VI. The Unified Modelling Language

Use Case Diagrams
Class Diagrams
Attributes, Operations and Constraints
Generalization and Aggregation
Sequence and Collaboration Diagrams
State and Activity Diagrams

UML Diagrams

- UML was conceived as a language for modeling software. Since this includes requirements, UML supports world modeling (at least to some extend).
- UML offers a variety of diagrammatic notations for modeling static and dynamic aspects of an application.
- The list of notations includes
  - Use case diagrams -- describe how an object is to be used;
  - Class diagrams -- describe classes and their inter-relationships;
  - Interaction diagrams -- describe sequences of events;
  - Package diagrams -- describe useful groupings of an information base;
  - Activity diagrams;
  - State diagrams -- describe states and state transitions
  - …more…
Use Case Diagrams

- A use case [Jacobson92] represents “typical use scenario” for an object being modeled.
- Modeling objects in terms of use cases is consistent with Cognitive Science theories which claim that every object has obvious suggestive uses (or affordances) because of its shape or other properties. For example,
  - Glass is for looking through (or breaking)
  - Cardboard is for writing on...
  - Radio buttons are for pushing or turning...
  - Icons are for clicking...
  - Door handles are for pulling, bars are for pushing...
- Use cases offer a notation for building a coarse-grain, first sketch model of an object, or a process.

Use Cases for a Meeting Scheduling System

© 2003 John Mylopoulos
Use Cases for a Car

Driver

GasAttendant

Mechanic

Drive

FillUp

FixCar

TurnOnEngine

CheckOil

FixCarOntheRoad

Use Cases

- Use cases may represent user goals, or user interactions; for example, ScheduleMtg can be thought as a goal (there are many ways to schedule a meeting), but ValidateUser is probably not.

- Use cases make sense for usable things, such as designed artifacts, including processes; they don’t make sense for unusable things (e.g., the sky), non-physical things (the concept of number), or imaginary things (e.g., square circle).

- (Consequently) Use cases constitute a special-purpose modeling construct for software or other person-made artifacts.

- [The notion of scenario, as a typical course of actions or events, is probably more appropriate for a general purpose modeling language.]

- [Note: Scenarios have become a very popular notion for requirements acquisition and modeling.]
Features of Use Cases

- An actor is a role that a user plays with respect to the object being described; don't think of actors as either users (e.g., Maria), or positions (e.g., department chair).
- <<extends>> relationships are used when one use case is similar, but does a bit more than another. For example, you may have a use case that captures the normal case, and use extensions to describe all the variations.
- <<uses>> relationships are helpful when one wants to factor out a chunk of behaviour which is part of several other use cases.
- (When do I stop??...) For any one software development project, you probably don't want more than 100 use cases.

<<extends>> vs <<includes>>

- <<extends>> implies that one use case adds behaviour to a base case; used to model a part of a use case that the user may see as optional system behavior; also models a separate sub-case which is executed conditionally.
- <<includes>> (or <<uses>>): adds behavior to a base case (like a procedure call); used to avoid describing the same flow of events several times, by putting the common behavior in a use case of its own.
Class Diagrams

- Class diagrams describe the kinds of objects found in the application, and their inter-relationships.
- There are two types of inter-relationships: associations and subtypes [Fowler97]
- [Note: The distinction should really be between associations and abstraction relationships, such as generalization, classification, and aggregation].
- Class diagrams are basically an adaptation of EER diagrams, with some minor differences.
- UML class diagrams may model some part of the real world (e.g., the world of meetings and schedulings), a design specification (e.g., for a system that does meeting scheduling), or an implementation.

Comparison of EER and Class Diagrams

- EER diagrams allow N-ary relationships, $N \geq 2$; Class diagrams only allow binary relationships.
- EER diagrams allow multi-valued attributes, class diagrams do not.
- EER diagrams allow the specification of identifiers (an often-encountered type of constraint), while class diagrams do not.
- Class diagrams allow dynamic classification, but EER diagrams do not.
Notes on Associations

- Associations represent semantic relationships. Each association can have up to two roles for participating objects. Each role can also have an associated cardinality range ("multiplicity").

- Associations represented with directed arrows are navigable only in one direction; for example, if the Meeting-Person association was represented with an arrow pointing towards Person, this would indicate that from a Meeting we can navigate to all meeting persons, but given a person, we can't find all the meetings she has participated in.

- [Along those lines, we may want to also use bi-directional arrows to indicate full navigatability, while undirected arrows mean unknown navigatability. UML notation seems to be ambiguous on this point.]
### Notes on Attributes

- Attributes are always single-valued.
- Attributes can have an associated type (a class), a default value, and a visibility value of + (public), # (protected) and - (private). They can also be derived (/attr) or not.
- There is no semantic difference in UML between attributes and directional associations.
- In other conceptual models, there is. Sometimes models distinguish between objects and values, and only allow attributes to relate objects to values.
- Other models treat attributes as associations with an existence constraint which says: if an object is deleted, so are its attributes and their values.
- Attribute visibility is of dubious value to conceptual modeling. Of course there is a need to capture the notion of private information, but PL-type visibility is way too simple-minded for that.

### Notes on Operations

- These are “the processes a class knows how to carry out” [Fowler97, p63]. They are specified in the third layer of a class box.
- Specification includes a visibility value, name, parameter list and returned value type.
- For conceptual modeling, Fowler argues -- rather vaguely -- that operations should be used to define the responsibilities of a class.
- It makes better sense to distinguish a subclass of objects -- agents/positions/roles -- which can participate in activities, and describe for each the activities they know how to carry out.
- For example, an accountant can participate in activities such as PrepareFinancialStatement, CollectFinancialData etc. There doesn’t seem to be much use of this for inanimate objects or abstract concepts.
Operations and Constraints

<table>
<thead>
<tr>
<th>Meeting</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ time: Time = 9am</td>
</tr>
<tr>
<td>+ place: Place = LP266</td>
</tr>
<tr>
<td>+ schedule(ConstrLst):Meeting</td>
</tr>
<tr>
<td>+ cancel()</td>
</tr>
</tbody>
</table>

if place=SF2201 then time≠12pm;
if Meeting.initiator is Disabled then place is AccessibleRm;

- How do you say that "If one of the participants is disabled, then the place must be disabled-accessible??
- We need some sort of a First Order language for constraints:
  
  ```
  if (exists p:Meeting.participant) p is Disabled
  then place is AccessibleRm
  ```

Object Constraint Language (OCL)

- Some constraints can be adequately expressed in the graphical language (ex. cardinality of an association).
- Some can not. For example, constraints within operation specifications (pre- and post-conditions)
- OCL expressions are constructed from a collection of pre-defined elements and types.
- The language has a formal syntax and semantics and supplements the expressiveness of UML.

Multiple and Dynamic Classification

- Classification refers to the relationship between an object and the classes it is an instance of.
- Traditional object models (e.g., Smalltalk, C++, ...) assume that classification is single and static. This means that an object is an instance of a single class (and its superclasses) and this instance relationship can't change during the object's lifetime.
- Multiple classification allows an object to be an instance of several classes that are not is-a-related to each other; for example, Maria may be an instance of GradStudent and Employee at the same time.
- If you allow multiple classification, you want to be able to specify which combinations of instantiations are allowed. This is done through discriminators.
- Dynamic classifications allows an object to change its type during its lifetime.

---

OCL Examples

<table>
<thead>
<tr>
<th>OCL expression</th>
<th>Interpretation</th>
</tr>
</thead>
</table>
| `Person
  self.age` | In the context of a specific person, the value of the property 'age' of that person—i.e., a person's age. |
| `Person
  self.income >= 5000` | The property 'income' of the person under consideration must be greater than or equal to 5000. |
| `Person
  self.wife->notEmpty implies
  self.wife.sex = female` | If the set 'wife' associated with a person is not empty, then the value of the property 'sex' of the wife must be female. The boldface denotes an OCL keyword, but has no semantic import in itself. |
| `Company
  self.employee->size <= 50` | The size of the set of the property 'employee' of a company must be less than or equal to 50. That is, a company cannot have more than 50 employees. |
| `Company
  self.employee->select (age > 50)` | This specifies the set of employees of a company whose age is greater than 50. |
Multiple Classification

- Mandatory means that every instance of person must be an instance of Male or Female.
- <<Dynamic>> means that an object can cease to be a TA and may become a Professor.

Generalization

- Multiple generalization involves a class which has two or more superclasses that are not is-a related. For example, TA is a specialization of Student and Employee.
- In UML multiple generalization is allowed. Inheritance conflicts are resolved by predefined order of superclasses; renaming of attributes or operations is also allowed.
- For each discriminator, the associated collection of classes can be declared to be complete/incomplete, also disjoint/overlapping.
Stereotypes

- Stereotypes offer “a high level classification of an object that gives you some idea of the kind of object it is” [Fowler97].
- Stereotypes define the types of constructs that can be used in a UML diagram. You can think of them as offering a metamodel of UML diagrams, or as giving the graphical syntax of UML diagrams.
- In UML, stereotypes are shown delimited by <<...>>.
- Note that the stereotypes shown in class diagrams (such as <<uses>>, <<extends>>) are metaclasses which define the UML metamodel; in other words, these metaclasses specify what are the basic types of constructs in UML.
- One can extend UML by creating new stereotypes as specializations of built-in ones.

Stereotypes as Metaclasses

- <<metaclass>>
  - Actor
    - user
  - UML Class
    - Use Case
      - uses extends
  - Class
    - Boundary
    - Control
    - Entity
Notes on Aggregation

- Aggregation represents the partOf relationship.
- Composition is a strong form of aggregation, where a part can only participate in one composition relationship at a time. For example, an engine can only be a part of a car or a train; moreover, every engine has to be a part of a car or a train.
- Aggregation has been formalized in [Motsc Hag93]. Every aggregation relationship can be classified along two dimensions:
  - Dependent/Independent -- if an aggregation relation is dependent, then when you remove the whole you also remove the part;
  - Shared/Exclusive -- if an aggregation relationship is exclusive, then the part could not be part of several wholes.
- So, composition amounts to a dependent, exclusive aggregation relationship.
**Objects vs Values**

- Values are mathematical objects, such as numbers, tuples, lists and sets. They come with their own equality predicate so that they can be compared. Values are immutable.
- (Reference) objects, on the other hand, have equality defined by their internal identifier. This means that two processes which have been running independently can never generate the same object, but may well be using the same values.
- Some conceptual models do make the distinction, [Fowler97] appears not to.
- The presence of values can influence (positively!) the semantics of attributes.

---

**Qualified Associations**

Idea is that when you have a multi-valued role, you may have a key for all the values of that role. For instance:

- University \(\text{empl}\#\) \(0..\ast\) Person
- Computer System \(\text{userId}\) \(0..\ast\) Person

© 2003 John Mylespooulos
**Association Classes**

Association classes allow you to treat associations like classes:

![Diagram](image)

In UML you can only have a single instance of an association class for every pair of objects; this doesn’t allow, for instance, several employments of the same person by the same employer.

---

**Template Classes**

Template classes are parameterized classes; this construct is useful if you want to model groups or lists whose elements are all of the same type:

![Diagram](image)
Visibility

- Private/public attributes and operations have obvious semantics.
- Protected attributes and operations can only be used by the owner class and its specializations.
- Question is: can one instance of a class see protected attributes of another instance of the same class?
- C++ allows this, Smalltalk does not. Smalltalk is obviously right...

Object Diagrams

- These are like class diagrams, except now we model instances of the classes defined in class diagrams.
- Object diagrams are mentioned in [Gogolla98], but not in [Fowler97].
Interaction Diagrams

- Interaction diagrams capture interactions among objects.
- Typically, an interaction diagram models what happens for a single use case.
- An interaction diagram shows a number of example objects and the messages that are passed between them during the execution of the use case.
- There are two (comparable) types of interaction diagrams: sequence diagrams, and collaboration diagrams.
- Use icons to denote the objects participating in an interaction diagram (sequence or collaboration).

Sequence Diagram for ScheduleMtg

1. call()
2. what’s up()
3. give mtg details()
4. *[for all participants] inform()
5. acknowledge()
6. *[for all participants] remind()
7. giveTimetable()
8. prompt(timetables)
9. show schedule()
10. [decision=OK] scheduleOK’ed()
11. *[for all participants] inform(mtg)
**Concurrency and Synchronization**

- Some of the features of sequence diagrams are useful for modeling concurrent computer processes, rather than for world modeling.
- Statecharts are much more elegant for modeling concurrency.
- Numbering messages is optional for sequence diagrams in UML.

---

**Selecting a Course to Teach**

- ![Diagram](image)
How to Use Collaboration Diagrams

- Collaboration diagrams are intended to model scenarios; each scenario describes a possible sequence of events and actions.
- For complex use cases use several collaboration diagrams; make sure each collaboration diagram is simple and easy to understand.
- A special designation is available for objects which are created or destroyed during a collaboration:
  - Created during execution of the collaboration
    :Employee{new}
  - Destroyed during execution
    :Employee{destroyed}
  - Created and destroyed
    :Employee{transient}
### Packages

- Packages allow one to define useful subsets of a model to facilitate understanding and other modeling tasks.

- There are many criteria to use in defining the subsets:
  - Ownership -- who is responsible for which diagrams;
  - Application -- each application has its own obvious partitions; e.g., a university department model may be partitioned into staff, courses, degree programmes,…
  - Use -- clusters of classes used together, e.g., course, course description, instructor, student,…
  - Perspectives -- Maria’s vs Peter’s.
  - …
A Package Diagram

- A dependency means that if you change a class in one package, you may have to change something in the other.
- The concept is similar to compilation dependencies.
- It's desirable to minimize dependency cycles, if at all possible.

State Diagrams

- These are state transition diagrams (with some additions) which describe the lifetime of some object (a person, a student,...)
- Transitions are supposed to represent actions which occur quickly and are not interruptible. States are supposed to represent longer-running activities. What constitutes "quickly" and "longer-running" depends on the application.
- A transition can have an associated event [guard] action, all of which are optional.
- When a transition has no associated event, it fires as soon as its source state activity is done.
- State diagrams can have superstates, consisting of several states. This is basically the statechart notation [Harel87].
Events

UML distinguishes four different types of events:

- **Change events** designate when a condition becomes true
  
  *E.g.*, when(balance < 0)

- **Signal events** designate the receipt of an explicit (real-time) signal from one object to another

- **Call events** indicate the receipt of a call for an operation by an object (*request* events would be more appropriate for non-software modelling)

- **Time events** mark the passage of a designated period of time from some time point

  *E.g.*, after(10 seconds,<event>)
Course Lifetimes

Initialization

offerNewCourse/
count=0;create(CourseRoster)

addStudent[count<10]
count=10;
cancel
cancel
cancel

getStudentInfo(info);
addStudent()

Open

Closed

Canceled

[Lochovsky98]

States

- A state represents a time period in the life of an object during which the object satisfies some condition, performs some action or waits for an event.
- In general a state can be characterized by a predicate which is true while the state is “on”.
- Such a predicate may be defined in terms of:
  - The value(s) of one or more attributes of the class
    E.g., a person’s address
  - The existence of a link to another object
- (In OO folklore...) The interval between two messages sent by an object typically represents a state
Activities

- Some states represent the lifetime of an activity that takes time to complete
  - starts when a state is entered
  - either completes or is interrupted by an event that causes an outgoing transition
- An activity state can have two possible forms:
  - No specified behavior;
  - eventName [guardCondition]/action
- Special activity constructs:
  - do / stateDiagramName(parameterList) -- "calls" another state diagram;
  - entry/action -- carry out the action when entering the activity;
  - exit/action -- carry out the action when exiting.

Course Lifetimes, Again

```
addCourse/ count=0;
CourseRoster.Create()
```

```
addStudent [count<10]
```

```
cancel
cancel
```

```
courseStarted
```

```
do / FinalizeCourse
```

```
do / Initialize
```

```
do / Open
```

```
do / OpenNew
```

```
do / Closed
```

```
do / Cancelled
```

```
do / Open
```

```
entry / RegisterStudent(Student)
exit / CourseRoster.AddStudent(student)
```

```
addStudent [count<10]
```

```
CourseRoster.Delete()
```

```
```

© 2003 John Mylopoulos
Superstates

- State transition diagrams can be very hard to read once they grow to more than a few dozen states.
- For UML activity diagrams, states can be composed into superstates. Such compositions make it possible to view an activity at different levels of abstraction.
- For example, the Closed state of the last activity diagram may have its own state transition diagram which describes what happens while this state is “on”.
- There are two types of superstates:
  - OR superstate -- when the superstate is “on”, one of its component states is “on”;
  - AND superstate -- when the superstate is “on”, all of its component states are “on”;

An OR Superstate

![Diagram of an OR superstate showing states Neutral, Forward, and Reverse with transitions for select R, select N, and select F.](image-url)
**State Diagram for Purchase Order**

- **Checking**: Check order. Do: check items [all items in stock].
- **Dispatching**: Do: package items.
- **Waiting**: Item received [some item not in stock]. Item received [all items in stock].
- **Cancelled**: /cancel.
- **Delivered**: /deliver package.

**OR Superstate**

**State Diagram for (Auto) Transmission**

- **Transmission**: Neutral, select R; Reverse, select N; First, select F; Second, select N; Third, select F.
- **Forward**: Stop, upshift, downshift; First, upshift, downshift; Second, upshift, downshift; Third, downshift.
**An AND Superstate**

- **Check**
- **Dispatching**
- **Authorized**
- **Waiting**
- **Cancelled**
- **Delivered**
- **Rejected**

**Complex State Diagram Transitions**

- Transition to superstate boundary $\equiv$ transition to initial state of the superstate.
  - entry actions of all regions entered are performed
- There may also be transitions directly into a complex state region (like program "gotos").
- Unlabelled transition from a superstate boundary $\equiv$ transition from the final state of the superstate
  - exit actions of all regions exited are performed
- Labelled transition from a superstate boundary means transition from any one substate.
- There may be transitions directly from within a complex state region to an outside state.
**Bridge Vulnerability Rules**

**Playing Bridge Rubber**

**N-S vulnerability**

- Not vulnerable → N-S game → Vulnerable → N-S game → N-S wins

**E-W vulnerability**

- Not vulnerable → E-W game → Vulnerable → E-W game → E-W wins

**Taking a Course**

**Taking a Course**

**Incomplete**

- Lab1 → lab done → Lab2

**Passed**

- Term Project → project done

- Final Exam → pass → Passed

- Final Exam → fail → Failed

© 2003 John Mylopoulos
**Auto Transmission**

**Ignition**
- Off to Starting: turn key to start (Transmission in Neutral)
- Starting to On: release key
- On to Off: turn key off

**Transmission**
- Neutral to Reverse: select R
- Neutral to First: select N
- First to Second: upshift
- Second to Third: upshift
- Third to Neutral: downshift

**Activity Diagrams**

- Like Petri nets, activity diagrams allow transitions with several input and output states:

**Before**

**After**
An Example

The Process Order use case:

“When we receive an order, we check each line item on the order to see if we have the goods in stock. If we do, we assign the goods to the order. If this assignment sends the quantity of those goods in stock below the reorder level, we reorder the goods. While we are doing this, we check to see if the payment is O.K. If the payment is O.K. and we have the goods in stock, we dispatch the order. If the payment is O.K. but we do not have the goods, we leave the order waiting. If the payment is not O.K., we cancel the order.”

Another Example: Order Processing
Order Processing Activity Diagram

Activities

- An activity state represents an action in the execution of the activity. An activity state normally contains an action expression and usually has no associated name.
- Actions may be described by:
  - Natural language
  - Structured English
  - Pseudo-code
  - Programming language
  - Another activity diagram
- An action expression may only use attributes and links of the owning object.
More About Activity Diagrams

- **Decision points:**

  ![Activity Diagram](attachment:image.png)

- **Dead ends:** There may be transitions in an activity diagram with no destination state; this can mean that:
  - Not all processing has been specified,
  - Or, that another activity diagram will take over.
More Swimlanes

More Swimlanes

Modeling with UML

(For comparison purposes) To model with UML you need to answer the following kinds of questions:

- What are the users doing? -- use cases (Jacobson)
- What are the objects in the real world? (Rumbaugh)
- What objects are needed for each use case? (Jacobson)
- How do the objects collaborate with each other? (Jacobson, Booch)
- What are allowable sequences of actions and activities?
- What are allowable lifelines for objects?
- [How will we implement real-time control? -- state models]
- How are we going to build this system? (Booch)
Conclusions

- UML amounts to a combination of EER diagrams, statecharts and other diagrammatic notations (also some stardust).
- Much of this is useful for conceptual modelling. Some diagrams, however, (such as deployment diagrams, not discussed in these notes), are appropriate for software modelling only.
- The great contribution of UML is that for the first time ever, there is a modelling standard; this has led to compatibility and portability of conceptual models.
- The great weakness of UML is that it was designed by committee, and some design decisions were based on political, rather than technical considerations.
- Another major weakness of UML is that it is (still largely) informal. Informal models are not scalable!

References