LECTURE 3: SOFTWARE DESIGN

Software Engineering

Mike Wooldridge

1 Design

- Computer systems are not monolithic: they are usually composed of multiple, interacting *modules*.
- *Modularity* has long been seen as a key to cheap, high quality software.
- The goal of system design is to decide:
  - what the modules are;
  - what the modules should be;
  - how the modules interact with one-another.
- In the early days, modular programming was taken to mean constructing programs out of small pieces: "subroutines".
- But modularity cannot bring benefits unless the modules are *autonomous*, *coherent* and *robust*.

Procedural Abstraction

- The most obvious design methods involve *functional decomposition*.
- This leads to programs in which procedures represent distinct logical functions in a program.
- Examples of such functions:
  - "Display menu";
  - "get user option".
- This is called *procedural abstraction*.

Programs as Functions

- Another view is *programs as functions*:

$$\begin{array}{ccc} input & & output \\ x \rightarrow & f & \rightarrow f(x) \end{array}$$

  The program is viewed as a function from a set I of legal inputs to a set O of outputs.
- There are programming languages (ML, Miranda, LISP) that directly support this view of programming.
- Well-suited to certain application domains — e.g., compilers.
- Less well-suited to distributed, non-terminating systems — e.g., process control systems, operating systems like Win95, ATM machines.

2 Five Criteria for Design Methods

- We can identify five criteria to help evaluate *modular design methods*:
  – modular decomposability;
  – modular composability;
  – modular understandability;
  – modular continuity;
  – modular protection.

2.1 Modular Decomposability

- This criterion is met by a design method if the method supports the decomposition of a problem into smaller sub-problems, which can be solved *independently*.
- In general, the method will be repetetive: sub-problems will be divided still further.
- *Top-down design* methods fulfill this criterion; stepwise refinement is an example of such a method.
- As a counter example, consider the idea of an *initialisation module*, which initializes all variable at the start of a program run. Such a module *does not* meet the decomposability criterion, as the initialisation module must access data from all other modules.

2.2 Modular Composability

- A method satisfies this criterion if it leads to the production of modules that may be *freely combined* to produce new systems.
- Composability is directly related to the issue of *reusability*, (which we will examine shortly).
- Note that composability is often at odds with decomposability; top-down design, for example, tends to produce modules that may *not* be composed in the way desired.

  This is because top-down design leads to modules which fulfill a *specific* function, rather than a general one.

- EXAMPLES
  1. The Numerical Algorithms Group (NAG) libraries contain a wide range of routines for solving problems in linear algebra, differential equations, etc.
  2. The UNIX shell (and to a lesser extent, MS-DOS) provides a facility called a *pipe*, written "—", whereby the standard output of one program may be redirected to the standard input of another; this convention favours composability.

2.3 Modular Understandability

- A design method satisfies this criterion if it encourages the development of modules which are easily understandable.
- COUNTER EXAMPLE 1. Take a thousand lines program, containing no procedures; it's just a long list of sequential statements. Divide it into twenty blocks, each fifty statements long; make each block a method.

  The methods that result cannot be understood without looking at the preceding and subsequent methods.
- COUNTER EXAMPLE 2. "Go to" statements.

2.4 Modular Continuity

- A method satisfies this criterion if it leads to the production of software such that a small change in problem specification leads to a change in just one (or a small number of) modules.
- EXAMPLE. Some projects enforce the rule that no numerical or textual literal should be used in programs: only symbolic constants should be used.
- COUNTER EXAMPLE. Static arrays (as opposed to open arrays) make this criterion harder to satisfy.

2.5 Modular Protection

- A method satisfied this criterion if it yields architecures in which the effect of an abnormal condition at run-time only affects one (or very few) modules.
- EXAMPLE. Validating input at source prevents errors from propogating throughout the program.
- COUNTER EXAMPLE. Using `int` types where subrange or `short` types are appropriate.

3 Five Principles for Good Design

- From the discussion above, we can distill five principles that should be adhered to:
  - linguistic modular units;
  - few interfaces;
  - small interfaces;
  - explicit interfaces;
  - information hiding.

3.1 Linguistic Modular Units

- A programming language (or design language) should support the principle of linguistic modular units:

  Modules must correspond to linguistic units in the language used.

- EXAMPLE. Java methods and classes.
- COUNTER EXAMPLE. Subroutines in BASIC are called by giving a *line number* where execution is to proceed from; there is no way of telling, just by looking at a section of code, that it is a subroutine.

3.2 Few Interfaces

- This principle states that the overall number of communication channels between modules should be as small as possible:

  Every module should communicate with as few others as possible.

- So, in a system with n modules, there may be a minimum of $n - 1$ and a maximium of

  $$\frac{n(n-1)}{2}$$

  links; your system should stay closer to the minimum.

3.3 Small Interfaces (Loose Coupling)

- This principle states:

  If any two modules communicate, they should exchange as little information as possible.

- COUNTER EXAMPLE. Declaring all instance variables as `public`!

3.4 Explicit Interfaces

- If two modules *must* communicate, they must do it so that we can see it:

  If modules A and B communicate, this must be obvious from the text of A or B or both.

- Why? If we change a module, we need to see what other modules may be affected by these changes.

3.5 Information Hiding

- This principle states:

  All information about a module, (and particularly *how* the module does what it does) shoud be *private* to the module unless it is specifically declared otherwise.

- Thus each module should have some *interface*, which is how the world sees it: anything beyond that interface should be hidden.
- The default Java rule:

  Make everything `private`.

4 Reusability

- A major obstacle to the production of cheap quality software is the intractability of the *reusability* issue.
- Why isn't writing software more like producing hardware? Why do we start from scratch every time, coding similar problems time after time after time?
- Obstacles:
  - economic;
  - organizational;
  - psychological.

5 Stepwise Refinement

- The simplest realistic design method, widely used in practice.
- Not appropriate for large-scale, distributed systems: mainly applicable to the design of methods.
- Basic idea is:
  - start with a high-level spec of what a method is to achieve;
  - break this down into a small number of problems (usually no more than 10);
  - for each of these problems do the same;
  - repeat until the sub-problems may be solved immediately.
- Breaking down one problem into a number of smaller ones is known as *refinement*.
- Including program code in refinement is *extremely bad practice* — this is *implementation bias*/

6 Object-Oriented Design

- For complex systems, stepwise refinement is inadequate.
- We use *object-oriented design*.
- For much of the remainder of this course, we focus on one particular OO design approach, using UML (the "unified Modelling Language").
- We beging by introducing basic object concepts.

6.1 What is an *Object*?

- An object is a *thing*!
  - *student*;
  - *transaction*;
  - *Lara Croft*;
  - *car*;
  - *customer account*;
  - *employee*;
  - *complex number*;
  - *spreadsheet table*;
  - *spreadsheet cell*;
  - *document*;
  - *paragraph*;
  - *GUI button*

  ... and so on.
- When trying to decide what is an object, look for *nouns* in your requirements specification.

6.2 What *isn't* an object?

- Two sorts of things:
  - *attribute of object*;
  - *operation on object*.
- Attributes:
  - *speed*, *color*, *make*, *model*, *owner*, and *position* are all attributes of a *car* object.
  - *number*, *owner*, *value* might be attributes of an *bank account* object.
- Operations:
  - *turn left*, *speed up*, *slow down*, *turn right* are all operations of a car object.
  - *open*, *close*, *deposit*, *withdraw*, are all operations on a *bank account* object.

  Operations (a.k.a. *behaviours*) correspond to *verbs* in a requirements specification.

  Example: *accelerate* the car, *process* the transaction.

6.3 Public & Private

- Each object has an *public interface* through which we can manipulate it.

  Car object interface: steering wheel, accelerator, ...
- The *only* way that we can manipulate an object is via its interface.

  Lifting the bonnet and fiddling with the engine directly is *not* going *around* the specification, and *can cause problems*: poor practice.
- Behind the scenes, an agent has a *private* part — its *state* and *internal operation*.
- The internal state & operation are *hidden* from the consumer.
- These ideas are known as:

  information hiding

  which is a *good thing*.

6.4 Objects & Classes

- We usually find it useful to *classify* objects into groups of similarity.
- For example, "Renault Clio" is a member of the class "car", as is "Peuguot 205".

  We say that "car" is a *class* and that "Renault Clio" and "Peuguot 205" are *sub-classes* of *car*.

  The sub-class relation is often written "is-a".
- Sub-classes are usually *specialisations* of their super-class.

  They tend to *inherit* the properties (attributes & operations) of their superclass.

- A *specific* object is an *instance* of a class.

  "My Peugot 205" is *an instance of* the "Peugot 205" class.
- Another type of relationship between classes: *aggregation* ("has-a").

  Example: car object contains four wheel objects, one steering wheel object, and so on.
- Individual objects have a unique *identity*, which makes them different from other objects of the same class.

  Two objects with the same state are not the same!

6.5 Object-oriented Programming

- The general process of OO software development involves:
  1. developing an appropriate *class/object model*, which identifies the classes and objects in your system;
  2. understanding the *attributes* and *operations* of classes;
  3. understanding the relationships *between* classes and objects (inheritance, aggregation);
  4. iterating steps (1) and (2) until satisfied;
  5. implementing the object model.

6.6 Summary

- Objects are *things*, which may correspond to physical things, events, legal institutions, or other abstractions (e.g., "discrepancy").
- Objects have:
  - a unique *identity*;
  - *attributes*;
  - *operations* or *behaviours*;
  - a *public interface*;
  - a *private component*.
- The public interface acts as a *contract*, or *specification* for the object.
- Objects are *instances* of a *class*.
- Classes can be related by:
  - the *sub-class* relationship ("is-a");
  - the *aggregation* relationship ("has-a").
- Sub-classes can *inherit* attributes and operations from superclasses.