Analysis, Design and Implementation

Topic 9:
Elements of Good Design

V1.0

Introduction

- Part of the process of building a design view of a system is improving upon what is already in place. Only rarely are systems developed without there already being something in place to model them on.
- In software development, 'improvement' is not a fixed quality. Different developers will have different opinions on what is best.
- However, there are certain things for which we can aim.

V1.0

Software Quality Attributes

- There are formal taxonomies about what constitutes good software. They all include broadly the same things.
- We can break these qualities into three rough categories:
  - System measures
  - Architectural measures
  - Project measures

V1.0

System Measures - 1

- ***System measures*** are those that describe and define the system while it is running.
  - Functionality
    - Does it do what it's supposed to do?
  - Performance
    - How efficiently does it accomplish its goals?
  - Security
    - How well protected are the sensitive parts of the system?
  - Reliability
    - How much can you rely on the software being available when you need it?

System Measures - 2

- Usability
  - How easily the system can be manipulated by users, especially those that may not be experts.
- Interoperability
  - How well can it work with the other systems with which it may need to communicate?
- Correctness
  - How correct is the functionality? Does it give answers that are suitably precise?

Architectural Measures - 1

- ***Architectural measures*** relate to the way the system was designed and coded. These include:
  - Maintainability
    - How easily can improvements and fixes be made to the system?
  - Reusability
    - How easily can elements of the system be incorporated into future systems?

Architectural Measures - 2

- Testability
  - How easily can we test that the system does what it is supposed to do?
- Portability
  - How easily can the system be built and deployed for a platform for which it was not originally written?

Project Measures

- ***Project measures*** are related to the management of the OOAD process.
  - Cost
    - How much did the system cost and how much was it costed for?
  - Schedule
    - How long was it supposed to take and how long did it take?
  - Marketability
    - Is it software designed for the marketplace, and if so what is it that sets it apart from the competition?

Assessing Quality

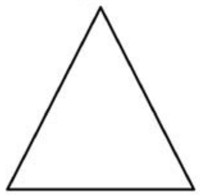
- Assessing quality is sometimes a qualitative process.
  - You go by what people say.
- Sometimes it can be quantified.
  - Running test cases can identify performance, reliability and correctness.
  - User testing can identify functionality and usability.
- Sometimes it is related to choices made in the design phase.
  - Portability, for example.

Trade-offs

- When performing the analysis, you must determine which of these qualities are going to be emphasised.
  - This will influence how you can emphasise others.
- During the design phase, you must decide how you are going to honour that emphasis.
  - Choosing to emphasise maintainability will influence the cost and efficiency of the system.
  - Emphasising speed of development will impact on the quality and cost.

The Project Triangle

Good



Cheap Fast

Pick Any Two

Assessing System Measures

- Assessing system measures can usually only be done once something has been implemented.
  - Not all of it, just enough to give a ‘ball park’ figure for quantifiable measures.
- Incorporating this analysis into your development process can be valuable.
  - Test driven development
  - Benchmarking

Test Driven Development - 1

- Regression testing is an important part of ensuring correctness of software.
  - It is estimated that for every two bugs you fix in a program, you introduce one more.
- Test driven development can help identify new problems as early as possible.
- Test driven development works by writing the tests ***before*** you write the code, and automating the running of those tests.

Test Driven Development - 2

- Whenever you make a change to a piece of code, you run all the automated tests.
  - In this way, you can make sure that the functionality you are developing does not break existing functionality.
- The process for development then is:
  - Add a test (These should be simple enough to be expressed in a single line, such as, 'is this list empty')
  - Run all the tests
  - Write the new code
  - Run the tests again
  - Refactor to resolve issues
  - Repeat

Benchmarking - 1

- Benchmarking allows for you to determine the efficiency of code and then optimise accordingly.
  - However: "Premature optimisation is the root of all evil" – Donald Knuth
- Sometimes you can make use of industry standard benchmarks.
  - For testing graphical performance, for example.
- More often you will need your own bespoke architecture for this.

Benchmarking - 2

- When sure you have correctly functioning code, you can run your benchmarks.
- These fall into two categories:
  - Profiling
  - Performance benchmarking
- The former will show you which parts of your system are using the most CPU.
  - These are the best candidates for optimisation.
- The latter will show you the impact of performance fixes you make.

Bespoke Benchmarking

```
import java.util.*;

public class Benchmarking {

public static void main (String args[]) {
        Date time = new Date();
        int iterations = 1000000;
        long now = time.getTime();
        long then;
        double total;
        for (int i = 0; i < iterations; i++) {
                String bing = new String ("Bing");
        }

        time = new Date();
        then = time.getTime();
        total = then - now;
        System.out.println ("Method took " + total + " milliseconds.");
        }
}
```

Source: Used with permission from http://www.monkeys-at-keyboards.com/java2/31.shtml

Optimisation - 1

- Once you have identified a performance issue in your system, you can optimise it.
  - Be aware of the 80/20 rule here.
- There are several standard techniques:
  - Strength reduction
    - Replacing slow code with faster code.
  - Sub-expression elimination
    - Re-use the results of calculations where possible.

Optimisation - 2

- Code motion
  - Move invariant code out of loops
- Re-use objects
  - Don't instantiate an object when you can re-use an existing object.
- Cache common operations
  - A cache lets you store the results of operations so that you can pull them out of the store rather than recalculate.

Architectural Measures

- Architectural measures are best assessed at the design phase.
  - The class diagram will be a useful tool for this.
- We want to aim for systems that have ***low coupling*** and ***high cohesion***.
  - Sadly, these are mutually exclusive measures of quality.
- Coupling defines inter-dependencies between various modules.
- Cohesion defines how tightly the methods of a module are related.

Types of Coupling - 1

- There are many different kinds of coupling, and some are worse than others. From worst to best:
  - Content coupling.
    - When a module makes use of the local data of another. The worst kind of coupling.
  - Common coupling
    - When two modules share the same global data store.

Types of Coupling - 2

- Data coupling
  - When modules share data via parameters.
- Callback coupling
  - Such as in the ***observer design pattern***. (We'll talk about that later in the lecture.)

Why is Coupling Bad?

- Coupling makes it hard to extract classes from their context. This makes re-use difficult.
- Coupling makes it difficult to change code.
  - You will most likely need to change tightly coupled code as well.
- However, it's not always bad.
  - If coupling was always bad, then surely no coupling can be good. This is not the case.
  - We always need to be able to communicate between subsystems.
- The best kinds of coupling are not 'good'.
  - They are just better than the alternatives.

Cohesion - 1

- The degree to which a module fills a single role determines its cohesion.
  - As in, all the parts of the module should be well aligned to solving a particular problem.
- Cohesion is a qualitative measure, and again can be measured in many ways.
- High cohesion is good because it makes it easier to:
  - Understand what classes do
  - Reuse the classes
  - Maintain the classes

Cohesion - 2

- There are many different kinds of cohesion. From worst to best:
  - Coincidental cohesion
    - No real connection between modules.
  - Logical cohesion
    - Modules are logically linked in what they do.

Cohesion - 3

- Temporal cohesion
  - Modules are linked together because they tend to be executed at the same point in a program's lifetime.
- Communication cohesion
  - Modules are linked together because they act on the same kinds of data.
- Functional cohesion
  - All modules contribute to the processing of a well defined task.

Fixing Architectural Problems - 1

- First of all, you must identify what those problems are.
  - Identify classes with low cohesion
  - Identify classes with high coupling
  - Identify the nature of the coupling between classes.
- Hide and encapsulate information in classes.
  - This will ensure that any coupling is of the better kinds.
- Refactor classes to improve their cohesion.
  - Merge and divide where necessary.

Fixing Architectural Problems - 2

- When you emphasise cohesion, you will have to sacrifice some potential coupling efficiencies, and vice versa.
- However, coupling is fine if it's the right kind of coupling and not too freely used.
- One of the reasons why design patterns are useful is that they represent a good balance between coupling and cohesion.
- When you identify coupling, either refactor it away or refactor it to a less problematic form.

Software Component Design

- One of the ways in which you can neatly resolve architectural issues is in treating each subsystem of your program as a component.
  - A black box which has no knowledge of how the rest of your system works.
- Components can be collections of classes.
  - They should all be linked together to process one well defined part of the system.
- Communication via different parts of the system is then handled via the ***observer design pattern***.

Observer Design Pattern

- The Observer design pattern allows for an object to maintain a list of other objects that are interested in when its state changes.
- When the state changes, it then notifies all of these interested objects (observers) that a change has been made.
- Objects are responsible for registering themselves as observers, and for deregistering them when it is no longer relevant.

Observer Example - 1

```
import java.util.*;

public interface AccountExampleInterface {
    public void stateChanged (int current, int change);
}

public class Account {
    private int amount;
    ArrayList<AccountExampleInterface> myListeners;

    public Account() {
        myListeners = new ArrayList<AccountExampleInterface>();
    }
    public void addListener (AccountExampleInterface a) {
        myListeners.add (a);
    }
    public void removeListener (AccountExampleInterface a) {
        myListeners.remove (a);
    }
    public void notifyListeners (int current, int amount) {
        for (AccountExampleInterface a : myListeners) {
            a.stateChanged (current, amount);
        }
    }
    void adjustBalance (int val) {
        amount += val;
        notifyListeners (amount, val);
    }
}
```


Observer Example - 2

```
public class InterestedParty implements AccountExampleInterface {
    public void stateChanged (int current, int amount) {
        System.out.println ("I was interested that the account was adjusted by " +
amount);
    }
}

public class ObserverExample {

     public static void main(String[] args) {
            InterestedParty ip = new InterestedParty();
            Account myAccount = new Account();
            myAccount.addListener (ip);
            myAccount.adjustBalance (1000);

            myAccount.removeListener (ip);
            myAccount.adjustBalance (-1000);

            myAccount.addListener (ip);
            myAccount.adjustBalance (2000);

    }
}
```


Software Components

- Software components permit you to subdivide your project.
  - Each component can be optimised separately.
  - Communication can be handled via loose coupling such as the observer pattern.
- By limiting the scope of any component, greater architectural elegance can be obtained.
  - This is the key to good software design.
- High quality software is a process, not a deliverable.

Conclusion

- Part of our role as software developers is to create ***good*** software.
  - This involves understanding the implications of our decisions.
- Software quality attributes involve trade-offs.
  - We can't have them all, so we must decide what we ***need***.
- There are various ways to assess and improve the quality of our software.
  - We have discussed a number of these.

Topic 9 – Elements of Good Design

Any Questions?