

## Essentials of UML

### Introduction

This part of the course fills in the essentials of UML - some remaining details will be filled in later. As before we follow the textbook quite closely.

### Some general points

- see section 4.3 of text re the terms 'System', 'design', 'model' and 'diagram', and how they are used in software development.

- Discussion of representations of aspects of the design, especially the high level architecture. Notion of distinguishing models on different "axes":

- The *use case model* - users' point of view
- A *static model* - elements of system & their relationships (e.g. class models of library)
- A *dynamic model* - behaviour of system over time (e.g. interaction diagrams, state transitions)

- Helpful to have different points of view:

- *logical*: what parts notionally belong together (e.g classes)? -> functional requirements met?
- *process*: what threads of control are there (synchronisation? concurrency? etc) -> performance and similar requirements met?
- *development*: What parts can be developed by same team, what can be re-used -> Help manage project
- *physical*: Which parts will run on same computer etc

- We must have consistency between different diagram types, which is where a good, reliably correct tool is invaluable.
- A modeling language (e.g. UML) is a way of expressing the various models produced during development. Often diagrammatic but could be text based.

- UML is claimed to match up well to following desiderata in a modeling language:

- 1) Expressive enough to adequately represent relevant aspects of the design, and able to reflect change easily
- 2) Easy enough to use
- 3) Unambiguous
- 4) Supported by suitable tools
- 5) Widely used (!)

### Preliminary discussion of Design by Contract 1 (PANEL 3.1 of text)

In the state diagram of the introductory case study we saw how "conditions" may impact on system behaviour. This is addressed further in the referenced "panel" of the text book. [Topic is related to "**syntactic**" and "**semantic**" checks on interfaces].

In reading this panel, we need to distinguish between (a) good design practices and (b) how they end up being implemented in code - for (b) much depends on what is available in the chosen programming language.

- **Types** in a programming language are a rough way to describe the properties which an object's attributes and operations should have. For example, in the sequence diagram of the introductory case study, if the message `okToBorrow` always returned the value `false` then that would be of the correct Boolean type but would never allow a copy to be borrowed. If possible, we would *like* to know something more precise than the type.

- **When trying to describe the provided interface of a module**, we can try to tie it down roughly first using type definitions and, if that is not precise enough, we can add constraint(s) to express additional matter. By constraints we mean

- operation preconditions & postconditions, and
- class invariants.

### - preconditions & postconditions:

Example	Commentary
<pre>okToBorrow pre: numberOfBooksOnLoan ≤ maxBooksOnLoan post: if numberOfBooksOnLoan &lt; maxBooksOnLoan return True Otherwise return False</pre>	<p>It is a "bug" to call this operation if the precondition is <code>False</code></p> <p>It is a "bug" if this postcondition ever turns out to be unsatisfied after the operation was called with the precondition <code>True</code></p>

Note that more precise notation may be needed if handling many such conditions. The "**Object Constraint Language**" (OCL) is such a notation and is commonly used with UML. We will return to OCL later in the course.

- **class invariants**: In our last example, the precondition does not just apply to `okToBorrow` but, in fact, it is always a "bug" if the precondition is `False`. Therefore, it is more appropriate to make it a class invariant, that is, put it as documentation of the `LibraryMember` class that a valid object of that class must always have `numberOfBooksOnLoan ≤ maxBooksOnLoan`.

- **Implementation issues:** The definition of constraints during design is good practice. However, the extent to which support is provided for checking such constraints in the implemented code is not great.

-- Many programming languages support *compiler-time* type checking but compiler-time constraint checking is more problematic - Eiffel is noted in the text as one language that does provide such checks.

-- **Assertions** are noted in the text as being provided by some languages - these are Boolean expressions (= conditions) that can be checked at *run-time* (though possibly only in "debugging mode").

-- Some languages allow declaration of subtypes, a facility that may be useful in implementation of constraints. For example, in Ada, we could declare

```
numberOfBooksOnLoan: INTEGER range 0 .. maxBooksOnLoan;
```

Then a CONSTRAINT\_ERROR would be raised if ever

```
numberOfBooksOnLoan had a value outside its range.
```

Note: Text book has another panel (3.2 Persistence) on how to make it possible to shut down and re-start the system without losing information. We omit this for now.

Presentation of Chapter 5 of text (**Essentials of class models**)

&

Brief Preview of Chapter 6 of text (**More on class models**)

Note: Useful to bear in mind =>

	Instance
Class	Object
Association	Link
Use case	Scenario

• 5.1 Identifying objects and classes

**5.1.1 What makes a class model good?**

Objectives (of a class model)	How to meet objectives
Build [, as quickly and cheaply as possible,] a system which meets our current requirements	Every required piece of behaviour must be able to be provided [, in a sensible way,] by objects of the classes we choose
Build a system which will be easy to maintain and adapt to future requirements	A good class model consists (as far as possible) of classes which represent <b>enduring</b> classes of domain objects, which don't depend on the particular functionality required today

**5.1.2 How to build a good class model**

- No hard and fast rules but there are useful guidelines and techniques
- Usually won't get class model right at start - expect some adjustments
- Can expect to identify the important classes of **domain** objects early on but identification of other classes may be harder.

- We present two techniques for identification of classes:

<u>noun identification</u> (for data driven design)	<u>CRC cards</u> (for responsibility driven design)
<b>Extreme caricature:</b> identify all data in the system and divide them into classes. Only after that, consider class responsibilities.	<b>Extreme caricature:</b> identify all responsibilities in the system and divide them into classes. Only after that, consider class data.

In practice, best to use a mixed approach rather than just one technique.

- noun identification: **We described this already** - it involves

- (1) Identify candidate classes by underlying noun and noun phrases in a requirement specification
- (2) Discard candidates that are inappropriate for any reason. E.G.

Discarded Candidate	Reason for discarding
library	outside scope
loan	event, not a thing [ <i>maybe not that obvious a discard!</i> ]
Member of library	redundant (same as something else)
item	Too vague
week	A measure, not a thing
system, rules	Not part of domain

The text adds some more guidelines/detail on discarding: (**see CRC also**)

**outside scope:** noun does not refer to something inside the system

**event (or operation):** If an instance of the event or operation does not have (see \*\* below) identity, state and behaviour should discard it as a candidate class

**redundant:** Choose a class name to encompass all you mean to include

**vague:** can't tell unambiguously what is meant by a noun

**Not part of domain (meta-language):** part of how we define things

**An attribute:** A noun refers to something simple which is an attribute of another class (e.g. "name" of library member on our case study)

\*\* (summary)

IDENTITY: Notion that an object has a continuing existence

STATE: All the data an object currently encapsulates - the current values of its attributes

BEHAVIOUR: The way an object acts and reacts, in terms of its state changes and message passing.

- CRC cards: We will describe this method shortly (corresponds to section 5.6 of text).

**5.1.3 What kinds of things are classes?**

The text notes that objects and their division into classes derive from one of the following (over-lapping) sources:

Source	
Tangible or 'real-world' things: book, copy, course	<i>Much more common sources of objects and</i>
Roles: library member, student, director of studies	<i>classes than following two</i>
Events: arrival, leaving, request	<i>Often help to find</i>
Interactions: meeting, intersection	<i>associations</i>

**Remark:** Here are some suggestions on discovering classes from a different author (Budd, *An introduction to object-oriented programming*):

- (a) Data managers, data, state - after the nouns in a problem description - the fundamental blocks
- (b) Data sinks & sources - do not hold data for a period of time (like (a) does) - generate data or process data on demand
- (c) View or observer classes - one often has base data (the model) and corresponding display (view)
- (d) Facilitator or helper classes (could include common utilities such as specific data structures (and their associated attributes & operations), math 'packages', etc)

**5.1.4 Real world objects Vs their system representation**

In our introductory case study, we adopted one approach for representing **actors** (in UML speak) within our system design. We will come back to this point when discussing *use cases*.

In section 5.1.4, the text makes two general points on the representation of real-world things within our software system:

1	Do not record information that is definitely irrelevant to the system
2	Do not lose sight of the fact that the objects <b>are</b> the system

Point 2 is important: Essentially object oriented design and programming involves the "creation of a host of helpers in the solution of a problem".

*This is in contrast to an extreme top-down, functional decomposition approach. In such a system, one sometimes (often?) had a single component that "knew" about everything and embodied all the interesting behaviour.*

In our system we will still probably need a "main" class to provide the entry into the system. Starting the system will automatically create an instance (the only instance) of this class and this will go on create other objects of the system. However, this main object should not usually have any complex behaviour of its own.

**• 5.2 Associations & 5.3 Attributes and operations**

- We saw that classes correspond to nouns and, similarly, *associations correspond to verbs*.

- **Definition:** Class A and class B are associated if some object of class A **has to know about** some object of class B (or subclasses of B for which substitutivity - see later in this section - holds), i.e. if one of following holds:

- An object of class A sends a message to an object of class B
- An object of class A creates an object of class B
- An object of class A has an attribute whose values are objects of class B or collections of objects of class B
- An object of class A receives a message with an object of class B as argument

- **Note (see section 6.4 of text):** Recall that "A depends on B if a change in B may force a change in A. ... . Notice the difference between a dependency between two classes and an association between them. An association between two classes represents the fact that **objects** of these classes are associated. A dependency is **between the classes** themselves, not between the objects of those classes ... for example, a class always depends on a class from which it inherits." The UML terminology can be a bit confusing (dependency, relationship, association) and we will try to make it clear in the rest of this part of the notes.

- Text emphasises (pp61, 62) that *throughout the development*, one should aim to develop a model which is good in both aspects:

- conceptual - reflecting real world associations between objects
- implementation - modeling interactions to achieve required functionality

**- Review of some basic elements of UML class diagrams:**

-- **Class icons** (simplest): rectangular boxes, each with its class name

Can be extended to depict the operations and attributes of a class. For example,

Book	<i>Name is in 1<sup>st</sup> (top) compartment</i>
title: String	<i>Attributes (2<sup>nd</sup> compartment) (with types)</i>
copiesOnShelf(): Integer	<i>Operations: (3<sup>rd</sup> compartment) - <u>signature</u> of each op. (selector, arguments, return type)</i>
borrow(c:Copy)	

-- **Associations** (simplest): straight lines between class icons concerned  
Can be extended by adding various annotations including, in particular, **label** (name of association), **arrows** (navigability), **multiplicity** (see next).

**Multiplicities:** One may have an **exact number**, a **range of numbers** (2 dots between a pair of numbers), an **arbitrary, unspecified number** (denoted \*), or a **combination of these** (comma separated list). The text (p63) gives the example

**[3, 12..15, 901..\*]** meaning there can be, of the items concerned, **exactly 3 or else between 12 and 15 inclusive or else at least 901**

-- Diagrammatic representations of classes and associations will evolve as a project proceeds, with more detail being added as appropriate. For example, initially one may not show navigability of an association. Similarly, class attributes and operations will emerge as responsibilities and interactions are analysed. One should avoid prematurely adding attributes that imply a particular implementation decision.

-- **Guideline to determine class operations and attributes:**

**Initially**, identify the data conceptually associated with an object & what messages it seems reasonable it should understand (*If a book could talk, what questions would you expect it to answer!*). **Subsequently**, we have to check that we have included enough data and behaviour for the requirements at hand. For this, we have to start looking at how the objects work together to satisfy the requirements (see CRC cards, below)

• **5.4 Generalization**

- We had an example of this in our introductory case study, namely that `LibraryMember` is a generalization of `MemberOfStaff`, and we saw how to represent it on a UML class diagram (*arrow directed from special to general*). Equivalently, we may say that `MemberOfStaff` is a specialization of `LibraryMember`.

From our example we see that

Needed in <u>generalization</u>	Comment
Object of class <code>MemberOfStaff</code> should conform to the interface given by <code>LibraryMember</code>	If some message is acceptable to any <code>LibraryMember</code> it must also be acceptable to any <code>MemberOfStaff</code>
A <code>MemberOfStaff</code> 's interface may be strictly broader than <code>LibraryMember</code> 's	<code>MemberOfStaff</code> may understand other, specialized messages which an arbitrary <code>LibraryMember</code> might not be able to accept

Rule **B** is concerned with where the subclass overrides any of the superclass's methods - to make sure "substitutivity" is preserved we require

- *Demand no more*: The precondition of the subclass method must be no stronger than that of the superclass

- *Promise no less*: The postcondition of the subclass method must be at least as strong as that of the superclass.

**Discussion point/Homework (p67 of text)**: "In your programming language, if a subclass overrides an attribute or operation from a superclass, is it allowed to change their types? In what way?"

- Note on 5.4.1: English can be misleading re "is a" relationships:

We know that C is probably a generalization of B if "every B is a C". However, sometimes in English "is a" can be used misleadingly and so can suggest a generalization that is not actually there. The text gives the example

Misleading	Need to be precise (pedantic?)
"A Border Collie is a breed" <i>(grammatically wrong)</i>	"Border Collie is a breed" <i>(grammatically correct)</i>
Conclude (falsely) that "breed is a generalization of Border Collie" (i.e. "breed" is a kind of dog!)	

Guided by this example, the text formulates a general design rule:

<b>A</b>	An object of a specialized class can be substituted for an object of a more general class in any context which expects a member of the more general class, but not the other way round
----------	--

The text gives a further design rule (guideline) about the design of classes where one is a specialization of the other:

<b>B</b>	There must be no conceptual gulf between what objects of the two classes do on receipt of the same message.
----------	---

This is a bit informal but what is in mind can be seen from our case study, operation `borrow(c:Copy)`. We expect (*and any client object should be able to assume*) that the way this operation is carried out by a `MemberOfStaff` should be comparable to how it is carried out by a `LibraryMember`. Could be more precise by requiring the ways to be identical but this would be a good deal more restrictive (*library example?*).

- The text includes **Panel 5.1 Design by Contract 2: substitutivity** to further examine the issues raised in rules A and B, above.

- As an object of a subclass is supposed to be usable anywhere that an object of the superclass is usable, we say that *the subclass should fulfill the contract entered into by the superclass*. This means that (cf rule A)

- a subclass could have extra attributes and operations but it cannot drop any of those of its superclass

- if a superclass has a class invariant (e.g. "a valid `LibraryMember` must always have `numberOfBooksOnLoan ≤ maxBooksOnLoan`"), then the subclass must have a class invariant that is "contained within" the class invariant of the superclass (or that is "at least as restrictive" as the superclass invariant).

- Re 5.4.2 Implementing generalization: inheritance:

A common way of implementing generalization is through inheritance (as can be done in Java and C++).

At one time inheritance was considered by some people to be a really powerful mechanism ("*a silver bullet!*"), but there are drawbacks one needs to be aware of:

1. [*Fragile base class problem*] A subclass is dependent on its superclasses, so using inheritance tends to increase the degree of tight coupling in a system. The consequences may be that if a superclass is changed it will probably be necessary (a) to recompile its subclasses or even (b) to change their code. In case of (b), particularly, it will be necessary to redo tests. Also, there are implications for configuration control - keeping track of versions of source, object and executable code of the various classes.

2. [*Amount of testing of subclass*] If one is confident that a superclass is correct, can one reduce appreciably the amount of testing needed on a subclass? In fact, the answer may be "no" - we will return to this point (*time permitting*).

Because of these problems, it is recommended to use the following:

**Guideline**: Use inheritance between classes only when it models a conceptual generalization relationship (that is, essentially, a relationship reflective of the real-world to which our system applies).

The temptation (contrary to this guideline) is to apply inheritance as a clever programming mechanism to make use of existing work, regardless of whether there is a significant or lasting connection between entities. The text-book offers the example of where a class `List` is available and we want to create a class `AddressBook` in which the addresses are stored in a `List`. We have two solutions:

Poor solution (inheritance)	Better Solution (composition)
Make <code>AddressBook</code> inherit from <code>List</code>	Make <code>AddressBook</code> own a <code>List</code> (by having, for example, an attribute <code>addresses: List</code> )

**Homework:** To assess the solutions,

- (i) Compare effort required in initial development of the `AddressBook`
- (ii) Compare the changes required if the implementation or interface of `List` changes
- (iii) Compare the changes required if it decided to use a different class instead of `List` (`Dictionary`, `say`).

• 5.5 Evolution of the class model during development (RECAP)

Start	Repeat as necessary	Finish
Record conceptual relationships & capabilities (implementation)	Add more detail (including of implementation) Introduce new operations more specific associations attributes	Complete & consistent design

• 5.6 CLASS RESPONSIBILITY COLLABORATION cards [an RDD technique]

- Technique to help in checking for a good design and refining it

- Cards format: (e.g. (14/10/04) <http://www.softstar-inc.com/Methodology/CRCIntro.htm>)

Name of class	
Responsibilities of class	Collaborators of class

*Responsibilities describe at a high level why the class exists. Related to operations but more general, to begin with at least No more than 4; else split up?*

*Collaborators are other classes which help to carry out responsibilities*

**Too many responsibilities->low cohesion/Too many collaborators->high coupling**

- Examples:

LibraryMember	
Responsibilities Maintain data about copies currently borrowed Meet requests to borrow and return copies	Collaborators Copy

Copy	
Responsibilities Maintain data about a particular copy of a book Inform corresponding Book when borrowed and returned	Collaborators Book

Book	
Responsibilities Maintain data about one book Know whether there are borrowable copies	Collaborators

- CRC cards usage can help to identify collaborators and hence associations; also, navigability directions.  
- Can use CRC cards to "walk through" the various scenarios of use cases in order to see how the class model provides the required functionality.  
- Using CRC cards as above should help to identify missing bits, problems, etc  
- As a result may have to modify responsibilities, collaborators, create new classes etc

- Points to bear in mind:

- a) Use of CRC cards may help in deciding to retain or reject classes suggested by noun identification technique.
- b) As one (or, better, a team) uses CRC cards, look at the wider implications of any change. Don't fix just a local problem
- c) Idea is to write modifications on CRC cards as one proceeds - eventually will have to write out cards freshly again (obviously)
- d) Notice that the system *actually works by means of objects* (not classes); so need to be clear whether a CRC card always represents the same object of a class, or whether it represents several objects.
- e) Use of CRC cards in a team can have benefits in terms of achieve a common understanding of the system, improved team spirit etc
- f) Nice suggestion is to pile cards related by generalization together, with most abstract at the top, so that specialization is only used when relevant.

Note: For another author's view on CRC cards, and on responsibility driven design [RDD] in general, Budd's book (cited earlier) has:

- 1) Main aspect is delegation of responsibility
- 2) Responsibility entails (a) an expectation of certain behaviour according to certain rules and (b) a degree of independence or non-interference (-> want to cut links between different parts as much as possible)
- 3) Establish who is responsible for each action that is to be performed
- 4) During design, distinction between class and instance (object) is blurred
- 5) Importance of good class names (Pronounceable/ Capitalization or underscores to mark each new word in a name/ Make sure abbreviations are understood by all/ etc)
- 6) Responsibilities should be short verb phrases - they describe what is to be done, the problem to be solved)
- 7) If card size is insufficient it probably means class is too large & complex
- 8) Collaborators (a) must include classes from which class needs services, and (b) possibly classes to which class provides services

- 5.6.4 Refactoring

Process of altering the class model of an object oriented design without altering its visible behaviour. A refactoring step might require moving an operation between classes and making all associated changes.

-- Something to watch for, in particular, is whether two classes have overlapping responsibilities and behaviour. If so, might be best to put the common behaviour into a new superclass from which both could inherit. [But bear in mind potential difficulties with inheritance noted before]. Notice that the new superclass would not have shown up in the requirements specification - it arises from design considerations and not from the (problem) domain.

• Very brief review note on Chapter 6: More on Class Models: We skip this chapter & come back to it later. As the authors say features considered are "less central than those described in Chapter 5 ...". For completeness, we list some of the topics considered:

6.1 More about associations	Distinguishing types of associations/ Navigability/ Adding fine details/ Constraints etc etc
6.2 More about classes	Interfaces (say, to specify some operations that are visible outside a class)/ Abstract classes
6.3 Parameterized classes	Sometimes called a <i>template</i> - e.g. " <code>List(T)</code> " might describe, given a class <code>C</code> to substitute for the formal parameter <code>T</code> , the class of lists of <code>C</code> objects"
6.4 Dependency	Mentioned this (short) section earlier
6.5 Components & Packages	Typically, a package is a collection of related classes and dependencies between them. Convenient for purpose of work allocation, forming a component, etc.
6.6 Visibility, Protection	