THE IMPLEMENTATION

THE IMPLEMENTATION /Development

THE IMPLEMENTATION

Development - Plan driven

Design +Development -Agile

What should we talk or learn ?
Any thoughts

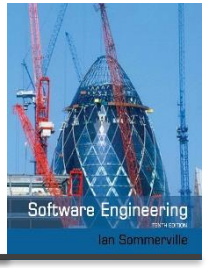
Overview

- Choice of programming language
- Good programming practice
- Coding standards
- Code reuse
- Integration
- Code Refactoring
- The implementation workflow: case study
- The test workflow: Implementation

Overview

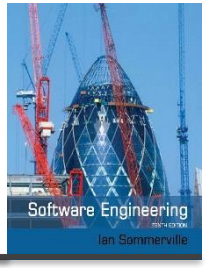
- What is a module?
- Cohesion
- Coupling
- Data encapsulation
- Abstract data types
- Information hiding
- Objects
- Inheritance, polymorphism, and dynamic binding

Design and implementation



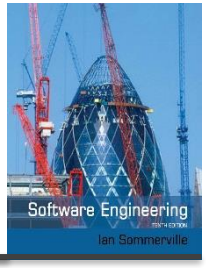
- Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- Software design and implementation activities are *invariably inter-leaved*.
  - Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
  - Implementation is the process of realizing the design as a program.

Build or buy



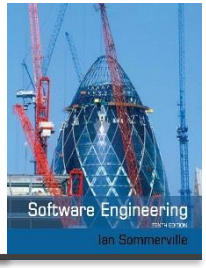
- In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
  - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.
- When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

An object-oriented design process



- Structured object-oriented design processes involve developing a number of different system models.
- They require a lot of effort for development and maintenance of these models and, for small systems, this may not be cost-effective.
- However, for large systems developed by different groups design models are an important communication mechanism.

Testing to Specifications versus Testing to Code



- There are two extremes to testing

- *Test to specifications* (also called black-box, data-driven, functional, or *input/output driven testing*) – **Functional testing**
  - Ignore the code — use the specifications to select test cases

- *Test to code* (also called glass-box, logic-driven, structured, or path-oriented testing)- **Unit testing**

Implementation(New perspective)

- Real-life products are generally too large to be implemented by a single programmer
- This chapter therefore deals with programming-in-the-many (Team/Groupwork)

13.1 Choice of Programming Language (contd)

- The Programming language is usually specified in the contract
- But what if the contract specifies that
  - The product is to be implemented in the "most suitable" programming language
- What language should be chosen?

Choice of Programming Language

Choice of Programming Language (contd)

- Example
  - QQQ Corporation has been writing COBOL programs for over 25 years
  - Over 200 software staff, all with COBOL expertise
  - What is “the most suitable” programming language?
- Obviously COBOL

Choice of Programming Language (contd)

- What happens when new language (C++/Java, say) is introduced
  - C++ professionals must be hired
  - Existing COBOL professionals must be retrained
  - Future products are written in C++
  - Existing COBOL products must be maintained
  - There are two classes of programmers
    - COBOL maintainers (despised)
    - C++ developers (paid more)
  - Expensive software, and the hardware to run it, are needed
  - 100s of person-years of expertise with COBOL are wasted

Choice of Programming Language (contd)

- The only possible conclusion
  - COBOL is the “most suitable” programming language
- And yet, the “most suitable” language for the latest project *may* be C++
  - COBOL is suitable for only data processing applications
- How to choose a programming language
  - Cost–benefit analysis (project management)
  - Compute costs and benefits of all relevant languages

Choice of Programming Language (contd)

- Which is the most appropriate object-oriented language?
  - C++ is (unfortunately) C-like
  - Thus, every classical C program is automatically a C++ program
  - Java enforces the object-oriented paradigm
  - Training in the object-oriented paradigm is essential before adopting any object-oriented language

13.2 Good Programming Practice

- Use of *consistent* and *meaningful* variable names
  - "Meaningful" to future maintenance programmers
  - "Consistent" to aid future maintenance programmers

13.2.1 Use of Consistent and Meaningful Variable Names

- A code artifact includes the variable names
  `freqAverage, frequencyMaximum, minFr, frqncyTotl`

- A maintenance programmer has to know if `freq, frequency, fr, frqncy` all refer to the same thing
  - If so, use the identical word, preferably `frequency`, perhaps `freq` or `frqncy`, but *not* `fr`
  - If not, use a different word (e.g., `rate`) for a different quantity

Consistent and Meaningful Variable Names (contd)

- We can use `frequencyAverage, frequencyMaximum, frequencyMinimum, frequencyTotal`
- We can also use `averageFrequency, maximumFrequency, minimumFrequency, totalFrequency`
- But all four names must come from the same set

13.2.2 The Issue of Self-Documenting Code

- Self-documenting code is exceedingly rare
- The key issue: Can the code artifact be understood easily and unambiguously by
  - The SQA team
  - Maintenance programmers
  - All others who have to read the code

Self-Documenting Code Example

- Example:
  - Code artifact contains the variable `xCoordinateOfPositionOfRobotArm`
  - This is abbreviated to `xCoord`
  - This is fine, because the entire module deals with the movement of the robot arm
  - But does the maintenance programmer know this?

Prologue Comments

- Minimal prologue comments for a code artifact

The name of the code artifact
A brief description of what the code artifact does
The programmer's name
The date the code artifact was coded
The date the code artifact was approved
The name of the person who approved the code artifact
The arguments of the code artifact
A list of the name of each variable of the code artifact, preferably in alphabetical order, and a brief description of its use
The names of any files accessed by this code artifact
The names of any files changed by this code artifact
Input–output, if any
Error-handling capabilities
The name of the file containing test data (to be used later for regression testing)
A list of each modification made to the code artifact, the date the modification was made, and who approved the modification
Any known faults

Figure 13.1

Other Comments

- Suggestion
  - Comments are essential whenever the code is written in a non-obvious way, or makes use of some subtle aspect of the language
- Nonsense!
  - Recode in a clearer way
  - We must never promote/excuse poor programming
  - However, comments can assist future maintenance programmers

13.2.3 Use of Parameters

- There are almost no genuine constants
- One solution:
  - Use `const` statements (C++), or
  - Use `public static final` statements (Java)
- A better solution:
  - Read the values of “constants” from a parameter file

13.2.4 Code Layout for Increased Readability

- Use indentation
- Better, use a pretty-printer
- Use plenty of blank lines
  - To break up big blocks of code

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <EditText
        android:id="@+id/idEdtUserName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter User Name"
        android:inputType="textEmailAddress" />

    <EditText
        android:id="@+id/idEdtPassword"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/idEdtUserName"
        android:hint="Enter Password"
        android:inputType="textPassword" />
</RelativeLayout>
```

13.2.5 Nested `if` Statements

- Example
  - A map consists of two squares. Write code to determine whether a point on the Earth's surface lies in `map_square_1` or `map_square_2`, or is not on the map

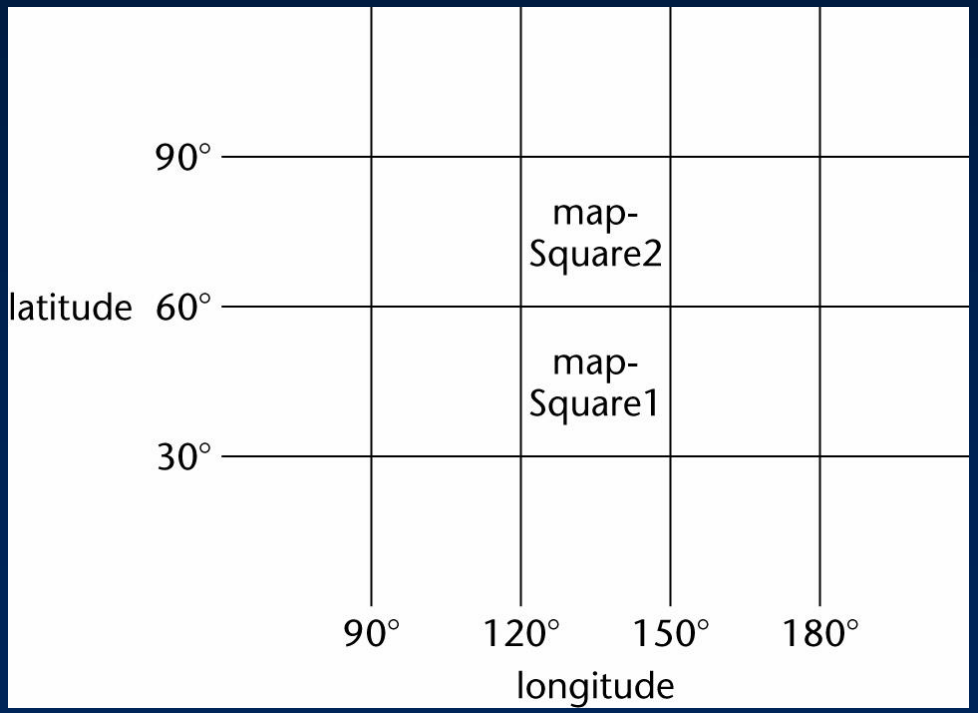


Figure 13.2

Nested `if` Statements (contd)

- Solution 1.

```
if (latitude > 30 && longitude > 120) {if (latitude <= 60 && longitude <= 150)
mapSquareNo = 1; else if (latitude <= 90 && longitude <= 150) mapSquareNo = 2
else print "Not on the map";} else print "Not on the map";
```

Figure 13.3

Nested `if` Statements (contd)

- Solution 1. Badly formatted

```
if (latitude > 30 && longitude > 120) {if (latitude <= 60 && longitude <= 150)
mapSquareNo = 1; else if (latitude <= 90 && longitude <= 150) mapSquareNo = 2
else print "Not on the map";} else print "Not on the map";
```

Figure 13.3

Nested `if` Statements (contd)

- Solution 2. Well-formatted

```
if (latitude > 30 && longitude > 120)
{
   if (latitude <= 60 && longitude <= 150)
      mapSquareNo = 1;
   else
      if (latitude <= 90 && longitude <= 150)
         mapSquareNo = 2;
      else
         print "Not on the map";
}
else
   print "Not on the map";
```

Figure 13.4

Nested `if` Statements (contd)

- Solution 2. Well-formatted, badly constructed

```
if (latitude > 30 && longitude > 120)
{
   if (latitude <= 60 && longitude <= 150)
      mapSquareNo = 1;
   else
      if (latitude <= 90 && longitude <= 150)
         mapSquareNo = 2;
      else
         print "Not on the map";
}
else
   print "Not on the map";
```

Figure 13.4

Nested `if` Statements (contd)

- Solution 3. Acceptably nested

```
if (longitude > 120 && longitude <= 150 && latitude > 30 && latitude <= 60)
  mapSquareNo = 1;
else
  if (longitude > 120 && longitude <= 150 && latitude > 60 && latitude <= 90)
    mapSquareNo = 2;
  else
    print "Not on the map";
```

Figure 13.5

Nested `if` Statements (contd)

- A combination of **`if-if`** and **`if-else-if`** statements is usually difficult to read

- Simplify: The **if-if** combination

```
if <condition1>
        if <condition2>
```

is frequently equivalent to the single condition

```
if <condition1> && <condition2>
```

Nested `if` Statements (contd)

- Rule of thumb
  - `if` statements nested to a depth of greater than three should be avoided as poor programming practice

13.3 Programming Standards

- Standards can be both a blessing and a curse
- Modules of coincidental cohesion arise from rules like
  - "Every module will consist of between 35 and 50 executable statements"
- Better
  - "Programmers should consult their managers before constructing a module with fewer than 35 or more than 50 executable statements"

Remarks on Programming Standards

- No standard can ever be universally applicable
- Standards imposed from above will be ignored
- Standard must be checkable by machine

Examples of Good Programming Standards

- "Nesting of `if` statements should not exceed a depth of 3, except with prior approval from the team leader"
- "Modules should consist of between 35 and 50 statements, except with prior approval from the team leader"
- "Use of `goto`s should be avoided. However, with prior approval from the team leader, a forward `goto` may be used for error handling"

Remarks on Programming Standards (contd)

- The aim of standards is to make maintenance easier
  - If they make development difficult, then they must be modified
  - Overly restrictive standards are counterproductive
  - The quality of software suffers

13.4 Code Reuse

- Code reuse is the most common form of reuse
- However, artifacts from all workflows can be reused (requirements, design and testing)

13.5 Integration

- Next Class