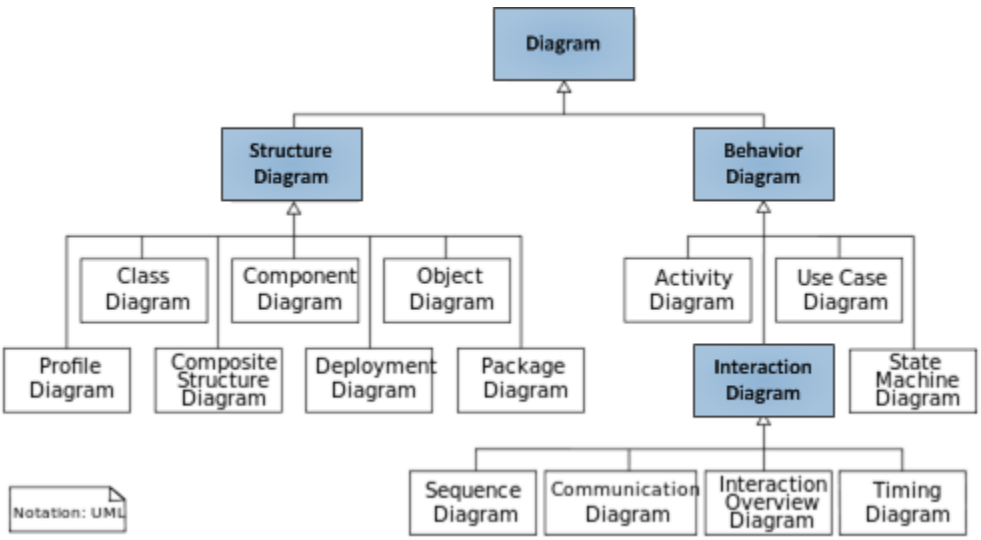
This view emphasizes the static structure of the system using objects, attributes, operations, and relationships. Ex: Class diagram, Composite Structure diagram.

Dynamic (or behavioral) view

This view emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. Ex: Sequence diagram, Activity diagram, State Machine diagram.

Diagrams Overview

UML 2.2 has 14 types of diagrams divided into multiple categories as shown in the figure below.

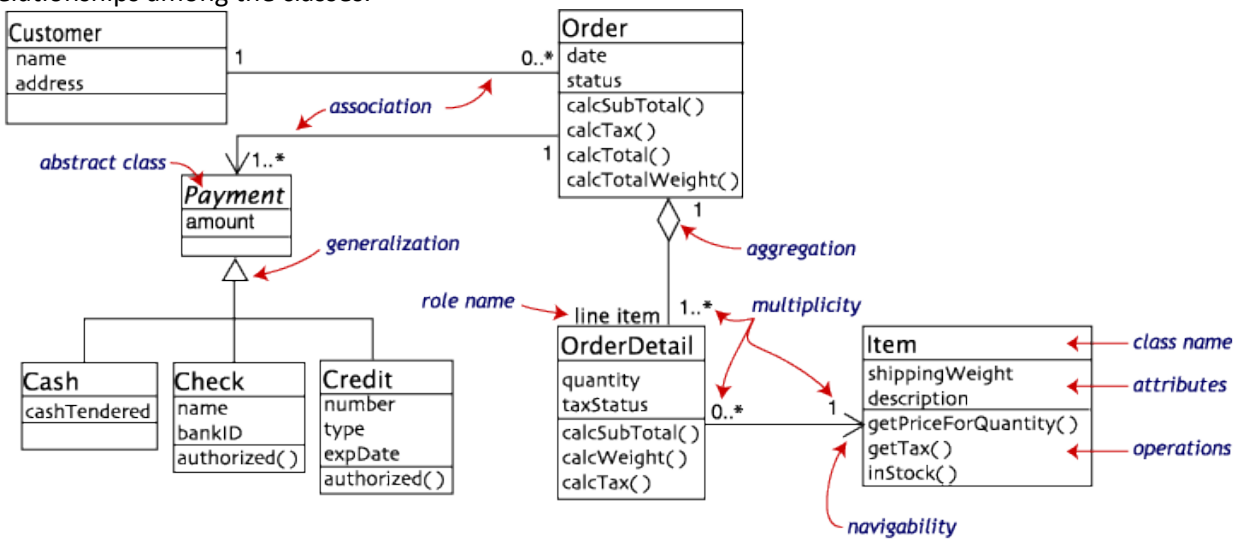


Structure Diagrams

These diagrams emphasize the things that must be present in the system being modeled. Since they represent the structure, they are used extensively in documenting the software architecture of software systems.

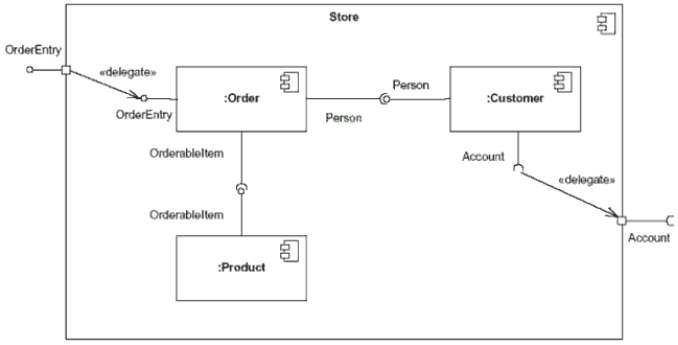
1. Class Diagram

Describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.



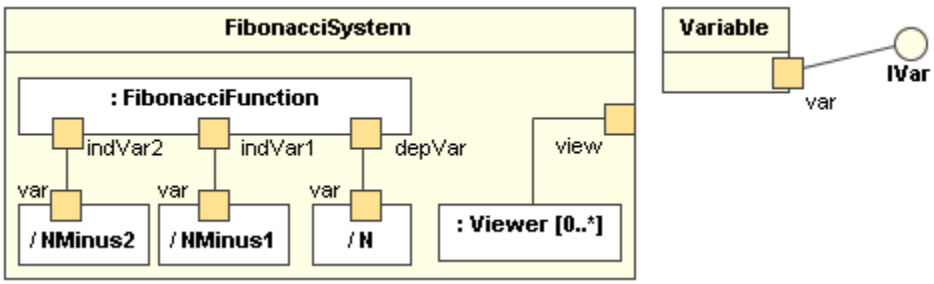
2. Component Diagram

Describes how a software system is split-up into components and shows the dependencies among these components.



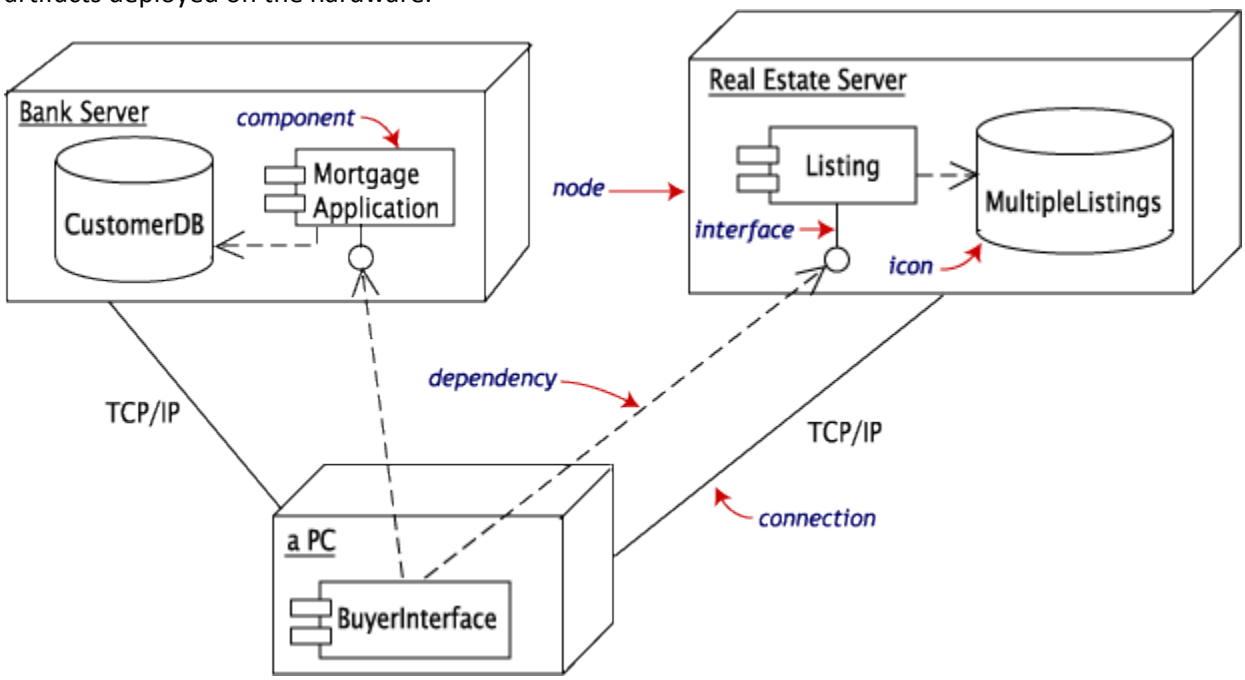
3. Composite Structure Diagram

Describes the internal structure of a class and the collaborations that this structure makes possible.



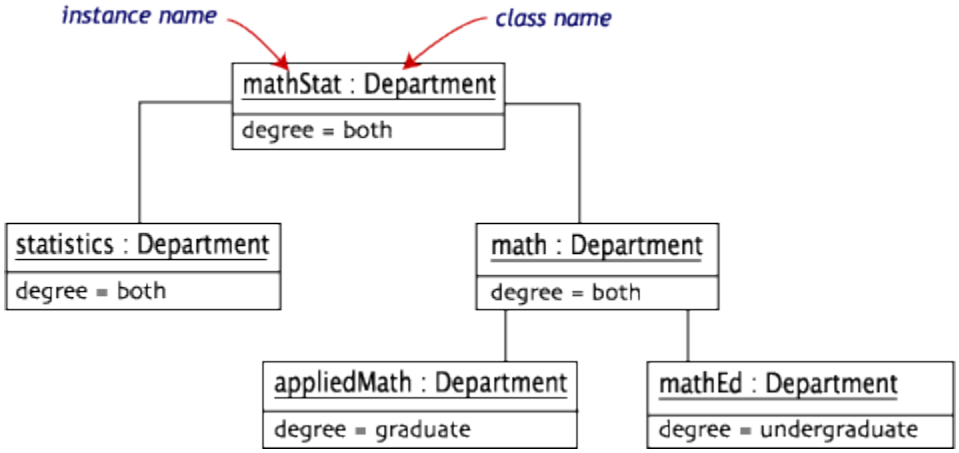
4. Deployment Diagram

Describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.



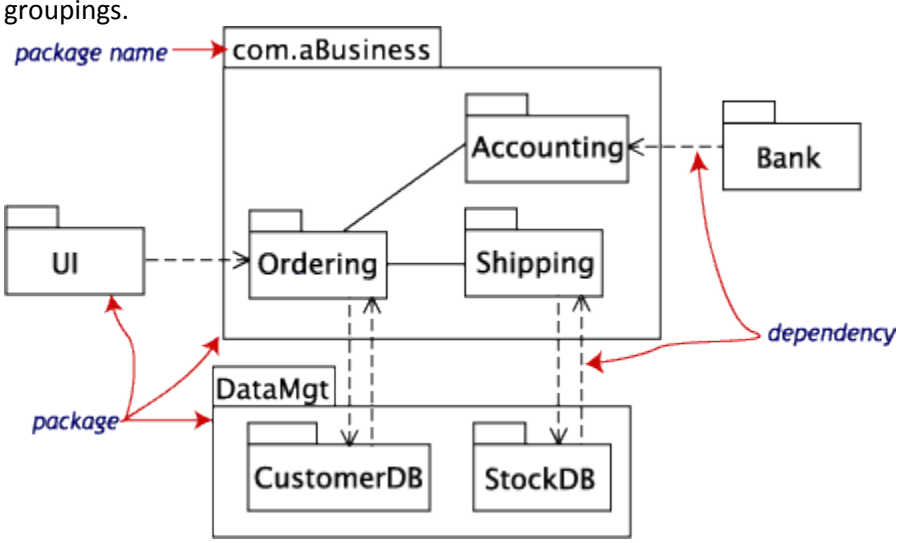
5. Object Diagram

Shows a complete or partial view of the structure of an example modeled system at a specific time.



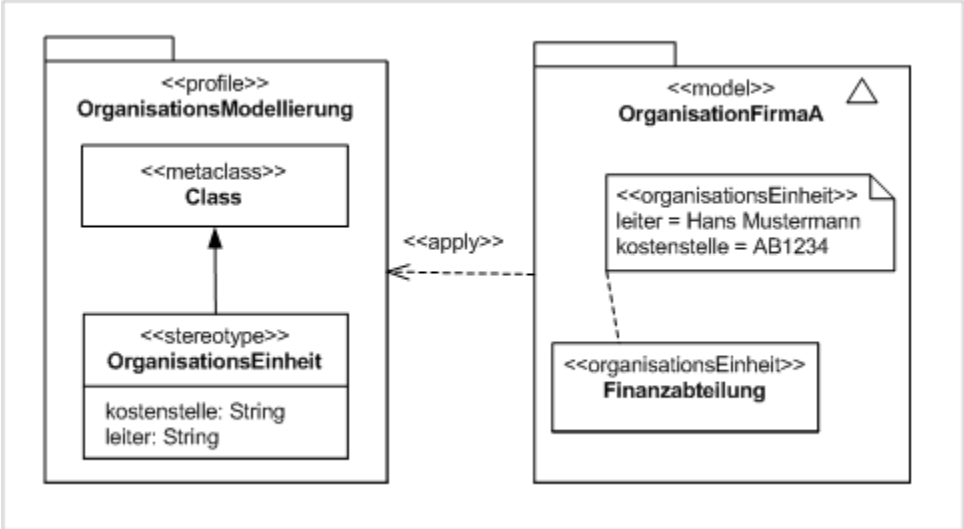
6. Package Diagram

Describes how a system is split-up into logical groupings by showing the dependencies among these groupings.



7. Profile Diagram

Operates at the metamodel level to show stereotypes as classes with the <<stereotype>> stereotype, and profiles as packages with the <<profile>> stereotype. The extension relation (solid line with closed, filled arrowhead) indicates what metamodel element a given stereotype is extending.

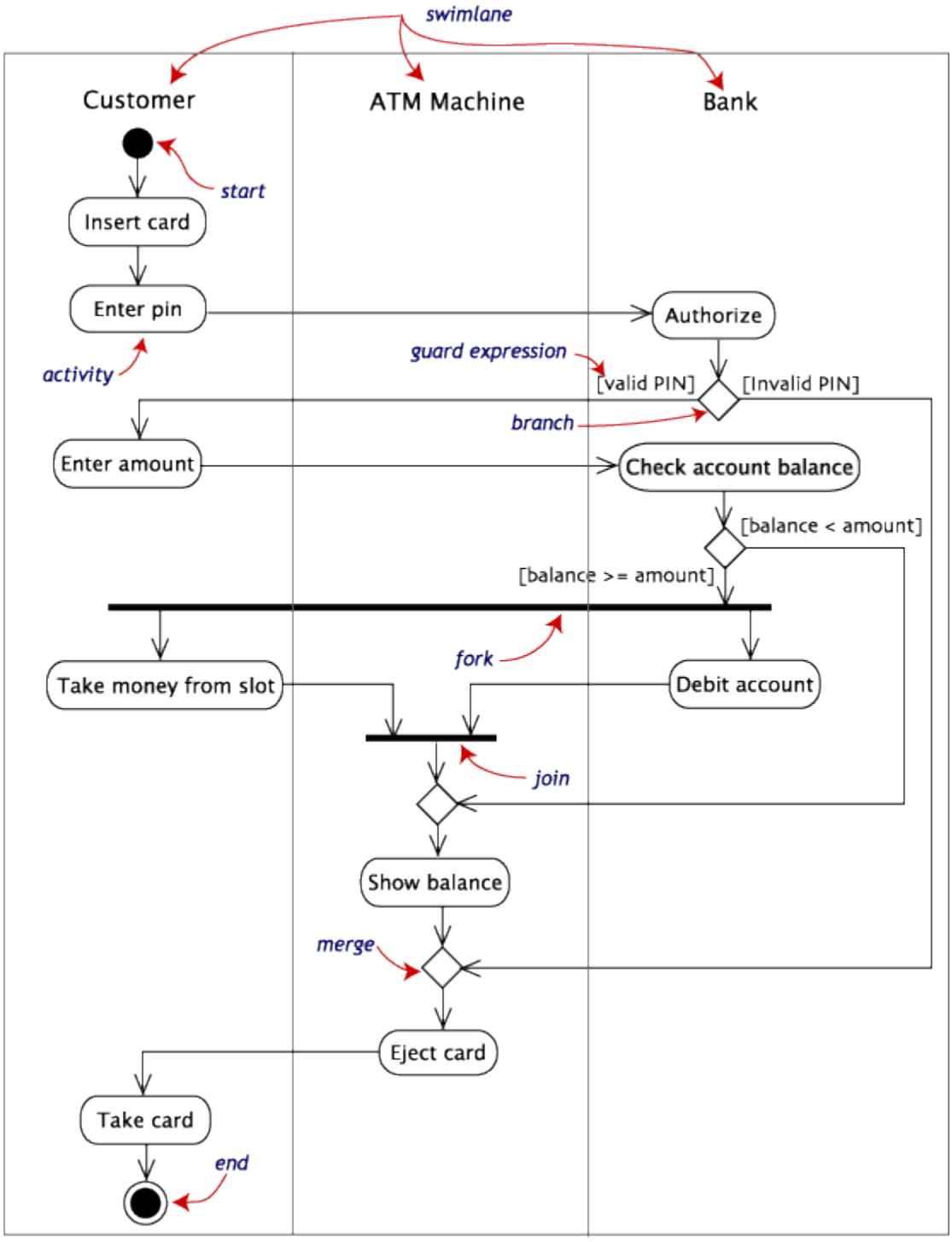


Behavior Diagrams

These diagrams emphasize what must happen in the system being modeled. Since they illustrate the behavior of a system, they are used extensively to describe the functionality of software systems.

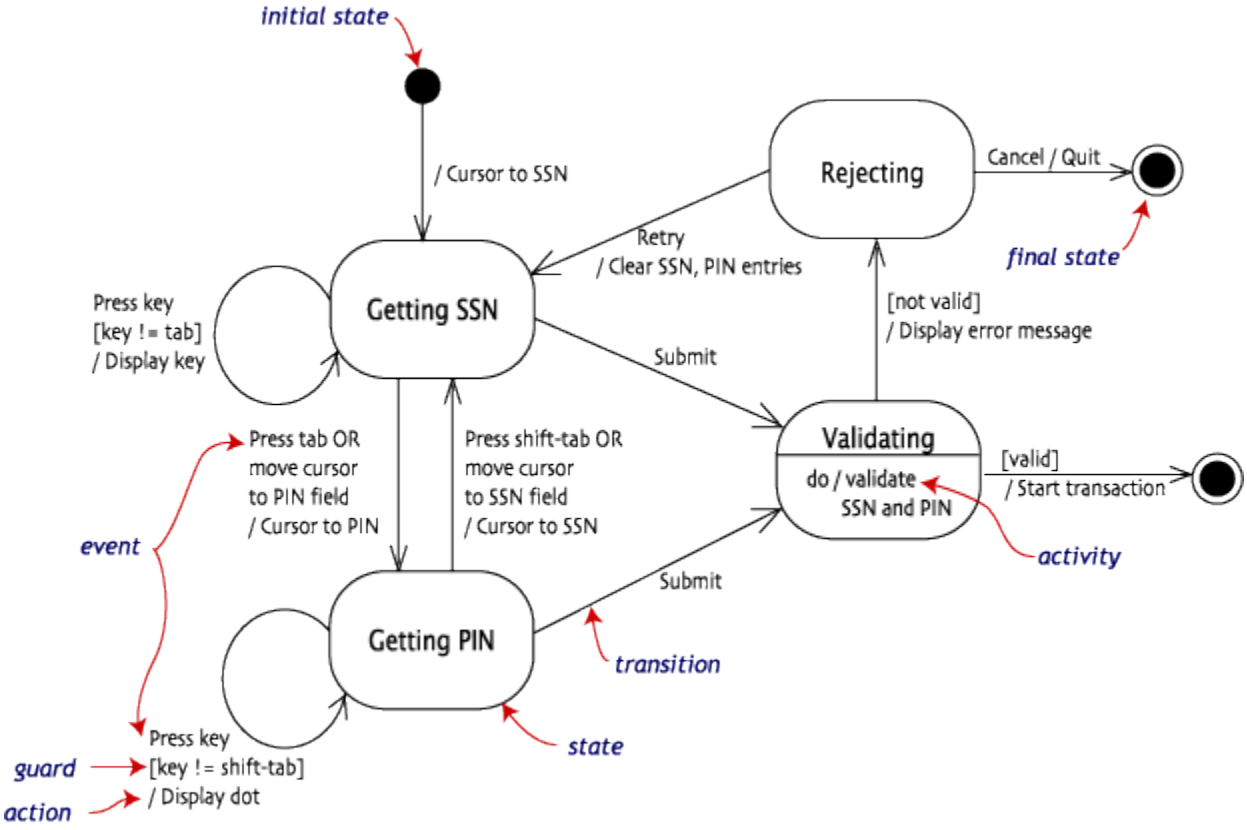
1. Activity Diagram

Describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.



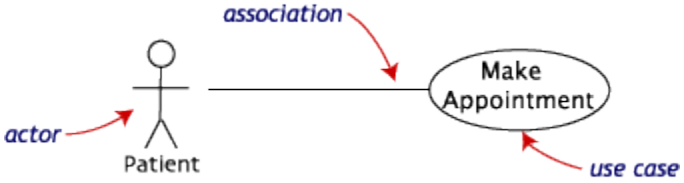
2. State Machine Diagram

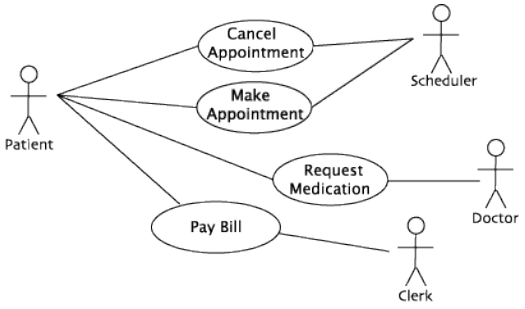
Describes the states and state transitions of the system.



3. Use Case Diagram

Describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.



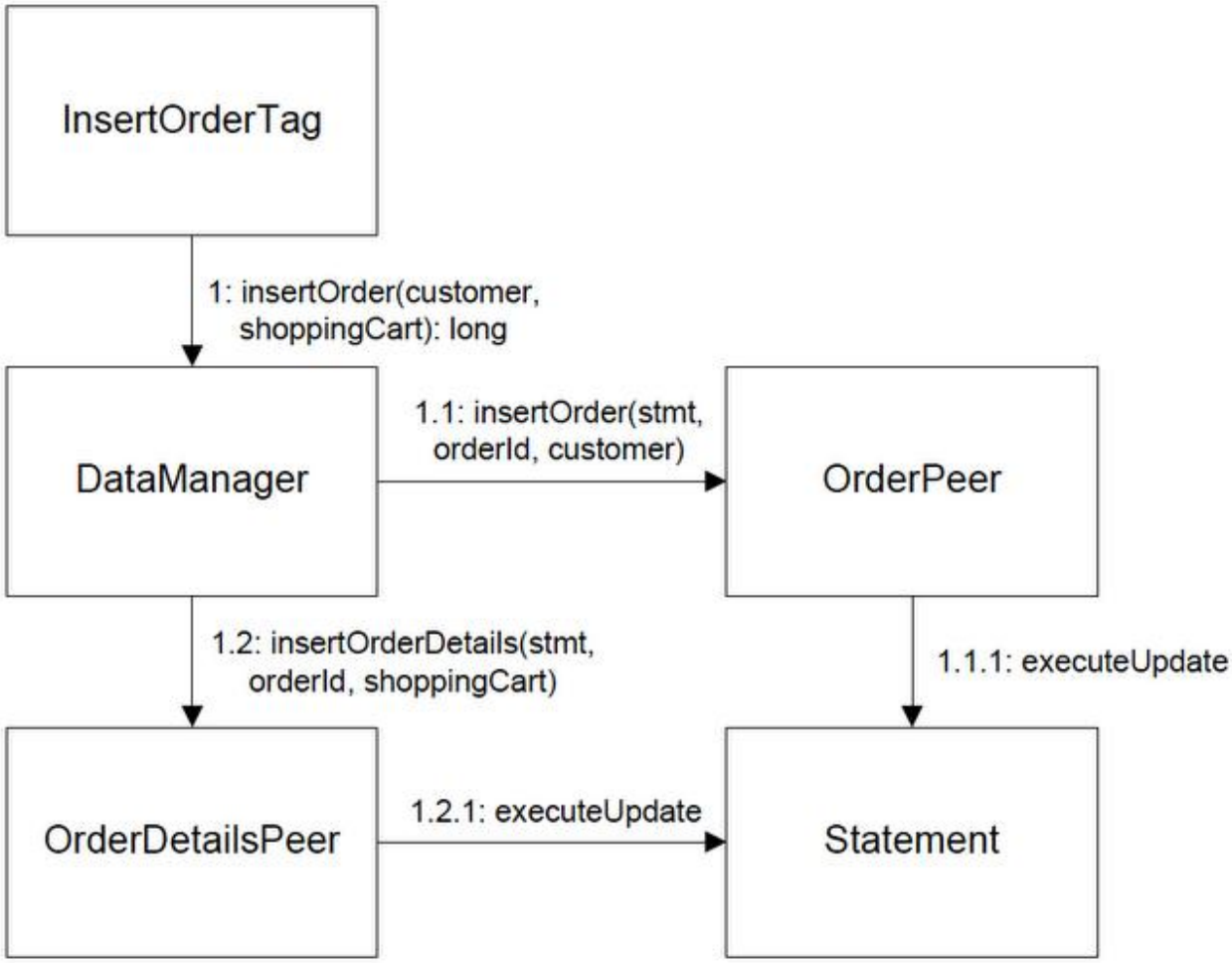


Interaction Diagrams

These diagrams are a subset of behavior diagrams, emphasizing the flow of control and data among the things in the system being modeled.

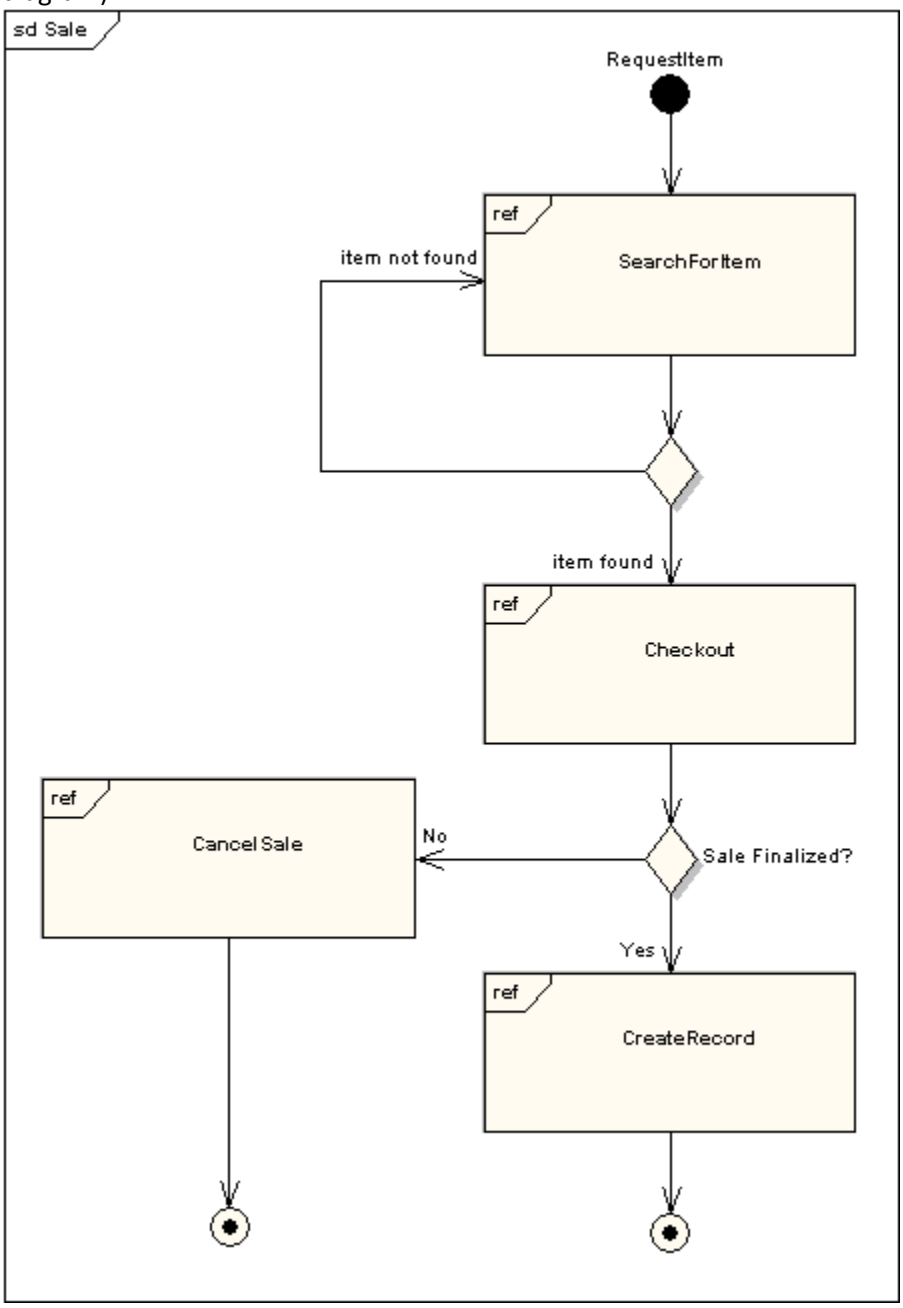
1. Communication Diagram

Shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system.



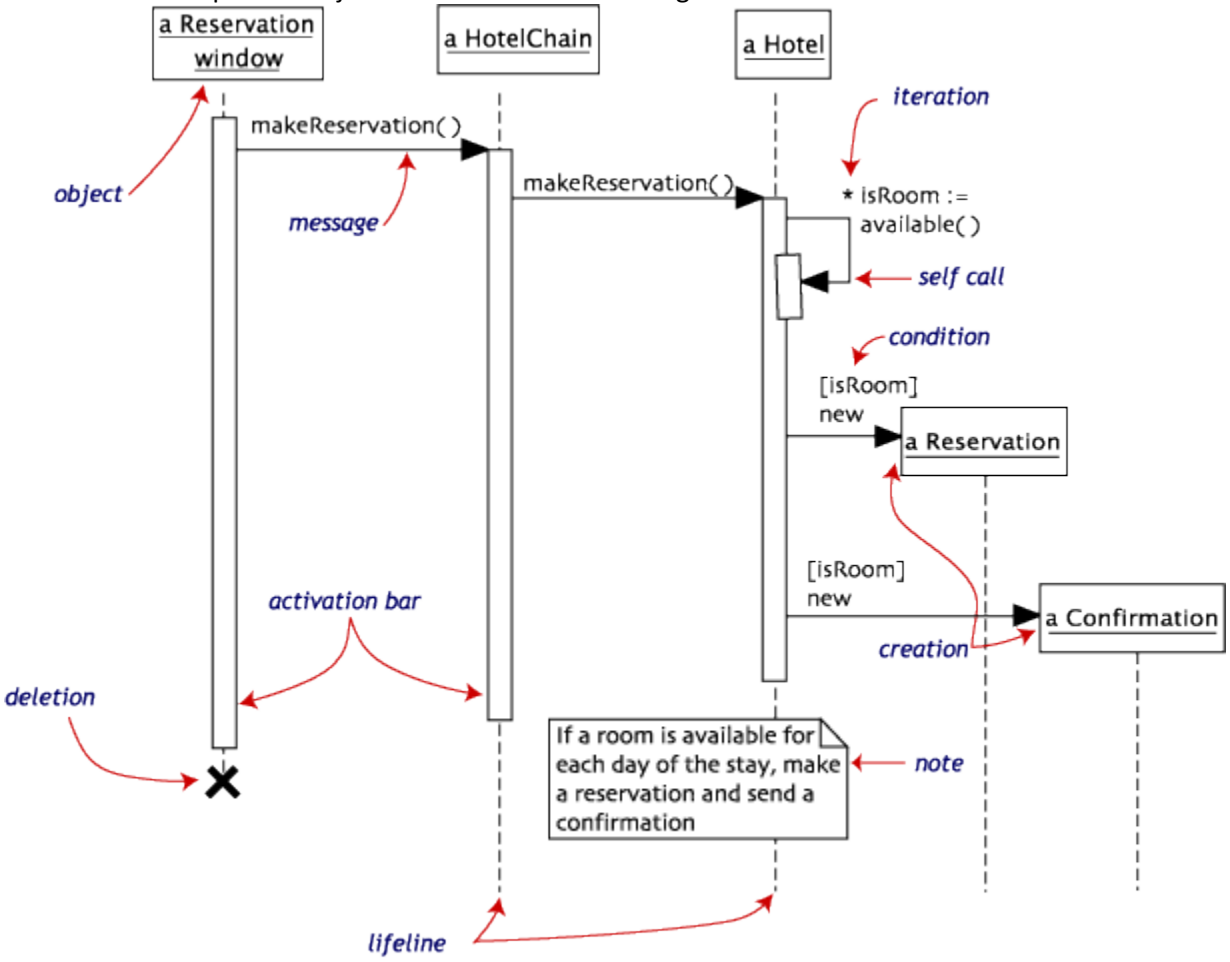
2. Interaction Overview Diagram

Provides an overview in which the nodes represent communication diagrams. They are activity diagrams in which every node, instead of being an activity, is a rectangular frame containing an interaction diagram (i.e., a communication, interaction overview, sequence, or UML timing diagram).



3. Sequence Diagram

Shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespans of objects relative to those messages.



4. Timing Diagram

A specific type of interaction diagram where the focus is on timing constraints. Timing diagrams model sequence of events and their effects on states and property values. Time flows along a horizontal axis from left to right. They can be used to show method execution profiling or concurrency scenarios.