Bringing British Education to You
www.nccedu.com

Analysis, Design and Implementation

Topic 11:

Maintenance and Refactoring

Introduction

- An element of implementation that is often overlooked is that of maintenance.
  - Writing the software is only part of the job. The other part is maintaining it.
- Much developer time is spent maintaining software.
  - More than is spent writing it the first time, although the exact ratios vary from study to study.
- Sometimes the maintenance is done by a different team, but more often than not it will be you who has to do it.

Types of Maintenance

- The way in which maintenance is applied to a software system falls into four categories:
  - Adaptive Maintenance
  - Corrective Maintenance
  - Perfective Maintenance
  - Preventive Maintenance
- The largest amount of time is spent on the first two of these.
  - These are the highest gain, and so more attractive when time and money is limited.

Adaptive Maintenance

- ***Adaptive maintenance*** is being performed when software is changed to meet changed requirements.
  - Adjusting software to be compliant with new regulations.
  - Adding support for a new protocol.
  - Alterations in an internal organisation workflow.
- Adaptive maintenance does not include new features.
  - It is used to keep a system up to date with changing requirements.
- This is a reactive maintenance process.

Corrective Maintenance

- ***Corrective maintenance*** involves identifying defects in software and then resolving them.
  - Many bugs in a system will not be discovered until people have started using it.
- Corrective maintenance is the ongoing process of prioritising bug-reports, implementing corrections, and then integrating them into the system.
  - This can be a complex task, especially in mission and safety critical systems.

Perfective Maintenance

- ***Perfective maintenance*** aims to enhance a system with new features and functionality.
  - Usually in response to user requests, but not always.
- It too is a reactive maintenance process.
- It can also include performance enhancements.
  - Increases in efficiency
  - Increases in reliability
- This phase usually includes some degree of refactoring.

Preventive Maintenance

- ***Preventive maintenance*** involves identifying problems before they occur and re-engineering so they don't.
  - This can involve refactoring and restructuring of a system.
- We often know where, architecturally, a system is likely to have problems in the future.
- We can put off fixing these problems until later in the development if the problems are not immediate.

Refactoring

- A big part of what ongoing maintenance involves is known as ***refactoring***. In its simplest terms, this is the process of turning bad code into good code.
- Ideally, refactoring is an invisible process.
  - If you do it right, no-one using the software should know you've done it at all.
- Refactoring is often a precursor to other kinds of maintenance.
  - It's about making it easier to work with software code

Impact of Change - 1

- Maintenance is often made easier or more difficult by the ***impact of change*** that goes with altering code.
  - Much of what software engineering is about is managing the impact of change.
- Highly coupled objects have a high impact of change.
  - If you make a change, you often need to make alterations to the objects that make use of it.

Impact of Change - 2

- Encapsulation limits the impact of change through the mechanism of data hiding.
  - The impact of change of a private attribute or method is limited to that one class.
- Public APIs (such as exposed by a facade) have a high impact of change.
  - You can’t modify them without concern for all the classes and objects that may be making use of them.
- Impact of change is a measure for how developer intensive a modification will be.

Impact of Change - 3

- Impact of change relates to the maintainability of your code.
  - How much of the code do you have to change when you make a modification?
- As developers, we strive to ensure minimum impact of change.
  - You need to labour under the assumption that if someone ***has access*** to a method or variable, they have taken advantage of that, no matter how obscure the method or variable may be.

Rules for Refactoring

- There are some firm rules that must be followed when refactoring:
  - Methods and variables may be made more visible. They may not be made less visible.
  - The functionality of public methods cannot change. If a public method does X, it should continue to do X (and nothing more or less) after it has been refactored.
  - The return type of a method cannot change
  - The name of a method or public/protected variable cannot change.
  - The parameter list of a method must remain the same, or there must be a translation scheme in place for a change.

Breaking the Rules

- You can however choose to break these rules if you have the authority to make changes throughout all affected parts of the system, provided you take the responsibility for fixing all the problems you cause.
- Breaking the rules can also be permitted when enough notice is given.
  - Announce your intention to change a part of the system.
  - Give people time to make the changes
  - Deprecate the existing code.

Refactoring

- Refactoring may involve a wide range of activities, but the process usually includes:
  - Removing dead code
  - Making inefficient code more efficient
  - Making code more readable
  - Making code more maintainable.
- Refactoring should be a proactive process. It should be an ongoing part of your development cycle, but it often isn't.

Common Refactoring Tasks - 1

- Some common structural tasks performed during refactoring:
  - Generalising object functionality
    - Moving a method from one class to a more general parent.
  - Specialising object functionality
    - Moving general functionality into a more specialised child.
  - Improving encapsulation
    - Relocating data while deprecating obsolete calls.
  - Lower the impact of change
    - Modifying access permissions while ensuring compatibility.

Common Refactoring Tasks - 2

- There are also refactoring tasks at the level of an object and method:
  - Simplifying internal structures
  - Improving variable names
  - Simplifying logical comparisons
  - Substituting one algorithm for another
  - Consolidating conditionals
  - Extracting functionality into separate methods
  - Reducing inconsistency in naming and parameter ordering.

A Simple Example

- Consider the following simple class:

```
public class Example {
        private int bing;

        public int getValue() {
                return bing;
        }

        public void setValue (int b) {
                bing = b;
        }
}
```

A Simple (?) Example

- Depending on the ***impact of change***, even changing a variable name can be problematic.

```
public class Example {
        public int bing;

        public int getValue() {
                return bing;
        }

        public void setValue (int b) {
                bing = b;
        }
}
```

Impact of Change

- Structural elements of a system usually carry with them a high impact of change.
  - It's usually safe to ***specialise***, it's usually ***not*** safe to generalise.
- In all cases, we want to refactor in such a way that our changes have limited impact on anyone else.
  - Fellow developers, mainly.
  - This is a necessary aspect of courteous development

Another Example

- Let’s say we have a method to which we need to add a parameter – say we need getValue to accept an integer parameter:

```
public class Example {
        private int bing;

        public int getValue() {
                return bing;
        }

        public void setValue (int b) {
                bing = b;
        }
}
```

Another Example

- How do we do this? There are two real choices:
  - Add the parameter to the method definition.
    - High impact of change – every other class making use of getValue will need to change.
  - Add in an overloaded method.
    - Low impact of change.
- However, there’s a trade-off here:
  - Adding a parameter may require lots of code to change.
  - Adding an overloaded method may reduce internal consistency.
- Incremental adjustments take time to ripple through a system.
  - Deprecated code sometimes takes decades to fix!

Remit of Refactoring - 1

- Where does your remit for refactoring lie?
  - It depends on how much of the code for which you are responsible.
    - This may extend over a whole program.
    - It may extend over a handful of classes.
  - You can unilaterally refactor only those elements of the program for which you have responsibility.

Remit of Refactoring - 2

- There are few things more frustrating than finding your programs no longer work because of someone else’s refactoring…
  - Usually you blame yourself rather than realising that the context of your code may have been unilaterally altered.

A Third Example

```
public class Example {

        private int value;

        public int makeDesposit (int value, int rate) {

                if (value > 100) {
                        return -1;
                }
                else if (valid < 0) {
                        return -1;
                }
                else {
                        if (rate < 30) {
                                value = value * rate;
                        }
                        else {
                                 value = value * rate;/2
        }
                        }
                }

                return value;
        }

}
```

Code Aesthetics - 1

- The aesthetics of your code are important.
  - They are usually a hint at the maintainability.
- However, it's important not to just discard complicated code as needing total refactoring.
  - Old code may not be ugly, it may be ***battle-scarred***.

Code Aesthetics - 2

- However, as you gain in experience and confidence as a developer, you can generally tell where the bits of code needing attention lie.

- Everyone has their own way of doing this – you might investigate the ***Wodehouse Method of Refactoring*** for one interesting example:

  http://basildoncoder.com/blog/2008/03/21/the-pg-wodehouse-method-of-refactoring/

Code Aesthetics - 3

- One way to improve the aesthetics is to break complicated functionality out into separate methods.
  - This fulfils a general rule of object-oriented programming, that each method should have one responsibility only.
- Complicated and nested structures are usually a good warning sign of the need to refactor.
- Consider where design patterns can help you deal with complicated data structures.
  - Design patterns are ‘good’ solutions to many endemic problems in software development.

A Third Example – Refactored - 1

```
public class Example {
        public int value;

        private boolean getValid (int value) {
                if (value > 100 || value < 0) {
                        return false;
                }

                return true;
        }

        private int getRate (int rate) {
                int rate;

                if (rate > 20 && rate < 30) {
                        rate = value;
                }
                else {
                        rate = value / 2;
                }
                return rate;
        }
}
```

A Third Example – Refactored - 2

```
public int makeDeposit (int value, int rate) {
       int rate;

       if (getValid (value) == false) {
              return -1;
       }

       rate = getRate (rate);


       value = value * rate;

       return value;

}
```

Refactoring and Test Driven Development

- Refactoring introduces no new functionality.
  - You can thus use the tests you have put in place previously.
- Having a comprehensive, full test-suite ensures that your refactored code behaves identically to the previous code.
  - The importance of that in a multi-developer environment cannot be stressed enough.

When Do We Refactor? - 1

- In a perfect world, refactoring would be an ongoing process we do all the time.
  - The ideal case is that we refactor our code on a continual basis.
- The realities of life dictate that we must prioritise.
  - Generally, we refactor code that is actively getting in the way.
  - There's also often a 'wish list' we keep as developers of code that we would like to refactor…

When Do We Refactor? - 2

- We refactor when code ‘smells bad’:
  - Code is duplicated across locations.
  - There are unjustifiable ‘god objects’.
    - Objects that have too much responsibility, too much power, and too much access to other parts of the system.
  - When cohesion is too low.
  - When coupling is too high.
- For more info on ‘bad code smells’:

http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm

Conclusion

- Maintenance is an important aspect of software development.
- Before we investigate particular maintenance needs, we often need to refactor code.
  - Refactoring is an intensive process.
- Refactoring allows us to re-engineer systems so that they are amenable to alteration.
  - This greatly simplifies our later maintenance requirements.

Topic 11 – Maintenance and Refactoring

Any Questions?