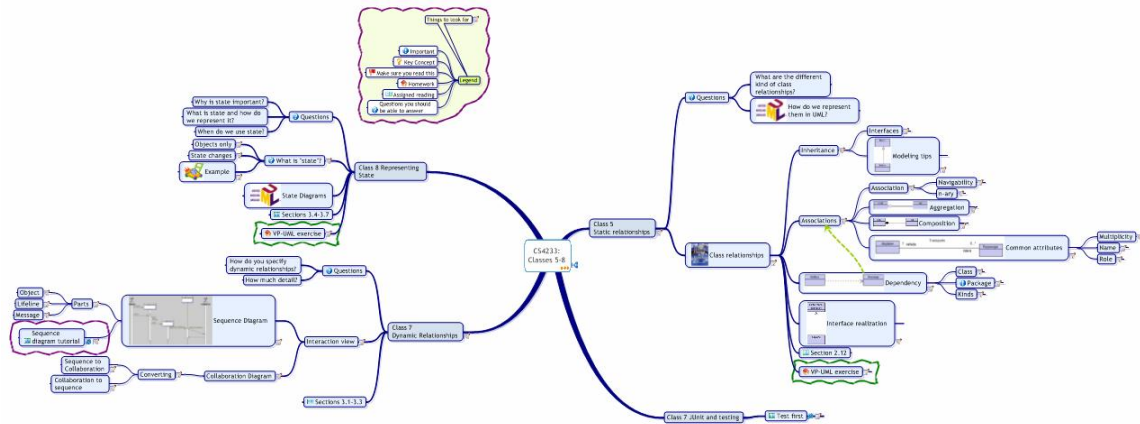


CS4233: Classes 5-8



See document: [CS4233 Object-Oriented Analysis and Design.mmap](#)

1 Class 5 Static relationships

Static relationships are usually specified between classes. Static relationships do not change as objects are created and destroyed during program execution.

1.1 Questions

1.1.1 What are the different kind of class relationships?

1.1.2 How do we represent them in UML?

1.2 Class relationships

We will group both class and object relationships into the discussion of class relationships. As an O-O developer one important activity will be to define and refine the relationships between classes in your designs.

There are several important class relationships. There are also different characteristics, or attributes that you should strive for in your designs that will be reflected by the relationships between classes.

Finally there are specific ways to represent the relationships with UML.

All of these are discussed in the subsequent items.

1.2.1 Inheritance

Inheritance represents an *is-a* relationship. It is usually the first class relationship that students learn when they are taught object-oriented concepts.

A subclass is a specialization of its parent classes. For this class we will only consider single inheritance as in Java, but we will often approximate multiple inheritance through the use of interfaces.

Interfaces

Some languages, like Java, provide specific interface mechanisms that allow you to factor out specific behavior that a class must provide. This mechanism may be used to provide many of the

benefits of multiple inheritance without experiencing the problems and complexity that comes with full implementations of multiple inheritance.

Modeling tips

Taken from Booch, et. al.

To model inheritance relationships,

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

1.2.2 Associations

There are several types of associations that can exist between two classes. These are often cause for confusion when you begin to learn OOAD.

The following items describe the differences between the associations.

See: <http://ootips.org/uml-hasa.html> for one description of the different types.

Association

Association represents the ability of one instance to send a message to another instance. This is typically implemented with a pointer or reference instance variable, although it might also be implemented as a method argument, or the creation of a local variable.

From Booch, et. al.,

An *association* is a structural relationship that specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can navigate from an object of one class to an object of the other class (and vice- versa). There may also be an association from a class to itself.

Navigability

Associations are represented by solid lines between classes. If the line has *no arrows* then the navigation occurs in both directions. That is, from an instance of either class, you can get to the associated object.

If there is an arrowhead then the navigation is one way, in the direction of the arrowhead. When the navigation is bi-directional, then no arrowheads are shown.

n-ary

Normally associations are binary. That is that they connect exactly two classes. This is not, however, required. There may be n-ary associations, where $n > 0$.

Aggregation

Aggregation is the typical whole/part relationship. This is exactly the same as an association with the exception that instances cannot have cyclic aggregation relationships (i.e. a part cannot contain its whole).

We often refer to the relationship as an *is-a-part-of*, or *has-a*, relationship. The text referst to it as a *has* relationship.

From Booch, et. al.:

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

Aggregation is modeled by an open diamond at the *whole* end.

Composition

Composition is exactly like Aggregation except that the lifetime of the 'part' is controlled by the 'whole'. This control may be direct or transitive. That is, the 'whole' may take direct responsibility for creating or destroying the 'part', or it may accept an already created part, and later pass it on to some other whole that assumes responsibility for it.

From Booch, et. al.:

Composition is a form of aggregation with strong ownership and coincident lifetime of the parts by the whole.

This means that the parts cannot exist outside of the whole. Composition is shown by a solid diamond at the *whole* end.

Common attributes

There are several attributes, or properties, of associations that can be added to the UML model. These should be used to clarify the meaning of the association. A disadvantage of using the attributes is that they can quickly clutter up a diagram. Remember, the purpose of a model is to show a system at an appropriate level of abstraction by providing a simpler, less detailed view than might otherwise be shown.

Multiplicity

Multiplicity is the most common attribute on an association. It is used to indicate a numerical relationship between classes. For example, one Aircraft may contain any number of passengers, you would place a 1 on the Aircraft end of the association and a * on the Passenger end. Multiplicity is represented by a number or a range, such as 1..4. Common multiplicity indicators are:

- 0
- 1
- 0..n (where n is some specific number)

- 0..* (which is the same as *, meaning any number)
- 0..1

You can also specify an ordering on multiplicity by using keywords such as *ordered* within braces, like: 0..4 {ordered}.

Name

Associations may have a name. The name is meant to express the purpose of an association, such as an Airplane "transports" riders.

In a diagram, the classes and names are usually read left-to-right or top-to-bottom. If the meaning of the name is not clear, you may draw a solid arrowhead showing the direction of the association.

Role

More properly, the *role* is a *rolename* with respect to associations. It is a name for a particular association end name. In UML 2.0, the official term for a rolename is an *association end name*.

The rolename is used as an indication of the variable name that is used to access appropriate instances of the opposite class when one generates code from the UML model. We will not be concerned with code generation in this course.

1.2.3 Dependency

See also: [Associations](#)

Dependency is often overlooked, yet is one of the most important relationships to examine to obtain a sound design.

A class instance that manipulates an instance of another class any way *depends* upon the other class.

Another way of thinking about dependency is that if the independent class changes (usually its interface), then the dependent class will require a change (usually).

Associations are dependencies. However, two classes can have a dependency relationship without having an association relationship.

In UML, dependency relationships are represented by a dashed line with an open arrow pointing to the independent element.

Class

Class dependencies will usually be represented as associations.

Package

Dependencies can occur between many different types of modeling elements. One important one is the package dependency. We will use this later in the course to measure some important properties of a system.

Packages are often a better granularity for computing metrics.

Kinds

There are many kinds of dependencies. Some that are described in Rumbaugh, et. al. are:

- access - a private import of the contents of another package
- binding - assignment of values to the parameters of a template to generate a new model element

- call - statement that a method of one class calls an operation of another class
- derivation - statement that one instance can be computed from another instance
- instantiation - statement that a method of one class creates instances of another class
- permission - permission for an element to use the contents of another element
- realization - mapping between a specification and an implementation of it
- refinement - statement that a mapping exists between elements at two different semantic levels
- send - relationship between the sender of a signal and the receiver of the signal
- substitution - statement that the source class supports the interfaces and contracts of the target class and may be substituted for it
- trace dependency - statement that some connection exists between elements in different models, but less precise than a mapping
- usage - statement that one element requires the presence of another element for its correct functioning

1.2.4 Interface realization

When you have defined interfaces, you will want to indicate a relationship between classes that realize, or implement, the interface and the interface itself.

In UML, this is represented by a dashed-line connector with a triangular arrowhead pointing to the interface. This is the same as a the inheritance connector except that the line is not solid.

Interfaces are shown as classes with the <<interface>> stereotype.

1.2.5 Section 2.12

1.2.6 VP-UML exercise

Create a class diagram that models the static relationship between the elements of this course. This should include classes, or interfaces, for: student, professor, homework, test, project, and other things that make sense. Use the class notes thus far, especially the course overview, to get ideas for classes.

You should show the static relationships that you think are necessary in order to completely describe the model of this course.

Print out your diagram, make sure your name is on it, and bring it to class 7.

2 Class 7 JUnit and testing

2.1 Test first

See document: 4929.html

The linked article describes the practice of writing tests before coding. It provides a gentle introduction to the practice.

Although it is not required that you test first, Test First Programming (TFP), or Test-Driven Development (TDD) as it is also called, is a good way to learn how to use JUnit well.

3 Class 7 Dynamic Relationships

Unlike static relationships, dynamic relationships are usually specified between objects. These relationships will change as objects are created, destroyed, and change state during program execution.

3.1 Questions

3.1.1 How do you specify dynamic relationships?

3.1.2 How much detail?

3.2 Interaction view

In the UML Reference Manual, 2ed. The dynamic interaction of objects is modeled in, what Rumbaugh calls, the interaction view. There are two types of diagrams used for the interaction view - the Sequence Diagram and the Collaboration Diagram.

Note that in UML 2.0, there are different types of collaboration diagrams (i.e there is a Communication Diagram), but this is beyond the scope of this course.

3.2.1 Sequence Diagram

The sequence diagram shows a temporal relationship between objects. That is, it shows a sequence of interactions between objects in a time ordering.

The diagram presents a two-dimensional view. The vertical dimension is the time axis, with time proceeding down the page. The horizontal axis shows the roles that are represented by individual objects.

Parts

There are several parts of a sequence diagram that you should be familiar with.

Object

Objects represent roles in a sequence diagram. They are depicted as rectangles, like classes; without the details of operations and properties, and with the name underlined.

Any object is given a name, such as anAirplane, and may be given a type as well, such as anAirplane:Airplane. If you leave the name off, you must supply a type. This is called an *anonymous object* and is named like :Airplane.

Most of the objects in a sequence diagram are

Lifeline

The lifeline is a dashed line that starts at the object and goes down from it. When there is an active procedure executing on the object, it may be shown as a double solid line that will often be drawn as a rectangle. This is also called the *execution specification*, or activation.

Message

A message is the invocation of an object's operation from another (possibly the same) object.

There are two types of objects, asynchronous, and synchronous. Asynchronous messages are shown as solid lines with open (stick) arrows in the direction of object whose operation is being invoked.

Synchronous messages have a solid triangular arrowhead.

Returns from messages are not usually shown in sequence diagrams.

Messages are given names and, depending upon the detail of the model, they may show the parameters / arguments passed to the receiving object.

Messages may or may not be numbered sequentially to emphasize the temporal nature. They may also use tumbler numbering, for example a message might be numbered 2.1.2.

Sequence diagram tutorial

See document: [UMLSequenceDiagrams.pdf](#)

Read the tutorial on sequence diagrams written by Bob Martin of ObjectMentor. Make sure that you understand how to read and construct such a diagram.

3.2.2 Collaboration Diagram

A collaboration diagram is semantically equivalent to a sequence diagram. The purpose of the collaboration diagram is to show more of a structural / spatial relationship between the objects than a temporal relationship.

The parts of the collaboration diagram are basically a subset of the sequence diagram. Objects and messages are the primary modeling elements. The life lines and activations are not present.

Converting

You can convert a sequence diagram to a collaboration and vice versa, quite easily.

Sequence to Collaboration

To convert a sequence diagram to a collaboration diagram, do the following:

- Take the first (topmost) message in the sequence diagram. Place the two associated objects on your collaboration diagram and draw the message between the two. Number the message (usually numbered 1, 2, etc., but some other numbering may be used to show sub-operations).
- Take the next message. If the objects at ends of the message in the sequence diagram are not on the collaboration diagram, place them on the collaboration diagram and draw the message. Number the message appropriately.
- Repeat the previous step until all messages have been accounted for.

Collaboration to sequence

To convert a collaboration diagram to a sequence diagram, do the following:

- Find the first message on the collaboration diagram. Place the two objects at the ends of the message on the sequence diagram. Add the lifelines, and draw the message between the two.
- Find the next message on the collaboration diagram. If either object at the end of the message is not already on the sequence diagram, add it with its lifeline. Draw the message below the last message drawn on the sequence diagram.
- Repeat the above step until all messages have been handled.

3.3 Sections 3.1-3.3

4 Class 8 Representing State

4.1 Questions

4.1.1 Why is state important?

4.1.2 What is state and how do we represent it?

4.1.3 When do we use state?

4.2 What is "state"?

State refers to the values of an *object's* attributes. It also includes a condition where the object satisfies some condition, performs some activity, or waits for some event.

4.2.1 Objects only

State applies to objects only. There can be static information for a class, but one does not think of a class as having state.

4.2.2 State changes

There are several ways that an object's state can change. The simplest way is to change the value of an attribute.

Realistically, most important state changes have to do with the values of many attributes that might be used to represent the object's point in time.

We indicate state changes as *transitions* from one state to another.

4.2.3 Example

Consider the RoboMower. What are the states that define the overall system? What causes the state to change (i.e., what are the transitions)? Is there just one way to represent the states for the RoboMower? If not, what are some alternatives?

4.3 State Diagrams

4.4 Sections 3.4-3.7

4.5 VP-UML exercise

Use the VP-UML model that I've put on myWPI in the Course Documents section, called "Voice Mail System.vpp" and add a sequence diagram for the Retrieve a message scenario.

This is due before the beginning of class 10.

Turnin command: `/cs/bin/turnin submit cs4233 hw4 homework4.vpp`