LECTURE 18: UML

Software Engineering

Mike Wooldridge

1 What is UML?

- In the mid 1980s, a number of techniques began to emerge for *object-oriented-analysis and design*.
- Examples:
  - Coad & Yourdon;
  - Booch;
  - Rumbaugh (the OMT technique);
  - Coleman (FUSION).
- All used similar techniques, but differed on details of notation, etc.
- Mid 1990s: a move towards standardisation, driven by the *Object Management Group (OMG)*:

  The Unified Modelling Language (UML).

- UML is essentially a *notation*, and *not* a technique.

  Notation can be used in many different ways: we show one.

2 UML Models

- UML provides a rich graphical notation for developing a series of *system models*.
- These models become increasingly less abstract, and more detailed.
- The models we discuss are

  Analysis:

  – use cases;
  – conceptual model;

  Design:

  – class model;
  – interaction and collaboration model;

3 Use Cases

- Use cases are a narrative + graphical document that describes the sequence of events of an *actor* using a system to achieve some particular goal.
- Use cases document system behaviour *from the actor's point of view*.
- By "actor" we mean either person interacting with system, or some other system.
- Use cases are useful in requirements capture and validation.

3.1 Schema for Use Cases

- *Use case*:

  [name of use case]

- *Actors*:

  [list of actors that can participate, naming the initiator, and indicating key player]

- *Purpose*:

  [one-line summary of *purpose* of the activity]

- *Overview*:

  [short summary of the use case]

- *Type*:

  [primary or secondary, essential or optional]

- *Cross references*:

  [references to requirements document]

- *Course of events*:

  [narrative summary of use case]

3.2 Example Use Case 1

- Use case:

  Buy items with cash.
- Actors:

  Customer (initiator), cashier
- Purpose:

  Capture a sale and its cash payment
- Overview:

  Customer arrives at checkout with items to purchase in cash. Cashier records the items and takes cash payment. On completion, customer leaves with items.
- Type:

  Essential, primary.
- Cross references:

  Buy items with credit card, buy items with cheque.

- Course of events:

1. Customer arrives at checkout with items.
2. Cashier records identifier of each item.
3. As each item is recorded, system responds with running total of cost of items.
4. Cashier indicates to system that there are no more items.
5. System responds with overall total cost of items.
6. Cashier takes cash payment from customer for total cost, and indicates this to system.
7. System prints receipt for amount tendered.

- We use *use case diagrams* to document the participants in use cases:

- We capture a number of use cases in a single use case diagram:

- Identifying use cases:
  - identify the actors involved in a system or organisation;
  - for each actor, identify the processes they initiate or act in;
  - refine processes into use cases;
- Common mistakes with use cases:
  - making them too small;
  - identifying *parts* of activities, rather then entire activities.

4 Conceptual Models

- The next stage in UML development involves identifying the key *concepts* in the system, and documenting the relationships between these concepts.

  The resulting *conceptual* model will evolve into the *object* model.

- A conceptual model documents:
  - the concepts in a system;
  - relationships between concepts;
  - attributes of concepts.

- Conceptual models are *not* design models.

  So avoid such concepts as:
  - "database";
  - "GUI" or "window";

- Key distinguishing feature of OO development:

  Understanding system in terms of concepts rather than functions.

- What are candidates for concepts?
  - physical or tangible things
    e.g., receipt, plane
  - specifications, designs, or descriptions of things;
    e.g., product specification
  - places
    e.g., airport, point of sale terminal
  - transactions
    e.g., deposit, withdrawal
  - roles of people
    e.g., casher, customer
  - containers of things
    e.g., plane, store room
  - things in a container
    e.g., passenger

- other systems
  e.g., www site
- abstract noun concepts
  e.g., hunger
- organisations
  e.g., sales department
- events
  e.g., robbery, death
- processes (sometimes)
  e.g., deposit
- rules and policies
  e.g., refund policy
- catalogues
  e.g., parts catalogues
- contracts & legal documents

4.1 Relationships

- In addition to recording the concepts, we must record the *relationships* between concepts.
- For example, the concepts *student* and *course* may be related by *is registered on*, i.e., a student is registered on a course.
- *Generalisation* is a special type of relationships where one class is a *subclass* of another.

Common types of relationship:

- is a physical part of;
  e.g., wing-plane
- is a logical part of
  e.g., module-course
- is physically contained in
  e.g., passenger-plane
- is logically contained in
  e.g., flight-flight schedule
- is a description for
  e.g., flight description-flight
- is reported/recorded in
  e.g., reservation-flight manifest

- is a member of
  e.g., pilot-airline
- is an organisational subunit of
  e.g., department-store
- uses or manages
  e.g., pilot-plane
- communicates with
  e.g., customer-cashier
- is related to a transaction
  e.g., passenger-ticket
- is owned by
  e.g., plane-airline

4.2 Concept Diagrams

- Concepts and the relationships between them are documented in a *concept diagram*.
- Basic notation for concepts:

Name of concept
Attribute 1 . . . Attribute n

- Example:

- *Concepts*:
  student, course, module
- *Relationships*:
  registered-on, has
- Thus:
  - *students are registered on a course*;
  - *a course has modules*.

4.3 Annotating Concepts

- We can *annotate* concepts to make relationships more precise.
- Involves stating *multiplicities*.

- Thus:
  - *many students are registered on one course;*
  - *one course has many modules.*

- Possible annotations:

1	one
n	exactly n
$m..n$	between m and n
$*$	zero or more
m, n	either m or n

- Note that identifying concepts is much more important than relationships or multiplicities;

4.4 Concept Attributes

- Attributes should correspond to *simple* data types.
- Common attributes:
  - address;
  - colour;
  - phone number;
  - serial number;
  - universal product code/barcode;
  - postal or ZIP code;
- Attributes should *not* be *composite*!
- Bad attributes:
  - URL
    is composed of three things
  - flight destination
    is an airport — a separate concept altogether;

4.5 Generalisations

- UML includes a special symbol for generalisations.

- Thus:
  - *rectangle* is a generalisation of *square*;
  - *shape* is a generalisation of *rectangle*.

- The generalisation relationship is actually the *subclass* relationship.
- Thus rectangle is a subclass of shape, and square is a subclass of rectangle.
- A subclass *inherits* all the attributes of its superclass.
- The generalisation relation is *transitive*: thus square is also a subclass of shape.
- Generalisation can prevent proliferation of similar classes ... but use it sparingly!

4.6 Finding Concepts

- We find concepts by looking for nouns:

  The library contains books and journals. It may have several copies of a book. Books are written by authors. Some books are for short term loan only. All other books may be borrowed by any library member for up to three weeks. Members of the library can normally borrow up to 6 items at a time, but members of staff may borrow up to 12 items at a time.

- Avoid concepts that are:
  - trivial (attributes);
  - redundant (duplicated concepts);
  - vague;
  - part of the meta-language;
  - outside the scope of the system;

- The library conceptual model:

- Missing concepts?

5 Collaboration Diagrams

- The conceptual model of a system will evolve into an *object model*, which captures the *static* aspects of a system.
- To capture the dynamics of a system, we need a *collaboration diagram*.
- A collaboration is:
  - a collection of *linked* objects;
  - which work together to achieve a task;
  - by invoking methods on each other.
- A collaboration captures stereotypical *control flows* (method invocations) between collaborating objects.
- NB: At least one collaboration diagram for each use case.

5.1 Example Collaboration Diagram

Intuition:

1. Person (borrower) begins by asking to borrow a copy of a book.

   This corresponds to a *method* `borrow(theCopy)` being invoked on the corresponding `libraryMember` object.
2. The library member object then asks *itself* whether it is allowed to borrow a book.
3. The `LibraryMember` object then invokes the `borrow()` method on the `Copy` object corresponding to the copy of the book that the user requests.

Note that:

- Boxes correspond to objects.
- Names inside boxes are the class of the object.
- Arrows correspond to method invocations.
- The numbers that label method invocations indicate the order of events.
- If an object is invoked by invocation number `x` then its method invocations will be numbered `x.1`, `x.2`, and so on.

- An object cannot "spontaneously" invoke a method: it can only invoke methods *in response* to methods being invoked on it.
- In order for an object `o1` to invoke a method on another object `o2`, it must *have a handle on it*:
  - `o1` may have `o2` as an instance variable;
  - `o1` may have been passed `o2` as an argument.

  There are clear implications for the *implementation* of objects.

5.2 Object Creation

- How to indicate *creation* of an object:

- Equivalent of following Java code:

```
class Library extends Object {

  ...

  void newLibraryMember(details : Details) {

    LibraryMember newLibraryMember =
      new LibraryMember(details);

    addNewMember(newLibraryMember);

  }

  ...

}
```

5.3 Multiple Invokations

- We can indicate that a method is invoked on an object *multiple* times by the "*" operator:

  indicates that `method()` is invoked on a `ClassB` object multiple times.
- We also have a `for` loop like notation:

  indicates the invocation of `method(1)`, then `method(2)`, up to `method(10)` on `ClassB` object.

5.4 Conditional Branching

- We can also indicate conditional branching:

- Condition goes in square brackets
- Sequence numbers `a`, `b` etc indicate the possible actions corresponding to conditions.
- Thus: if the condition `isStaff` is true, we create a `StaffMember` object, otherwise we create a `LibraryMember` object.

6 Sequence Diagrams (Swimlanes)

- An alternative to collaboration diagrams is sequence diagrams:

- The y axis is time;
- Dotted line indicates the lifeline of an object
- Horizontal arrows indicate method invocation;
- Vertical blocks indicate periods of object activity.

7 Statecharts

- Statecharts, or state diagrams, illustrate the behaviour of an object — its *lifecycle*.
- UML uses a finite state machine-like model of statecharts:
  - directed graph;
  - nodes correspond to states of the system;
  - arcs correspond to events that cause state changes;
  - start state:
  - end state:

- We can associate *guards* with transitions; these are conditions that must be true if the state transition is to occur.

8 Class Diagrams

- The final UML model we look at is the *class diagram* or *object model*.
- This model is an *elaboration* of the *conceptual model*.
- How to construct a class diagram:
  1. using concept model and collaboration diagrams, identify the classes in your system;
  2. using collaboration diagrams, name the methods;
     (if you invoke `m` on an object `o` of class `C`, then `C` must provide method `m`)
  3. determine visibility of methods;
  4. determine *navigability*.

- An example class diagram.

- The third compartment in concept boxes contains *methods* that the class corresponding to the concept must perform.
- These methods are derived by studying the messages that are sent to the class in concept diagrams.
- These methods are annotated as follows:

  + means *public*;

  - means *private*;

  # means *protected*.

8.1 Generating Code

- The transformation of class diagrams is now straightforward:

  From class diagrams…

  – attributes become instance variables;

  – methods become methods!

- One final step involves determining *navigability*.
- If there is a relationship between C_1 and C_2, then C_1 may need an instance variable corresponding to record this.
- Example: `Book` class has relationship to `Author`.

  So may want instance variable `Author` in `Book` class.