**Object Oriented Software Engineering 2**

**Lecture 1: Introduction to Software Engineering**

The term “software engineering” was coined by Margaret Hamilton.

Software Systems

A simple system interacting with its environment using input and output messages.

In the basic form a system has input messages and output messages. Sometimes modules interact with each other through input messages and output messages. For example this could be calls to an API.

A complex system with multiple sub-systems and interactions.

* A software system is a web of interconnected sub-systems, where each sub-systems may or may not be divided into further sub-systems

Abstraction

A way of understanding a complex system is to implement it with abstraction.

This is where you remove the unnecessary details in a problem. There are two main forms of abstraction procedural and data abstraction.

Procedural Abstraction

Where the programmer doesn’t need to know the exact details of the process. For example, a function on a routine.

This is simplifying how the computations are performed.

Data Abstraction

Helps us reduce the systems complexity by grouping together the pieces of data that describe some entity.

So the programmers can manipulate that data as a single unit.

When modelling a component you want to identify the salient parts and behaviour of something in the thing we want to model

Object Oriented Design

An object combines data and operations on that data (object is an instance of class)

Data: class variables

Operations: methods.

In real life you will be spending a lot of time designing the software before you actually start coding.

Three main principles of Object Oriented Design

Encapsulation

* Combining data and operations in one entity

Inheritance

* Classes can inherit from other classes (sub-classing) e.g. animals, dogs and so on

Polymorphism

* Means “many forms”
* A mechanism that enables several methods in related classes to have the same name and implements the same abstract operations

Object Oriented Design

Given a problem statement and requirements, you carry out the following activities

* Identify Objects
* Identify Operations
* Create Interfaces
* Object Interaction Design
* Etc.

Identifying Objects

Identify (real-world) Objects:

* Identify objects that exist in the problem Statement and requirements
* Typically, select nouns, ignoring irrelevant ones, such as synonyms

We want to look for relationships amongst the objects that were identified

* Generalization – relates to inheritance
* Containment -where one object contains another
* Multiplicity – determine the quantity relationships between objects (e.g. one animal can have many legs)

Identify Operations

Once you have identified the objects then you need to identify the objects:

Identify the operations of objects:

* Typically selecting verbs from the problem statement
* Associate each operation with the object that is responsible for providing the behaviour

Creating Interfaces

* An interface is created for each object that is to be represented by a class
  + The interface describes how the class can be used, by specifying its public operations
* An interface should include:
  + Return type
  + Purpose (i.e. a description)
  + Pre and post conditions

Object Interaction Design

* Describe how the objects **communicate** with each other via **operations.**
* And how the object, operations and communication affects the end-users.

Software Design

Designing software is a **symbiotic relationship** between the end-user and designer that requires the software designer that requires the software designer to make the right design decisions:

* E very design decision reflects an intent on how the software is to function or be used
* As well as end users’ expectations as how the software is compatible with contextual norms.

What makes a good software design?

A good software design is a balance of the following factors:

1. Modularity
2. Modifiability – The system should be easy to modify and update to keep up to date with changing times
3. Ease of Use
4. Efficient
5. Correct
6. Maintainability
7. Understandability – Comes in different forms
8. Reusable
9. Portable
10. Fail-Safe
11. … etc

**Lecture 2: UML Diagrams**

Software Design

Design: specifying the structure of a software system and its functions (behaviour)

* Its an opportunity to get insights on design alternatives to make appropriate design choice
* You can also evaluate the extent to which the system compiles with end user expectations

A transition from “what” the system must do, to “how” the system will do it

* What classes will we need to implement a system that meets our requirements?
* What fields and methods will each class have

Does Design Matter?

When do you think it is most suitable to design a software system?

1. Design while implementing the code (i.e. think about design using program code as a language for system modelling) – Then automatically generate model representations (such as UML class diagram)
2. Before implementation using an abstract modelling language such as UML
3. After implementation (you generate the design automatically from the code)

Object-Oriented Analysis

Domain Model:

A conceptual model of the domain that incorporates both behaviour and data

What is UML

UML stands for Unified Modeling Language. It is a Diagrammatic representation of an Object Oriented system

* Programming languages are not abstract enough for OO design
* UML is an open standard; lots of companies use it

What is legal UML?

* A descriptive language

Class Diagrams

What is a UML class diagram?

* A diagram of the classes in an OO system, their fields and methods, and connections between the classes that interact or inherit from each other

How do we design classes?

Identify classes and interactions from project requirements:

* **Nouns** are potential classes, objects, and fields
* **Verbs** are potential methods or responsibilities

Relationships Between Classes

**Generalization**: an inheritance relationship

* Inheritance between classes
* Interface implementation

**Association**: a usage relationship

* Dependency
* Aggregation
* Composition

Generalization Relationships

Hierarchies drawn top-down with aarrows point upward to parent.

Line/arrow styles indicate if parent is a(n):

**Lecture 3: Software Metrics**

Outline

* Motivation and how the quality of software can be measured
* Control Flow Graphs (CFG)

Desirable Properties of Software

Generally, when building software we want to:

* Reduce complexity
* Increase modularity
* Increase maintainability
  + Increase cohesion
  + Reduce coupling
* Increase reusability
* Increase usability

Motivation for Metrics

* Estimate the cost and schedule of future projects
* Evaluate the productivity impacts of new tools and techniques
* Establish productivity trends over time
* Improve software quality

Definitions

* Measure – quantitative indication of extent, amount, dimension

How can Quality be Measured?

* To define what can be used as a basis for measurement, Bassili proposed a top-down goal oriented framework for software metrics:
  + Step 1. Develop a set of Goals
  + Step 2. Develop a set of questions that characterise the goals
  + Step 3. Specify the Metrics needed to answer the questions
  + Step 4. Develop Mechanisms for data Collection and Analysis
  + Step 5. Collect Validate and Analyse the Data
  + Step 6. Analyse in a Post Mortem fashion

Control Flow Graph (CFG)

* A representation, using graph notation, of all paths that might be traversed through a program during its execution

McCabe’s Cyclomatic Metric

Given a control flow graph G, where the cyclomatic complexity is represented by , then:

Where is the number of nodes in and is the number of edges in .

Advantages

* Easy to compute (parser)
* Empirical studies: good correlation between cyclomatic complexity and understandability

Disadvantages

* Only control flow
* No data flow
* May be inappropriate for OO programs (trivial functions)

CK Metrics

In 1994, Shyam Chidamber and Chris Kemerer defined six simple metrics for object-oriented programs

* Since then this work has been extended over 300 metrics

CK Metrics: Objectives

* **WMC:** Weighted Methods Per Class
* **DIT**: Depth of Inheritance Tree
* **NOC:** Number of Children
* **CBO:** Coupling between Object Classes
* **RFC:** Response for a Class
* **LCOM:** Lack of Cohesion of Methods

Weighted methods per class (WMC)

This is the sum of the complexities of methods in a class

* Ci is the complexity of each method Mi of the class
* Complexity is the McCabe complexity of the method
* Smaller values are better
* WMC is a predictor of how much TIME and EFFORT is required to develop and maintain

Depth of inheritance tree (DIT)

* DIT is the length of the path from the node to the root of the tree

The greater values of DIT

1. The greater the number of methods (NOM) it is likely to inherit, making more COMPLEX to predict its behaviour
2. The greater the potential RE-USE of inherited methods
3. Small values of DIT in most of the system’s classes may be an indicator that designers are forsaking RE-USABILITY for simplicity of UNDERSTANDING
4. The objective is to achieve the appropriate trade of

Number of Children (NOC)

For any class in the inheritance tree, NOC is the number of immediate children of the class (the number of direct subclasses)

The greater values of NOC:

1. The greater is the RE-USE
2. The greater is the probability of **improper encapsulation**

Coupling between Object classes (CBO)

* For a class, C

Response for class (RFC)

It is the number of methods of the class plus the number of methods called by any of those methods.

Normally RFC is calculated up to the first method call level, and not through the transitive closure of allmethod calls.

Smaller numbers are better

* Larger numbers indicate increased complexity and debugging difficulties.

**Lecture 4: Bugs in the Code**

What is Software Reliability

The probability of failure-free software operation for a specified period of time in a specified environment

It denotes a product’s trustworthiness or dependability

Software Reliability

Software reliability not caused due to aging but **due to bugs**

The more the bugs, the lesser the reliability of the software

Still failures seem random, hence reliability theory can be applied

It should be able to run forever

What is a Bug?

A software bug occurs when one or more of the following five rules is true:

1. The software doesn’t do something that the product specification says it should do
2. The software does something that the product specification says it shouldn’t do
3. The software does something that the product specification doesn’t mention.
4. The software doesn’t do something that the product specification doesn’t mention but should
5. The software is difficult to understand, hard to use, slow, or – in the software tester’s eyes – will be viewed by the end user as just plain not right.

Why Do Bugs Occur?

1. The number one cause of software bugs is the specification, why?
   1. It is not written
   2. It is not complete
   3. It is not clear
   4. It is constantly changing
   5. It is not communicated well to the development team
2. The next largest source of bugs is the design.
   1. Inappropriate modelling
   2. Lack of modelling tools
   3. Time to market pressure

A lot of times developers do not consider design as an important step. Hence unable to identify design bugs introduces as inappropriate consideration of design choices and the alternatives

1. Coding Errors
   1. Software complexity
   2. Poor documentation
   3. Limited time
   4. Programmer skills

Bug Tracking Work Flow

A tester find a bug and report it

**Lecture 5: Finding Bugs in Software (Static Analysis)**

Introduction to debugging techniques

Debugging Technique:

* Threat Modeling: Look at design, write out and/or diagram what could go wrong
* Manual code reviews and inspection
* Automated Tools, e.g., static analysis

What is a Bug Pattern:

Definition: Bug patterns are recurring colrrelations between signalled errors and underlying bugs in a program.

Common pitfalls of a programming language that are documented so that developers can learn to avoid them

Challenge

* How to identify them
* How to treat them
* And how to prevent them

Example: Bug – Using Object Deserialization:

* Object deserialization of untrusted data

**Solution**

* Avoid deserializing object provided by remote users.
* If deserialization of objects from remote users cannot be avoided

Bug – Trust Boundary Violation:

A trust boundary can be thought of as line drawn through a program

On one side of the line, data is untrusted. On the other side of the line, data is assumed to be trustworthy

The purpose of validation logic is to allow data to safely cross the trust boundary – to move from untrusted to trusted.

A trust boundary violation occurs when a program

Bug Patterns

Not all bugs are subtle

Code Inspection

Manually examine source codeto look for bugs

Limitations

* Labour intensive
* Subjective: Source code might appear to be correct when it is not
  + Can you spot the typo

Debugger

A debugger is a special program used to analyse other programs in order to find bugs

A debugger is used to automatically detect bug pattern in program code

Idea:

* Use a program ot analyse your program for bugs
* Analyse statements, control flow, method calls, etc

Can Automated Program Analysis Work?

“Everything interesting about the behaviour of programs is undecidable”

In general, we can’t tell whether a program P has some property

The limits of static analysis (The Halting Problem)

1. Does program have bug X
2. Can program P reach state X?

Soundness

A bug detection system is sound if whenever there is a

Program Analysis Trade-Offs

Generally, most program analysis are conservative (i.e. they are sound and imprecise)

But, the detection of bugs in a program is an approximation that involvers

**Lecture 6: Introduction to Software Design**

* What is design?
* Discuss the different software design principles
* Broadly distinguish between different design patterns
* Identify anti-patterns
* History of design patterns

20% of a software engineering job is actually programming and 80% is actually designed

**What is Design?**

The process of envisioning and planning the creation of objects, interactive systems, sercies, etc.

* User-centered, i.e. users are at the heart of the design thinking approach
* About creating solutions for people, physical items or abstract systems to address a neeed or a problem

Software design is the process of creating a specification of a software artifact, indented to accomplish goals, using a set of primitive components and subject to constraints

Software design may refer to either:

“all the activity involved in conceptualiazing, framing, implementing, commissioning, and ultimately modifying complex sytems”

Or

“the activicty following requirements specification and beforeprogramming

**Design Concepts**

Fundamental software design principles PHAME Principles:

* Hierarchy
* Abstraction
* Modularisation and
* Encapsulation

**Principle of Hierarchy**

* Provides you the ability to break a system design into a taxonomical representation through the abstraction of packages and classes.
  + Enables you to understand different aspects of the hierarchies
* A subclass inherits state and behaviour in the form of variables and methods from its superclass and the rest of its ancestors.
  + As you drop down in the hierarchy, the classes become more and more specialised

**Principle of Abstraction**

* Abstraction is the property for which only the essential details are displayed to the user
* Data abstraction is the process of identifying only the required characteristics of an object ignoring the irrelevant details

**Principle of Modularisation**

“Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules’

* Strive to build modules that are cohesive (by grouping logically related abstractions).
* Aim for loosely coupled software (by minimizing)

**Principle of Encapsulation**

* Encapsulation involves bundling data and methods that work on that data within one unit, e.g., a class in Java
* Encapsulation provides explicit barriers among different abstractions and thus leads to a clear separation of concerns

The SOLID Principles of Software Design

SOLID principles that help softwaredevelopers design maintainable and extendable classes

* Single responsibility
* Open-closed
* Liskov substitution
* Interface segregation and
* Dependency inversion

**Lecture 7: The SOLID Principles of Software Design**

Single Responsibility Principle:

* A class should have one, and only one, reason to change

Open-Closed Principle:

* You should be able to extend a class’s behaviour, without modifying it

Liskov Substitution Principle:

* Derived classes must be substitutable for their base classes.

Interface Segregation Principle:

* Make fine grained interfaces that are client specific.

Dependency Inversion Principle:

* Depend on abstractions, not on concrete implementations

Design patterns helps designers apply best practices

Recurring design problems

* Most of the problem concern looking for optimal suitable arrangement of classes to provide a particular feature
* Can we add a new feature easily?
* Can we change how part of the system is implementd

Gang of Four

Creational Patterns:

* These are design patterns that provide a way to create objects while holding

Structural Patterns

Compound Patterns

These are patterns build from two or more design patterns:

* The most well known of these is the MVC or model view controller pattern
* The MVC combines strategy.

**Tutorial 2: Liskov’s Substitution Principle(LSP)**

When creating class hierarchies, ensure that the new derived classes just extend without replacing the functionality of old classes.

Liskov’s Substitution Principle states that if a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module

How to Achieve LSP

* **Demand no more:** The subclass would accept any arguments that the superclass would accept
* **Promise no less:** Any assumption that is valid when the superclass is used must be valid when the subclass is used

LSP – Design by Contract

**The contract of a method:**

* **Precondition:** A property that must be true before calling the method. If the precondition fails, the results are undefined
* **Postcondition:** A property that is guaranteed to be true after calling the method. If the postcondition fails, it shall not return.
* **Invariants:** A property that is guaranteed not to change after executing the method.

The LSP in terms of contract

A derived class is substitutable for its base class if

1. Its preconditions are not stronger than the base class method
2. Its postconditions are not weaker than the base class method
3. The invariants remain the same

LSP Rules

**Rule 1:**

**Rule 2:**

When you override a method in a base class, the postcondition of the overriding method should be stronger than the postcondition of the overridden method.

**Lecture 10 – The Visitor Design Pattern**

Dealing with Structures and Hierarchies

* Often times we need to manipulate a hierarchical collection of primitive and composite objects
  + Processing of a primitive object is handled one way, and processing of a composite object is handled differently
  + But having to query the “type” of each object before attempting to process it is not desirable
* How do we make surt that Leaf nodes and Composite nodes can be handled uniformly?
* So abstractions are necessary in designing hierarchical composite structures
* But each abstract must eventually be resolved to a concrete type in order to do actual work

Visitors Design Pattern

Assume the Client want to do arbitrarily many operations on file system objects

Context

Assume you want to design operation(s) across a heterogeneous composition of objects

You also want the operation(s) to be defined without changing the class of any of the objects in the collection because:

* You want to avoid “polluting” the node classes with these operations
* You don’t want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

Intent:

* Externalise and centralise operations on an object structure so that they can vary independently but still behave polymorphically.

Approach:

* Encapsulate a desired operation in a separate object, called a visitor
* The visitor object then traverses the elements

Implementation

Defining a Visitor:

1. Define the ComponentVisitor based class.
2. Add accept operation to Component base class and subclasses
3. Define a concrete Leaf and Composite visitors as subclass of ComponentVisitor.

Implementation: The Computer File System

* Adding accept operation to base class and subclass:
  + Observe that the semantics and structure for accept in all subclasses are the same

The client:

When a client needs an operation to be performed

Traverse the file system and visit each file in the system

Consequences

* The hierarchy of the structure remains intact

**Lecture 11: Static Code Analysis With JavaParser**

Static Analysis is the systematic examination of an abstraction fo program state space.

Systematic – ensure everything is checked in the same way

Abstract Syntax Tree (AST)

Intermediate program representation which depicts source code as a tree

* Defines a tree – preserves program hierarchy
* Nodes can be associated with properties
* Generated by parser

Getting Started with Java Parser

* Quickest way is to install JavaParser:
  + Create a simple Maven project in Eclipse
  + Add the following dependency to the POM.xml file

Java Parser API

* **JavaParser** class is what produces the AST from the code
* **CompilationUnit** is the root of the AST
* **Visitors** are classes which are useful to find specific parts of the AST

Visitor Class

How JavaParser applie the Visitor Design Pattern

Architecture is used when the aim is not to modify the underlying AST.