

## UML – Unified Modeling Language

UML is the successor to the wave of object-oriented analysis and design methods that appeared in the early '90s. It unifies the methods of Booch, Rumbaugh (OMT), and Jacobson.

UML is a modeling language, not a method. Most methods consist of both a modeling language and a process:

- *modeling language*: notation (mainly graphical) used to express designs.
- *process*: what steps to take in doing a design.

### Why do Analysis and Design?

#### I. Learning OO

It's not that difficult to learn how to program in an OO language. The problem is that it takes a while to learn to exploit the advantages that objects languages provide.

The techniques in UML, were to some degree designed to help people do good OO. CRC cards, interaction diagrams, class diagrams . . .

#### II. Communicating with Domain Experts

One of the biggest challenges in development is that of building the right system – one that meets the users' needs at a reasonable cost.

Achieving good communication, along with good understanding of the users' world, is the key to developing good software.

Many of the techniques (use cases, class diagrams, activity diagrams) used in UML help with communication about surface and detailed things.

#### III. Understanding the Big Picture

Design techniques help you acquire an overall view of the system for both outsiders and to the project team (class diagrams, interaction diagrams, package diagrams, patterns).

## Use Cases

A use case is a typical interaction between a user and a computer system.

Ex: Word processor

*Make some text bold*

*Create an Index*

From those examples you can get a sense for a number of properties of use cases:

- It captures some user-visible function
- It may be small or large
- It achieves a discrete goal for the user.

Sometimes we have to distinguish between a *user goal* and a *user interaction*.

Interaction: *define a style, change a style, move a style from one document to another.*

Goal: *ensure consistent formatting for a document, make one document's format the same as another.*

Both have their applications. Interactions are better for planning purposes; goals are important so that you can consider alternative ways to satisfy the goals.

Some recommend to focus on goals first, and then come up with use cases to satisfy them. Eventually you would have one set of system interaction use cases for each user goal.

## Use Cases

A use case is a typical interaction between a user and a computer system.

Ex: for a Word processor application

- *Make some text bold*
- *Create an Index*

From those examples you can get a sense for a number of properties of use cases:

- It captures some user-visible function
- It may be small or large
- It achieves a discrete goal for the user.

Difference between a *user goal* and a *user interaction*.

Interaction: *define a style, change a style, move a style from one document to another.*

Goal: *ensure consistent formatting for a document, make one document's format the same as another.*

Both have their applications. Interactions are better for planning purposes; goals are important so that you can consider alternative ways to satisfy the goals.

Some recommend to focus on goals first, and then come up with use cases to satisfy them. Eventually you would have one set of system interaction use cases for each user goal.

## Use Case Diagrams

### Actors

A role that a user plays with respect to the system. A user may play more than one role. Thus, it is important to think about roles rather than people or job titles. Actors carry out use cases (1 - m or m - 1).

With big systems it is often easier to arrive at the list of actors first, and then try to work out the use cases for each actor.

Actors don't need to be human. It can also be an external system that needs some information from the current system.

There usually no need to worry too much about the exact relationships between actors and use cases except in configuring the system for different kinds of users.

Some use cases don't have clear links to specific actors. Ex: the use case *send out bill*, of an utility company system.

A good source for identifying use cases is external events. Think about all the events from the outside world to which you want to react. This will help you identify the use cases.

## Uses and Extends

Use the **extends** relationship when you have one use case that is similar to another but does a bit more. When you are describing a variation on normal behavior.

- Capture the simple, normal use case first.
- Then ask "What could go wrong here?" and "How might this work differently?"
- Then add variations as extensions of the given use case.

The **uses** relationship occurs when you have a chunk of behavior that is similar across more than one use case and you don't want to keep repeating it.

When to use Use Cases: Capturing use cases is one of the primary tasks of the elaboration phase - in fact, it is the first thing you should do.

## Class Diagrams: The Essentials

A class diagram describes the types of objects in the system and the various kinds of static relationships that exist among them.

There are two principal kinds of static relationships:

- **Associations**
- **Subtypes**

Class diagrams also show the attributes and operations of a class and the constraints that apply to the way objects are connected.

There are three *perspectives* you can use in drawing class diagrams:

1. **Conceptual:** draw a diagram that represents the concepts in the domain under study. These will naturally relate to the classes that implement them, but there is no direct mapping.
2. **Specification:** now we are looking at the interfaces of the software, not the implementations. We are thus looking at types rather than classes.
3. **Implementation:** in this view, we really do have classes and we are laying the implementation bare. Probably the most used, but in many ways the specification is often a better to take.

When you draw a diagram, draw it from a single, clear perspective. To read it, make sure you know from which it has been drawn.

## Associations

Associations represent relationships between instances of classes.

**Ex**: a person works for a company, a company has a number of offices.

From the *conceptual* perspective, they represent conceptual relationships between classes.

**Ex**: an *Order* comes from a single *Customer*, a *Customer* may take several *Orders*, each *Order* has several *Order Lines*, each of which refers to a single *Product*.

Each association has a *direction* (role) and a *multiplicity*.

From the *specification* perspective, the associations represent responsibilities.

**Ex**: There are one or more methods associated with *Customer* that will tell me what orders a given *Customer* has made.

There are methods within *Order* that will let me know which *Customer* placed a given *Order* and what *Line Items* comprise an *Order*.

It also implies some responsibility for updating the relationship. There should be some way of relating the *Order* to the *Customer*.

These responsibilities do *not* imply data structure, however. From a specification level diagram, I can make no assumptions about the data structure of the classes.

The diagram indicates only the interface – nothing more.

If this were an *implementation* model, we would now imply that there are pointers in both directions between related classes.

The diagram would now say that *Order* has a collection of pointers to *Order Lines* and also a pointer to *Customer*.

```
Class Order {  
    private Customer aCustomer;  
    private Vector orderLines;
```

```
Class Customer {  
    private Vector orders;
```

In this case, we cannot infer anything from the associations about the interface. The operations on the class would give us this information.

Arrows in a Class diagram will indicate *navigability*.

In a *specification* model, this would indicate that an Order has a responsibility to tell you which Customer it is for, but a Customer has no corresponding ability to tell which Orders it has. Thus, we have responsibilities on only one side of the line.

In an *implementation* diagram, one would indicate that Order contains a pointer to Customer but Customer would not point to Order.

*Bi-directional associations* include an extra constraint, which is that the two roles are inverses of each other.

This means that every Line Item associated with an Order must be associated with the original Line Item.

## Aggregation and Composition

**Aggregation** is a part-of relationship. It is difficult to determine the difference between aggregation and association.

UML offers a stronger variety called **composition**: the part object may belong to only one whole; further, the parts are usually expected to live and die with the whole.

Fig. 5-3 indicates that any instance of *Point* may be in either a *Polygon* or a *Circle*, but not both. An instance of *Style*, however, may be shared by many *Polygons* and *Circles*. This also implies that deleting a *Polygon* would cause its associated *Points* to be deleted, but **not** the associated *Style*.

Fig. 5-4 shows another notation for composition in which you put the component inside the whole.

**Abstract** classes are indicated in italics or labeled as abstract. Figs. 5-8 and 5-9 show two alternative notations for **interfaces**.

A **Refinement** is used to indicate a greater level of detail. It can be used for implementation of interfaces.

The link between *OrderReader* and *DataInput* is a **dependency**. It shows that the OR uses the DI interface for some purpose. Essentially, a dependency indicates that if the DI interfaces changes, the OR may also have to change.

**Association classes** allow you to add attributes, operations, and other features to associations.

## Tips for Class Diagrams

- Don't try to use all the notations available. Start with the simple ones, and introduce others as needed.
- Fit the perspective to the state of the project:
  - In analysis draw *conceptual* models.
  - When working with software, concentrate on *specification* models.
  - Draw *implementation* models only when illustrating a particular implementation technique.
- Don't draw models for everything; instead, concentrate on the key areas.

The biggest danger with CDs is that you can get bogged down in implementation details far too early. To combat this, focus on the *conceptual* and *specification* perspectives.

## Interaction Diagrams

Interaction diagrams are models that describe how groups of objects collaborate in some behavior.

Typically, an interaction diagram captures the behavior of a single *use case*.

You should use interaction diagrams when you want to look at the behavior of several objects within a single use case.

They are good at showing collaborations among the objects; they are *not* good at precise definition of the behavior (use state transition, and activity diagrams)

There are two kinds of interaction diagrams: *Sequence* and *Collaboration* diagrams.

## Sequence Diagrams

Objects are shown as boxes at the top of a vertical line called the **lifeline**. Each message is represented by an arrow between lifelines. The order in which the messages occur is shown top to bottom.

A **self-delegation** is a message that an objects sends to itself.

A **condition** is used to indicate when a message is sent ( *[needsReorder]* ). The message is only sent if the condition is true.

The **iteration** marker shows that a message is sent many times to multiple receiver objects. Show the basis for the iteration within brackets ( *\*[ ]* ).

A **return** indicates the return from a message, not a new message. Show returns only when they improve the clarity.

## Collaboration Diagrams

Objects are show as icons and the sequence is indicated by numbering the msgs. More difficult to follow but the layout allows you to show how objects are linked together.

The object naming scheme takes the form: *objectName: ClassName*, where either one may be omitted.

When to use Interaction Diagrams: to look at the behavior of several objects within a single use case. They are good at showing collaborations among objects; not so good at precise definitions of the behavior.

To look at the behavior of a single object across many use cases, use a state transition diagram. To look at the behavior across many use cases or many threads, consider an activity diagram.

## State Diagrams

Technique to describe the *behavior* of a system. They describe all the possible *states* a particular object can get into and how the object's state changes as a result of *events* that reach the object.

In most OO techniques, SDs are drawn for a single class to show the lifetime behavior of a single object.

A SD shows *actions* and *activities*. Both are processes that are typically implemented by some method, but they are treated differently:

**Actions:** associated with *transitions* and are considered to be processes that occur quickly and are not interruptible.

Label: *Event [Guard] / Action*

**Activities:** associated with states and can take longer. May be interrupted by some event.

Label: *do / activity*

When a transition has no event within its label, it means that it occurs as soon as any activity associated with the given state is completed.

A *guard* is a logical condition that will return only “*true*” or “*false*”. A guarded transition occurs only if the guard is true.

## When to Use State Diagrams

SDs are good at describing the behavior of an object across several use cases. They are *not* good at describing behavior that involves a number of objects collaborating.

For such, combine them with other techniques: *interaction* diagrams for behavior of several objects in a single use case, and *activity* diagrams for sequence of actions for several objects and use cases.

Don't try to use them for every class in the system. Use them only for those classes that exhibit interesting behavior, where building the diagram helps understand what is going on.

Usually UI and control objects have the kind of behavior that is useful to depict with a state diagram.

### **Activity Diagrams**

Useful in connection with workflow and in describing behavior that has a lot of parallel processing.

The core symbol in this diagram is the *activity*. The interpretation of this term depends on the perspective from which you are drawing the diagram:

- In an conceptual diagram, it is some *task* that need to be done.
- In a specification or implementation perspective, it is a *method* on a class.

### **Deployment Diagrams**

Shows the physical relationships among software and hardware components in the delivered system.

A good place to show how components and objects are routed and move around a distributed system.