COMP6700/2140 Program Design

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

24 March 2017

Modular Design

A large program should be divided into modules (classes, packages etc.), each with a distinct area of responsibility.

Deriving this division from the program specification is a focus of *Software Design*, which is a discipline in its own right.

- when done as a separate step (without immediate implementation), it is one of the traditional SE methodologies;
- when done iteratively as part of the coding process, it is a more modern style (*Agile Development*, *DevOps*)

In a simple system, the class design is often *ad hoc*: performed by choosing the class structure after analysing the data which program will be operating on: **data-driven approach**.

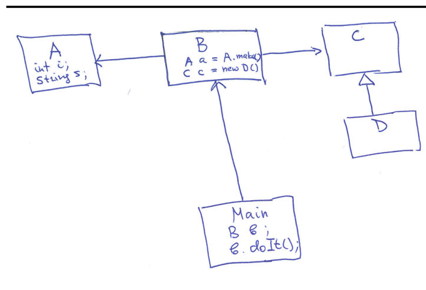
Operations are defined in method bodies. They often implement complex algorithms, however, *a method should do one thing, and only one thing*.

A Simple Program Design

1. One main class (with the application `main` method) which is relatively small:
   - Does not contain type declarations (classes *etc*)
   - Only performs initialisation and assembly of initial data set-up (opens file for data input, initialises `final` and `static` variables by reading command-line arguments, properties *etc*);
   - if the `main` logic is long and/or complex, one can define a few `static` helper methods inside the main class, but this is essentially it.
2. A number of classes, interfaces, enums and exceptions defined in separate files. Some may have interdependencies (composition, inheritance, delegation). These are your data abstractions:
   - Often represent *domain* (real-world) concepts in your program specification
   - If designed well, they can be reused in other programs
3. If required, one can also have a class (or several) with `public static` methods (similar to `java.lang.Math`, or `java.util.Arrays`, or `java.util.Collections`) to be used as your own ("micro") library of useful helper and utility methods, algorithm implementations *etc*.

Crude Design

- Do design even for simple systems
- Do it even if it's very crude
- Crude dodgy design (but not blatantly incorrect or misleading!) is better than no design.



Top-Down Design

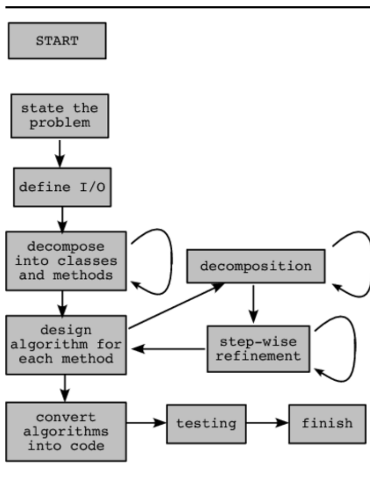
Ad hoc design may not be appropriate for a more complex system. Therefore, you have to postpone coding and first **design** the solution.

The most often used design method is *top-down* approach:

1. List the requirements (plain language description of the task(s) the system will perform)
2. Define the inputs and outputs (what data are transformed)
3. Decompose the program into classes and their associated methods
4. [Design an algorithm for implementing each method and describe it in *pseudo-code*]$^+$
5. Turn the pseudo-code into Java statements
6. Test the resulting Java program

Abstraction, decomposition and *step-wise refinement* make the most essential skills for writing software, regardless of the programming language.

Top-down Design Flow



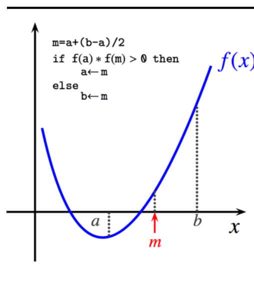
Pseudo-Code and Structured Programming — 1

How can you learn to program to solve problems, not to merely program?

To solve any problem requires application-specific skills - *domain knowledge*. A wide range of problems can be solved by thinking in terms of computational tasks — **computational thinking**.

Here is an elementary mathematics application domain.

A step-wise refinement approach to a problem of finding roots. **The problem**: Find a point `x` inside the interval between points `a` and `b`, `x∈[a,b]`, where the function `f(x)` crosses zero; it is given that `f(a) < 0` and `f(b) > 0`.



Pseudo-Code and Structured Programming — 2

The pseudo-code is a loose mixture of Java and English (or your native tongue). As you refine your design more and more, the "English" statements get replaced by Java's. Your pseudo-code algorithms should clearly show the steps which are *sequential*, those which are *conditional* (if blocks) and those which are *iteration* (loops). Any algorithm can be composed of these three elements. A computer language which has these constructs is *Turing-complete*. Thinking about algorithms in this way is called *structured programming* (E. Dijkstra).

- Top level: gist of the algorithm
  1. Evaluate `f` in the middle, choose that half of the original interval on which `f` changes its sign; continue until the interval shrinks to zero
- Refinement: elaborate on essential steps
  1. Take `a,b,f` and precision ε as inputs
  2. Divide `(a,b)`-interval in the middle, and evaluate function `f(m), m = (b-a)/2`
  3. If `f(m)=0` — Bingo! otherwise choose `[a,m]` or `[m,b]` on which `f` has opposite signs
  4. Repeat the above steps until the interval gets shorter than ε

Pseudo-Code and Structured Programming — 3

The next step in implementing the root-finding function — the pseudo-code:

- Pseudocode for the method `bisect(f,a,b,e)`

```
bisect(f, a, b, e)
   if sign f(a) = sign f(b) then return fail
   while |b - a| >= e
      m ← (a+b)/2
      if sign f(m) = f(a)
      then a ← m else b ← m end if
   end while
   return a
end bisect
```

Finally, the pseudo-code is converted into the language code (whatever the language is used for implementation). In Java, one cannot pass a function to a method as argument. We shall ignore the `f` argument, assuming that `f(x)` denotes an already defined function (method), and `bisect` calculates its root:

```java
double bisect(final double a, final double b, final double e) {
   // complete this code as a exercise
}
```

see *FindRoots.java* example code

Developing Code using Step-Wise Refinement — 1

This is a more mundane problem (used for one of the yesteryear assignments): Create a program which reads in a file containing a list of student names and IDs, then asks the user ("lecturer") to enter marks which every student earned in a course, calculates final mark and grade for every student, and displays the results in different order (alphabetical, ranking etc).

1. Input: a plain text data file with student (one per line) information, and user prompted input during execution
2. Output: sorted lists of students, and program messages which guide the user
3. Define a *Student* class, initialise a list structure to contain all student objects, open an input file for reading, read in line by line, determine names and ids, use them to create a student object, add it to the student list.
4. Go through the list, prompt user to enter *correct* marks (this is an infinite loop because the user should be given a choice of asking for help, he can enter a wrong value and the program must be able to correct him etc. — the dialog must be robust against incorrect input). When all marks are collected, calculate final marks and grades.
5. Sort the list in alphabetical or in ranking order.
6. Prompt user to choose a student to correct marks; repeat until user terminates the program (this dialog must be equally robust).
7. Save data in a file.

Developing Code using Step-Wise Refinement — 2

The program skeleton (high-level pseudocode expressed in Java) may look like this:

```
List students = new ArrayList<Student>(); // Define Student class separately
Scanner sc = new Scanner(infile);
while (sc.hasNextLine()) {
   studentData = sc.nextLine();
   student = new Student(studentData); // Define appropriate constructor in Student
   students.add(student);
}
procureMarks(); // first dialog session with the user
sort(students); // sort the list
collectErrata(); // second dialog to collect corrections
saveData(outfile, students);
```

First, methods can be defined as *stubs*; stubs have empty body or single `return` statement. When the top-level structures are complete, implement every stub method as necessary.

Data-Driven Approach

Another approach to designing a complex program is a bottom-up, which is better known as *data-driven*.

Data-driven programming makes a strong distinction between code (operations) and data structures on which the code acts. In design, the data are given priority, they are defined as the first step, after which the necessary operations follow. Change in logic of the program is achieved by editing the data structures, not the code. Transformation of data structures in the process of computation determine the program control flow (here is the difference between data-driven and object-oriented programming; also, if OO is also concerned with *encapsulation*, DD's goal is to minimise writing the actual coding instructions; in this aspect, DD programming is related to *functional programming*).

Data-driven approach often involves automatic *code generation* (writing code using programs). DD programming with code generation is used to produce visual components of graphical user interfaces (applications which allow the user to interact with it through a set of graphical elements, so called *widgets*, and other means of receiving "physical" input). Java was late to adopt such technology — *XAML* for Microsoft Windows, *Glade* for X/Gnome; *Qt* had *QtDesigner*, and recently added *QtQuick* and *QLM* language; *Apple* uses this technology heavily in *InterfaceBuilder* to create GUI for *MacOSX/iOS* applications/apps. Java has finally caught up with them with *JavaFX* GUI library and tools. We shall learn about it later in the course.

Principles of Class Design

- Keep the data (fields) private – data representation changes but that the way in which data is used changes much less frequently
- Initialise fields — don't rely on defaults for class instance fields but initialise them on declaration, in initialisation blocks or in constructors
- Don't use too many basic types in a class — multiple related uses of basic types should probably be another class:

  ```
  private String street;
  private String city;
  private String town;
  private int postcode;
  ```

  should be made part of a separate class *Address*
- Consider carefully which fields need individual accessors and mutators — some of them may be immutable (declare them `final` ?)
- Use standard (consistent) form for class definitions — introduce non-static fields, then static fields (if any), then constructors, then public methods, then private methods, then mutator methods, then accessor methods, then inner classes (if any)
- Break up classes with "too many" responsibilities — classes should be small! (try to adhere to *Single Responsibility Principle*)
- Make names of your classes and methods reflect their responsibilities (Noun, AdjectiveNoun or GerundNoun)

Designing Class for Extension

When a class is extended and an implementation of its method is changed, it's said then the method is *overridden* (not confuse with method *overloading*). When we are dealing with instances of a class, it is the *actual class of the object* which determines which implementation is used. Apart from changing implementation, the overriding can *widen* the access modifier, *ie*, make protected method `public`, but it cannot make it `private`. If the parental implementation need to be used, it can be done by using the reference to the superclass: `super.do()`.

When designing a class for extension, one must consider which access modifier to give to fields (`private` or `protected`, the last choice can be good for performance but must be exercises carefully) and to methods (`protected` or `public`). As every design aspect, this is a complex issue and requires experience and care, but the rule of thumb is that if you do not trust the class's possible children to preserve the integrity of the class contract, you should limit the access of its members. If the class method should not be overridden no matter what, it must be declared `final` (this will disallow any possible future overriding of the method by its descendants). If the whole class is not meant to be extended, it itself must be declared `final` (all methods in such a class are implicitly `final`).

```
final class NonExtendableClass { .... }
```

Attempts to declare a class which extends *NonExtendableClass* will be met with obstinate compiler error.

OO without Classes — Prototype Programming

What is more important for OOP — classes or objects? A class is an accumulation of properties which are common to instances. When a computation is conceived "from the top", defining classes is an adequate approach to capture abstraction, and class based implementation for creating objects is appropriate. Since Java is a statically typed compiled language, once an object is created, its structure and behaviour will not change (unless you are using the reflections or some other "black art", that is). This is good since it guarantees that object's contract will not change (which gives a layer of security to be expected in statically-typed languages), but sometimes one can benefit from the ability to change object's properties while it is alive. Dynamic OO languages (like Python) allow just that, but they still begin with classes (in Python, there are ways to restrain object's variability via the *slot*-mechanism).

What if the problem in question does not yield to easy classification: multiple objects do exist, but finding common features to define their class is not possible. (In philosophy, this problem was emphasised by Ludwig Wittgenstein). There is an alternative paradigm which is known as *prototype based* (OO) programming.

In languages like *Self, Lua, Smalltalk* (which allows classes, too) and (most popular) *JavaScript*, objects are created by cloning from a set of predefined object literals which are called **prototypes**, with the ability to add new attributes and behaviours during the object lifetime. In the prototype-based languages, the notion of type hierarchy does not exist.

Testing and Debugging

Three types of errors:

1. compile-time errors — use Java compiler error messages to find and correct them (usually mere syntax errors)
2. run-time errors — dangerous illegal ops; must be eliminated, *some* can be dealt with by *Java exception*
3. logical errors (*eg*, dangling `if-else` and other kinds) — most insidious, eliminated by
    1. code review (code reading)
    2. testing (unit testing *etc*)
    3. formal verification (special tools, mathematical logic representation)

   Advice on how to avoid the logical errors:

    1. break long statements into smaller ones
    2. check the correct use of methods in API docs
    3. check balance of parentheses, expression type in mixed arithmetic, etc
    4. *Static Code Analysis* tools (like FindBugs for Java) analyse your code (or bytecode) against a standard list of errors without running the program; bugs, potential performance and other problems are highlighted with suggestions for fixing them (I may show a demo later!)
    5. modern *IDE* (*Inspector* feature in *IntelliJ IDEA*) help to spot many errors and problematic choices (unused variables/methods etc), offer alternatives which often are superior...
    6. *etc, etc etc* — code, code, code — it comes with practice

Further Reading

- Andy Hunt and Dave Thomas, *The Pragmatic Programmer: From Journeyman to Master* (Addison-Wesley, 2000)
- Robert C. Martin, *Clean Code* (Prentice Hall, 2009)