Introduction –Developing Maintainable Software

COMP2013 (AUT1 23-24)

Dr Marjahan Begum and Dr Horia A. Maior

Overview

- Introduction to the team for COMP2013
- Learning and Teaching Structure
- Explore your previous knowledge
- About this module
- This week!
- Software Quality - Maintainable Software
- Software Maintenance (high level)
- What is actually done in software maintenance?
- Final thoughts…

Dr Marjahan Begum – Assistant Professor

- PhD Computer Science Education from University of Nottingham (back in the days)
- Learning scientist interested in Computer Science
- Psychology of Computer Science
- Women in Computing
- Diversity in Computing
- Human aspect of software engineering (software teams)

Dr Horia A. Maior - Assistant Professor

- PhD and Postdoctoral from University of Nottingham
- Around since 20212
- Taught in Nigbo Campus in China

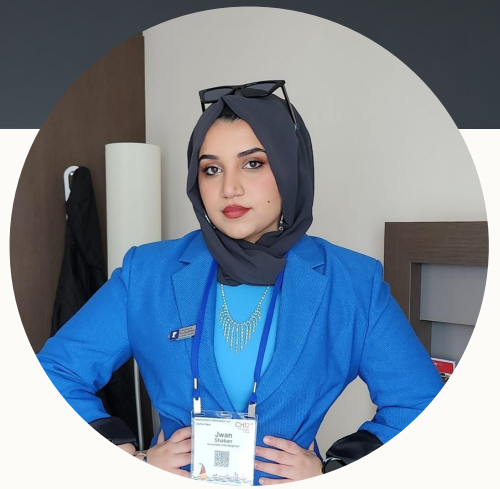
My research intersect:

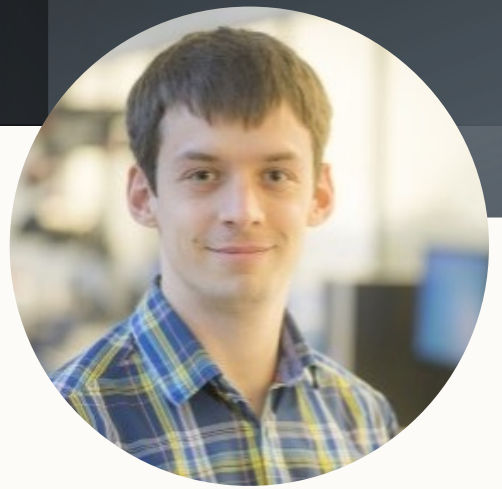
- Human Computer Interaction
- Brain Computer Interfaces
- Human Robot Interaction
- Human AI Interaction
- Responsible Research and Innovation

Amazing team of Teaching Support

- Dalma Kilic (PhD) – Teaching Associate
- Jwan Shaban – PhD
- Lewis Stuart – PhD
- Warren Jackson (PhD) – Teaching Associate
- Weiyao Meng(PhD) – Teaching Associate

Teaching Associate and PhDs



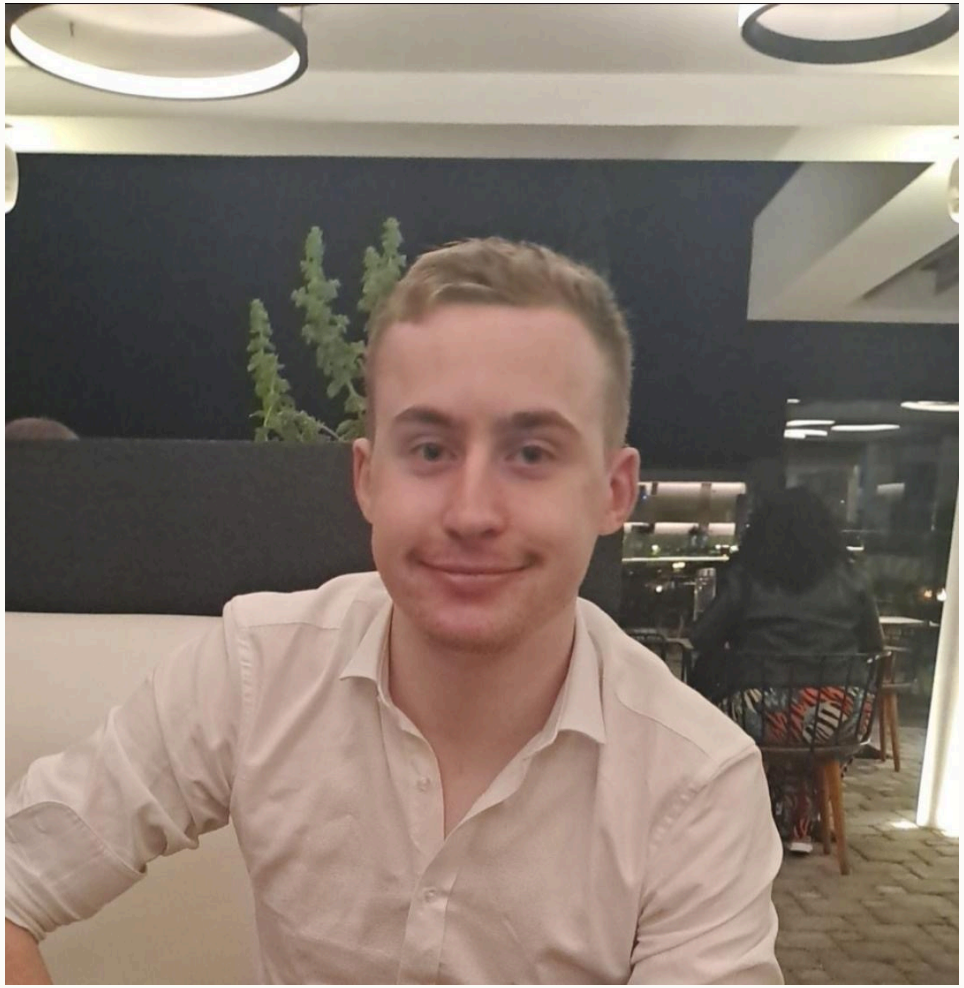


- Weiyao Meng – Teaching Associate
- Jwan Shaban – PhD
- Warren Jackson – Teaching Associate
- Dalma Kilic – Teaching Associate
- Lewis Stuart – PhD
- We will introduce all in the lab

Dalma Kilic

Lewis Stuart

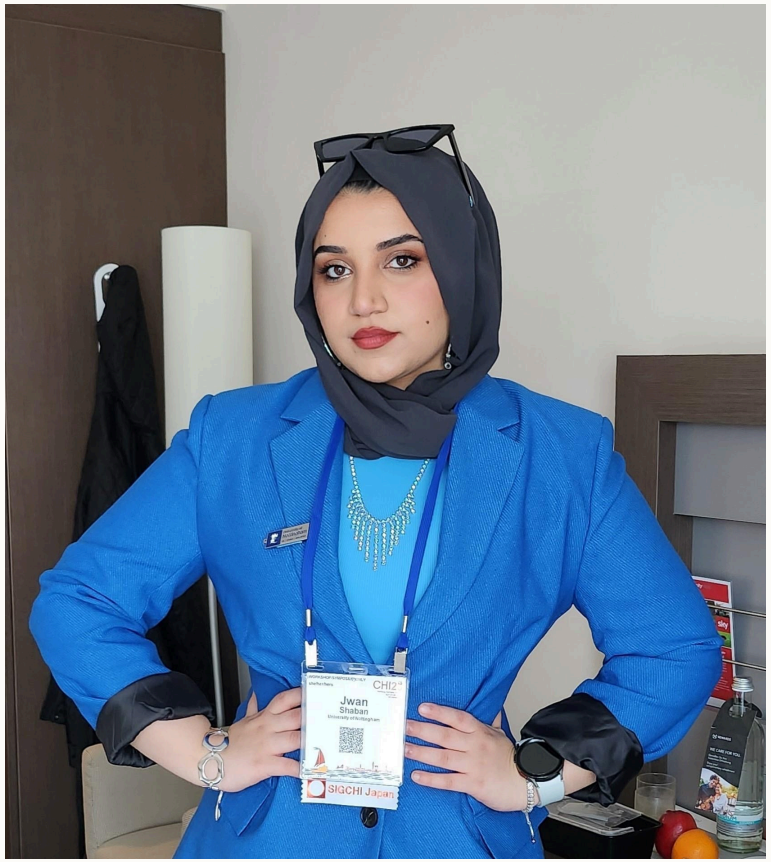
- Leeds University (first MEng Advanced Computer Science).
- Computer Vision Lab 3D reconstruction of plants from multiple views, using a UR5 robotic arm to capture data automatically.
- Taught on the Computer Graphics (COMP3011) module.



Jwan Shaban

- Tracking Mental Workload and personal informatics.

- BSc in software engineering from Manchester met University and a Masters in computer science from Nottingham.

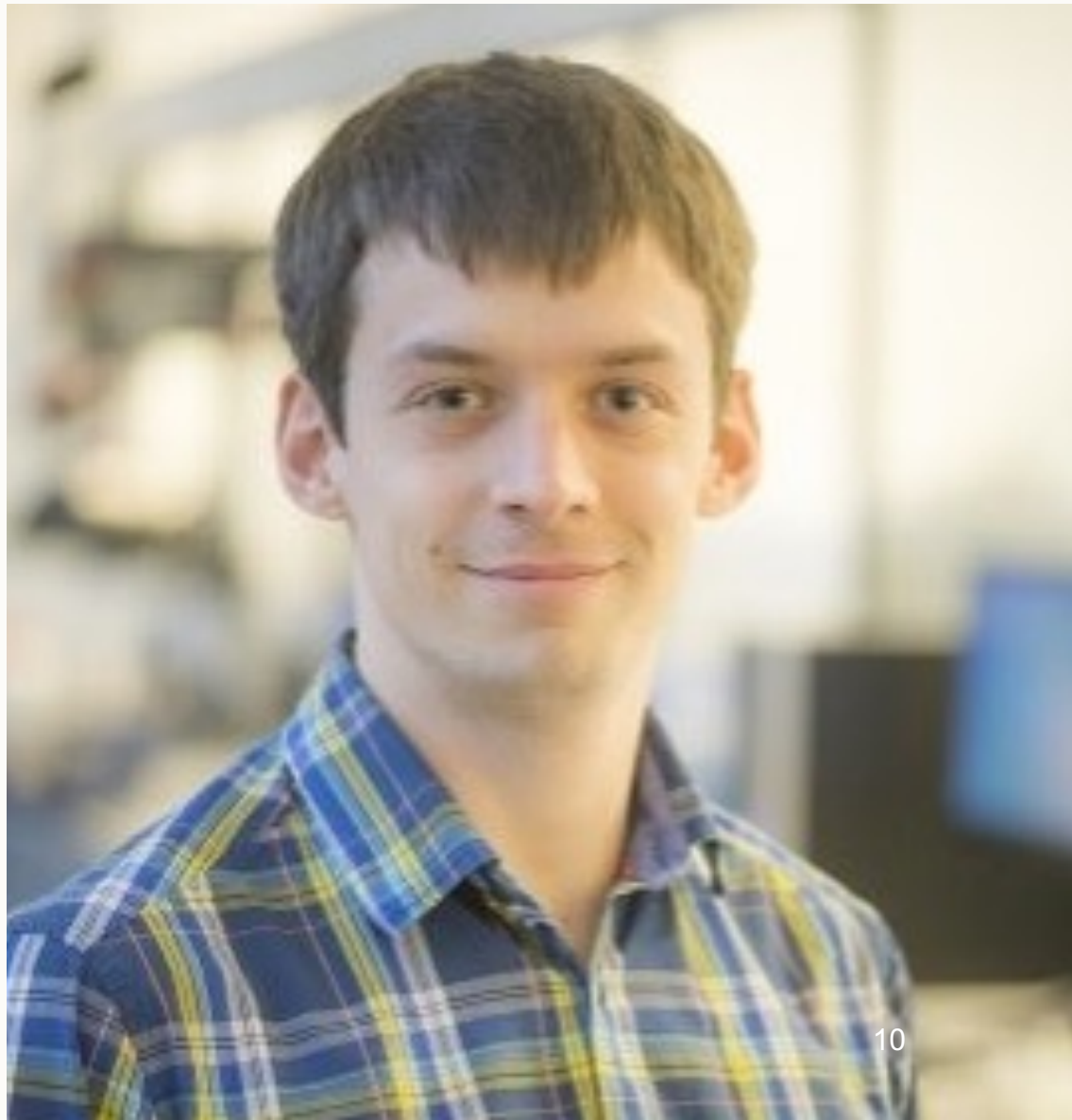


Dr. Weiyao Meng

- PhD at the Computational Optimisation and Learning (COL) Lab, machine learning for Vehicle Routing Problems with Time Windows.
- Co-convened COMP1008 (Fundamentals of Artificial Intelligence) in 2022-23 and has previously supported the Mathematics for Computer Scientists, Data Modelling and Analysis, Simulation and Optimisation for Decision Support.

Warren Jackson – Teaching Associates

- Experienced teacher in Computer Science
- PhD student within the Computational Optimisation and Learning (COL)



Learning and Teaching Structure

Office Hours

Dr Marjahan Begum

Tuesdays: 14:00 to 15:00

Room: B72

Dr Horia Maior

Thursdays: 14:00 to 15:00

Geospatial Engineering Building (3rd floor)

Learning and Teaching Structure

- **Lecture A** is on Mondays : the Jubilee Campus Business School South Auditorium from 16:00 to 18:00.
- **Lecture B** is on Thursdays 5th: the Jubilee Campus Business School South Auditorium from 13:00 to 14:00
- **Lab** is Fridays (A07, A32, Computer Science Atrium).

Assessment

- Coursework (75%)
  - Individual coursework
  - Refactor and extend an existing code base
  - We aim to release it by the end of week 5
  - A milestone setting up infrastructure and design documentations week 8
- Exam (25%)
  - ROGO exam (90 minutes)

How to participate

- All communication will be through Moodle forum and teams
- Some individuals email will be answered depending on circumstances
- Team channel – please help each other without compromising academic integrity

ChatGPT

Tell me about your previous knowledge and skills?

Polls

- Tell me technologies you are familiar with from year 1 and previous experience?

- How do you think this module is related to previous years?

Do these people remind you of something??

- You have had Java overview lectures from Jason
- Max has introduced you to Agile
- Graham has enticed you with his Haskell
- Some of you are very accomplished programmers
- You will have seen some of the content before

- Practical and prepares you for the real job..

Previous topics you will have covered

- Programming in Java:
  - Classes and interfaces
  - Objects
  - Arrays
  - Strings
  - Inheritance
  - Java Collections
  - Dealing with exceptions
  - Software testing
  - Refactoring

- Tools

Previous topics you will have covered

- Software Engineering
  - Requirements Engineering
  - Prototyping
  - Object Oriented Design
  - Implementation Strategies
  - Debugging
  - Release, Acceptance, and User Testing
  - Software Quality
  - Software Evolution
  - Agile Methodologies
  - Project Management

So ... What is the difference?

What do you want to get out of this is module?

The overall learning objectives of this module

- To ensure you understand that software maintenance is central to software development
  - Maintainable software and software maintenance is essential to developing large codebases and code written and updated by multiple people
  - Software can only be maintained if good practices of coding and design are followed

- To ensure you can write maintainable code
  - So that you understand that software maintenance is not an afterthought in the software process models and lifecycle

- To transform you from a 'coding caterpillar' to a 'developer butterfly'

This quote is from Julie :)

COMP2013 is all about BIG software projects

- It's about ...
  - How to create them from scratch so you can maintain them more easily later
  - How to approach an existing large mass/mess of code!
  - The tools you need to maintain this code "collaboratively"

- All this is very practical advice that you will find useful in the real world
  - We (hopefully) will have some guest lecturers to provide a real-world take on this
  - It should give you lots of pointers for COMP2002 (the group project)

How best to get most out of this module?

- Ask questions and clarify the requirements for this module through Team's channel and during the lab.

- Engage with the in lectures and lab by asking questions, attempt the lab and come to labs with questions, read the coursework line by line and clarify where required.

- Please keep in mind: The module is not about teaching you the latest Java and software engineering technologies and libraries, but **it's about learning the principles** of "how to develop maintainable software" and "how to maintain software".

What tools, languages and software?

Relevant Software

- Software we use in this module:
  - Java 12+ (requires 64 bit system > use the virtual desktop if you don't have a 64 bit system)
  - IntelliJ IDEA (this is the IDE to be used for your coursework)
  - Visual Paradigm (also available as online solution, but requires registration)
  - JavaFX 12+ graphics library and Gluon Scene Builder
  - Maven/Gradle
  - Git
- Our recommendation: Use the latest (Long-Term-Support) versions

What are we doing this week?

Topics for this Week

- Lecture 01A
  - Welcome to the module
  - Developing maintainable software
  - Why COMP2013?
  - Challenges of software maintenance
- Lecture 01B
  - OO and Java programming refresher
  - Lab sheet
- Lab 01
  - IntelliJ basics
  - Practicing Java basics
  - Working with existing cod

Learning Outcomes for this Week

- At the end of this week …
  - You should understand what developing maintainable software is about
  - You should understand why this module is useful
  - You should be set up for future lab sessions
  - You should have a good working knowledge of the basics of the object-oriented concepts, Java, and the IntelliJ IDE

What do you think developing maintainable software is about?

What are the relationships?

Maintainable Software (Qualities)

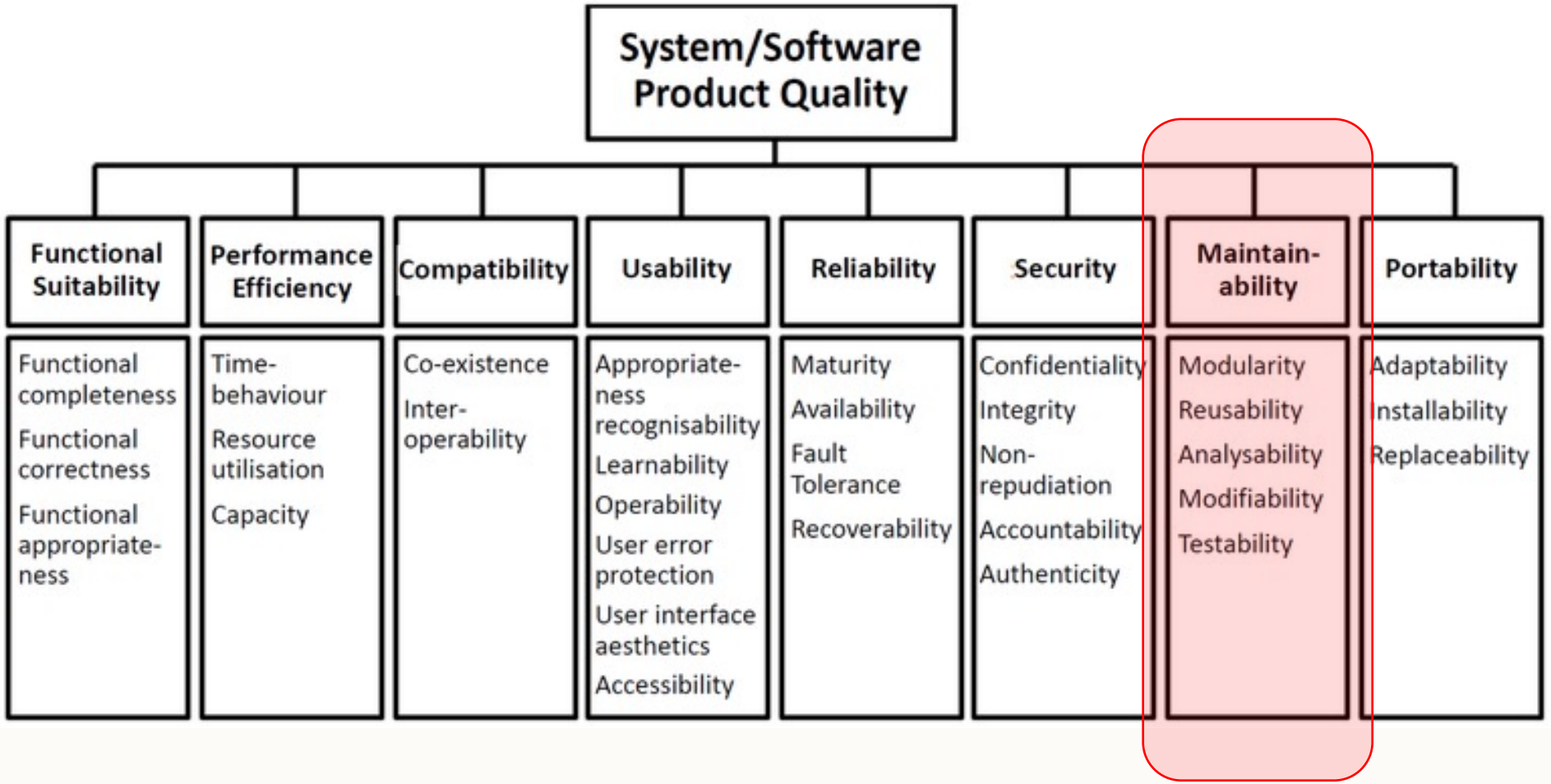
Software Maintenance

Maintainable software

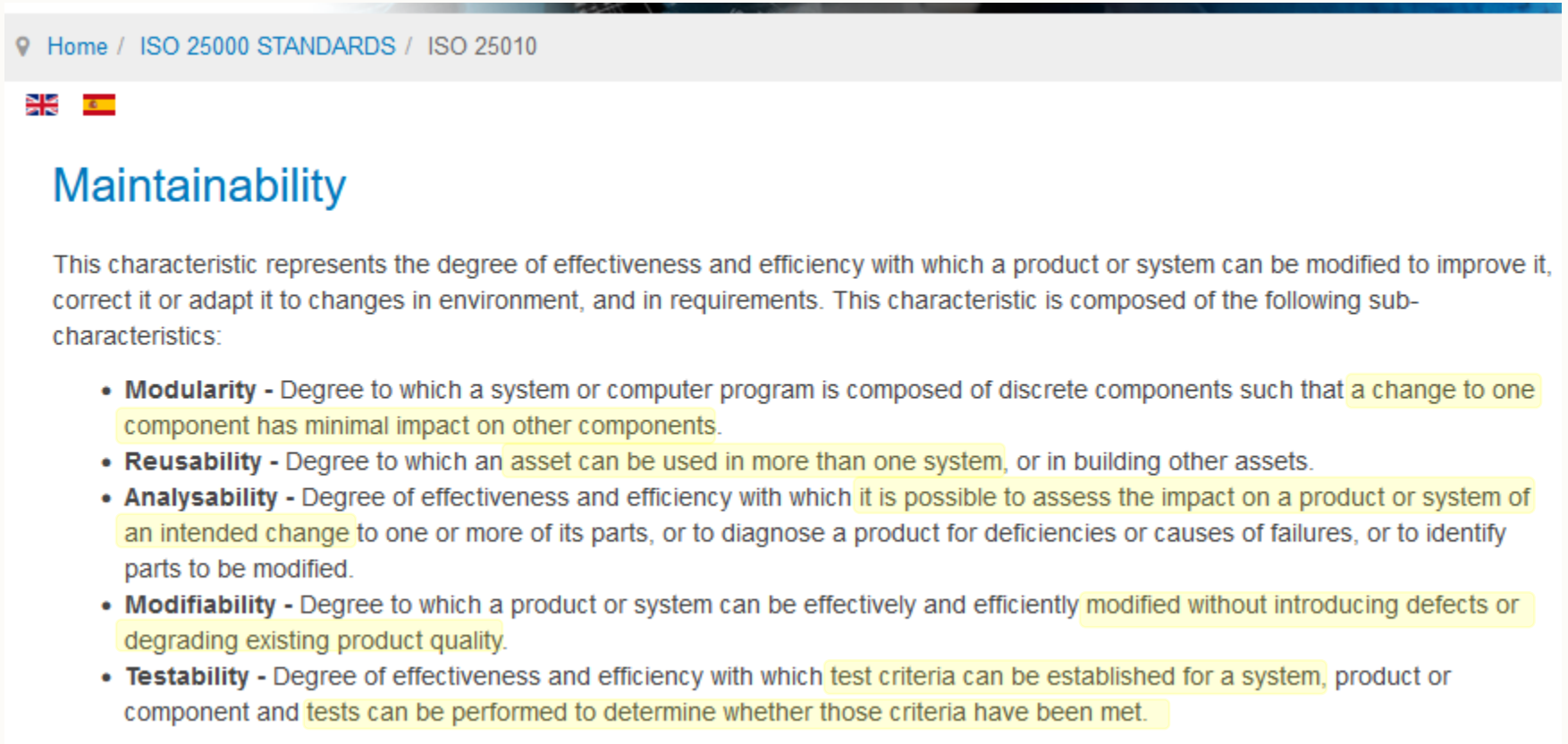
Software Quality – principles and guidelines – wish list

Developing Maintainable Software

- ISO 25010



Developing Maintainable Software



Home / ISO 25000 STANDARDS / ISO 25010

Maintainability

This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements. This characteristic is composed of the following sub-characteristics:

- **Modularity -** Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
- **Reusability -** Degree to which an asset can be used in more than one system, or in building other assets.
- **Analysability -** Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
- **Modifiability -** Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
- **Testability -** Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/57-maintainability

Maintainability

- **Modularity**
  - Discrete components
  - Change to one component have minimal impact on others
    - Low ripple effect
- **Reusability**
  - Reuse Reuse Reuse
- **Analysability**
  - Effectively and Efficiently assess impact of individual component

Maintainability

- **Modifiability**
  - Effectively and efficiently modify code
    - Without defects/degrading product quality

- **Testability**
  - Effectively and efficiently write test criteria
  - Test can be performed to evaluate software qualities

Developing Maintainable Software

- Three principles for developing maintainable software:
  - Maintainability benefits most from adhering to simple guidelines
    - e.g. never write methods that have more than 15 lines of code
  - Maintainability is not an afterthought, but should be addressed from the very beginning of a development project
  - Some violations are worse than others; the more a software system complies with the guidelines, the more maintainable it is

Visser et al (2016)

Developing Maintainable Software

- Some simple guidelines to develop maintainable software:
  1. Write short units (constructors/methods) of code
  2. Write simple units of code
  3. Write code once
  4. Keep unit interfaces small
  5. Separate concerns in modules (classes)
  6. Couple architecture components loosely
  7. Keep your codebase small
  8. Automate your development pipeline and tests
  9. Write clean code

Visser et al (2016)

A Poem

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.[a]
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.[b]
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

Software Maintenance

Software Quality – checking of it the maintainability

Software Maintenance

- What does it involve?
  - "Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."

IEEE Standard for Software Maintenance:
https://ieeexplore.ieee.org/document/257623

- But we also need to consider how we can reduce the effort of maintenance!
  - Building software that is easy to maintain and extend in the first place!

Resources Spend on Initial Development vs. Maintenance

- It's important to learn about maintenance
- It's important to build maintainable software

	Resources spend	
Author	initial development	maintenance
Daniel D. Galorath	25%	75%
Stephen R. Schach	33%	67%
Thomas M. Pigoski	<20%	>80%
Robert L. Glass	20-60%	40% - 80%
Jussi Koskinen	<10%	>90%

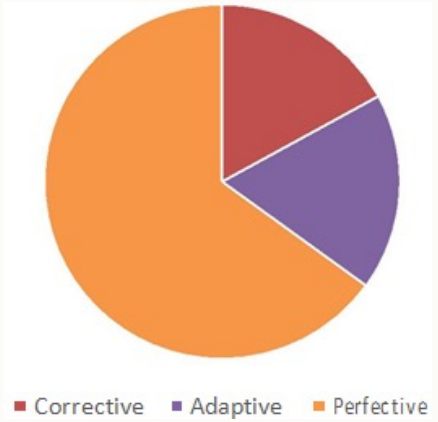
http://blog.lookfar.com/blog/2016/10/21/software-maintenance-understanding-and-estimating-costs/

Resources Spend on Initial Development vs. Maintenance

Three (or four, depending on authors) Main Categories of Maintenance

- Corrective Maintenance
  - Finding and fixing errors in the system
    - Example: Fixing bugs

Maintenance effort

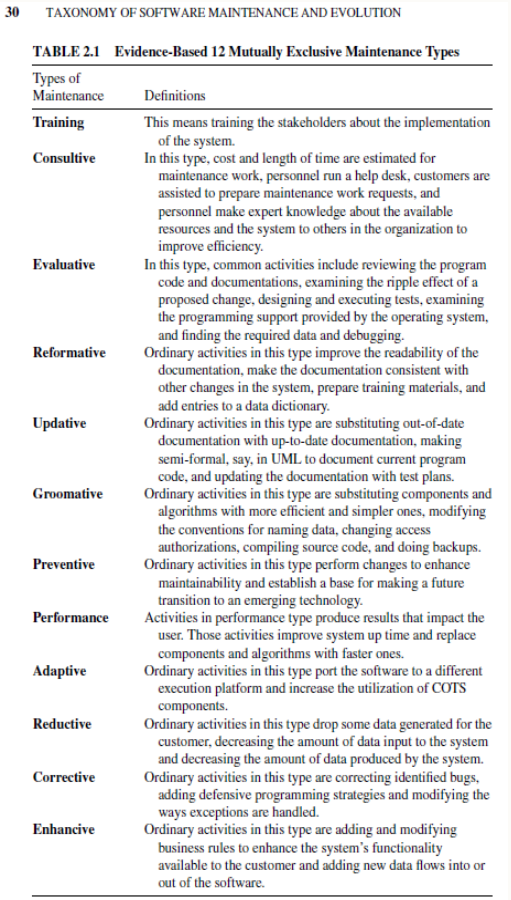


- Adaptive Maintenance
  - The system has to be adapted to changes in the environment in which it operates
    - Example: Updating links to resources after change

- Perfective + Preventive Maintenance
  - Users of the system (and/or other stakeholders) have new or changed requirements
  - Ways are identified to increase quality and prevent future bugs from occurring

Actually there can be lots more!

- Maintenance vs Evolution?
  - Maintenance:
    - Preserving software in a working state
  - Evolution:
    - Improving software (e.g changing requirement)
- What we will learn will largely be applicable to both



30 TAXONOMY OF SOFTWARE MAINTENANCE AND EVOLUTION

TABLE 2.1 Evidence-Based 12 Mutually Exclusive Maintenance Types

Types of Maintenance	Definitions
Training	This means training the stakeholders about the implementation of the system.
Consultive	In this type, cost and length of time are estimated for maintenance work, personnel run a help desk, customers are assisted to prepare maintenance work requests, and personnel make expert knowledge about the available resources and the system to others in the organization to improve efficiency.
Evaluative	In this type, common activities include reviewing the program code and documentations, examining the ripple effect of a proposed change, designing and executing tests, examining the programming support provided by the operating system, and finding the required data and debugging.
Reformative	Ordinary activities in this type improve the readability of the documentation, make the documentation consistent with other changes in the system, prepare training materials, and add entries to a data dictionary.
Updative	Ordinary activities in this type are substituting out-of-date documentation with up-to-date documentation, making semi-formal, say, in UML to document current program code, and updating the documentation with test plans.
Groomative	Ordinary activities in this type are substituting components and algorithms with more efficient and simpler ones, modifying the conventions for naming data, changing access authorizations, compiling source code, and doing backups.
Preventive	Ordinary activities in this type perform changes to enhance maintainability and establish a base for making a future transition to an emerging technology.
Performance	Activities in performance type produce results that impact the user. Those activities improve system up time and replace components and algorithms with faster ones.
Adaptive	Ordinary activities in this type port the software to a different execution platform and increase the utilization of COTS components.
Reductive	Ordinary activities in this type drop some data generated for the customer, decreasing the amount of data input to the system and decreasing the amount of data produced by the system.
Corrective	Ordinary activities in this type are correcting identified bugs, adding defensive programming strategies and modifying the ways exceptions are handled.
Enhancive	Ordinary activities in this type are adding and modifying business rules to enhance the system's functionality available to the customer and adding new data flows into or out of the software.

Tripathy and Naik (2015)

Building vs Maintaining Software

Visual Studio MAGAZINE

HOME NEWS TIPS & HOW-TO NEWSLETTERS W

VISUAL STUDIO | VISUAL STUDIO CODE | C#/VB | .NET CORE | XAMARIN/MOBILE | TYPESCRIPT/BLAZ

PRACTICAL .NET

In Praise of the Maintenance Programmer

The developers building new applications are very nice people, of course. But the real heroes of the programming world are the developers maintaining and extending existing applications.

By Peter Vogel 12/16/2014

Back in 1984, I was fresh out of school and ready to be hired as a developer. I was hired by a large multi-national corporation ... and immediately put on the maintenance team for an existing application. At the time, that decision seemed reasonable. In retrospect, it seems spectacularly stupid. Actually, "crazy" would be a better description.

Maintenance is *much* harder than new development.

https://visualstudiomagazine.com/articles/2014/12/01/in-praise-of-the-maintenance-programmer.aspx

What is actually done in software maintenance?

What is involved in Software Maintenance?

- Understanding the client (need to listen and translate into requirements)

- Understanding the existing code (often harder than you think)
- Refactoring the existing code (apply the guidelines for developing maintainable code)
- Extending the existing code (adding functionality)

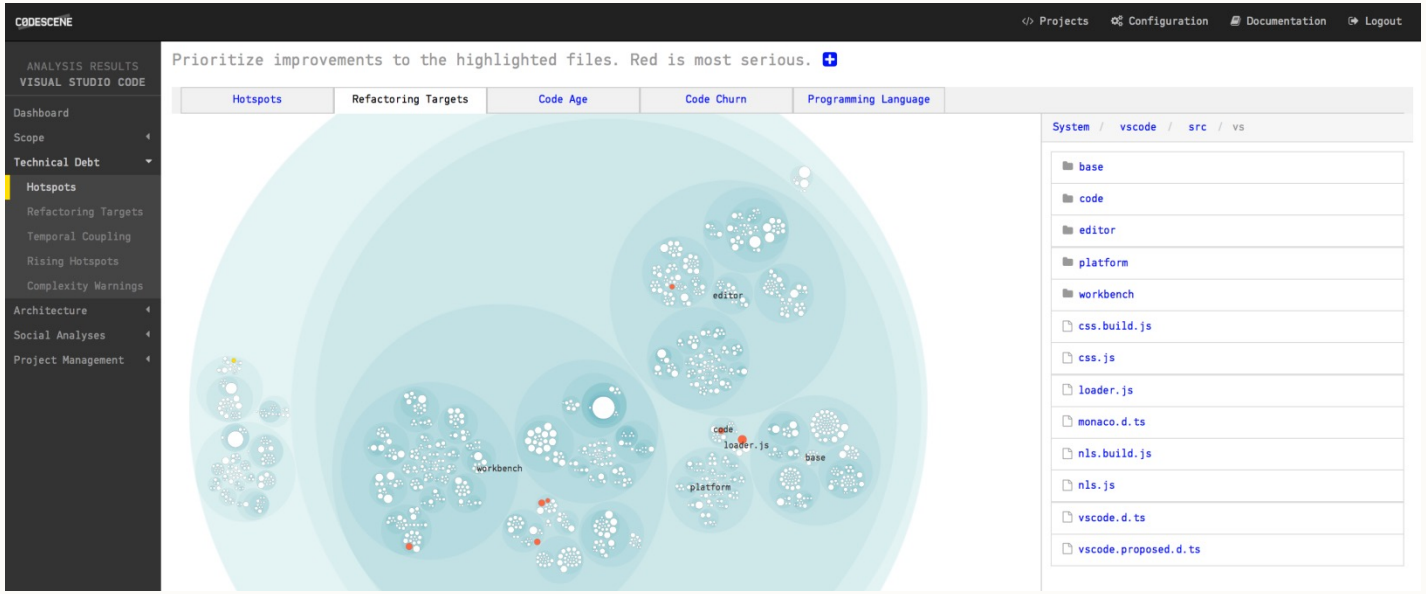
- Working as a team
- Managing client expectations (deliver what the client wants to a high standard in time)
- Managing the maintenance process (same as development; need to be organised)

Maintenance as a Murder Mystery!

- Fixing a problem requires in depth knowledge of the crime scene, some specialist tools, and detective work!
  - Where did the crime take place?
  - Who committed the crime?
    - (not a witch hunt – but useful to know to try to locate related bugs)
  - Who and what else was involved?
  - Part detective, part programmer

Maintenance as a Murder Mystery!

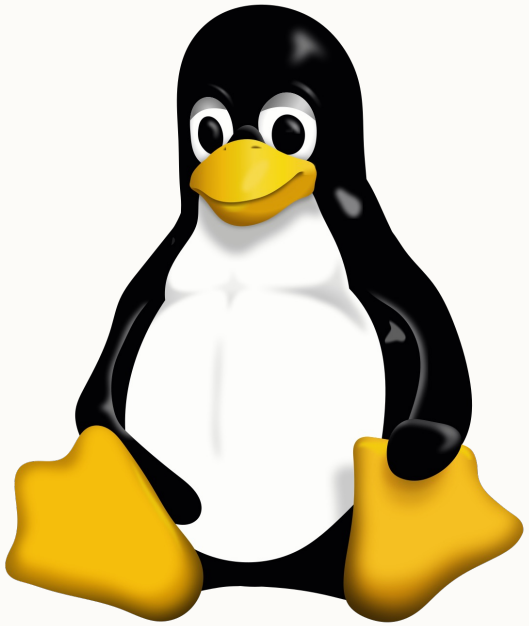
- Some tools take this analogy a step further
  - e.g. CodeScene (https://codescene.io/)
    - Identifies patterns in the evolution of your code
    - Software forensics



https://www.youtube.com/watch?v=n4P_I9rXKbE

How big is big? How complex is complex?

- Example: Linux Kernel
  - How many lines of code?
    - 15 million lines of code (204.5 for a full Linux distribution)
  - How many patches per release?
    - Approx. 10,000 patches in each release
    - Releases every 2-3 month
  - How many developers?
    - Each release by 1,000s of developers
    - 200 corporations

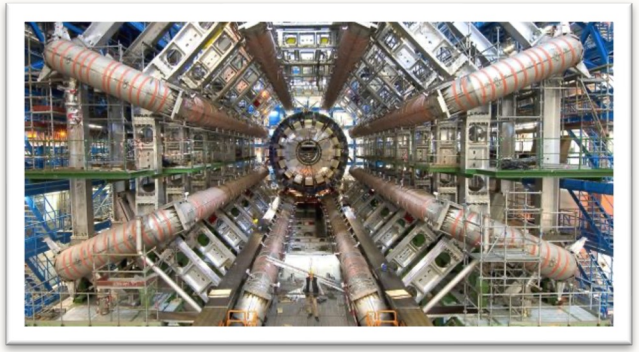


http://www.linuxfoundation.org

How big is big?

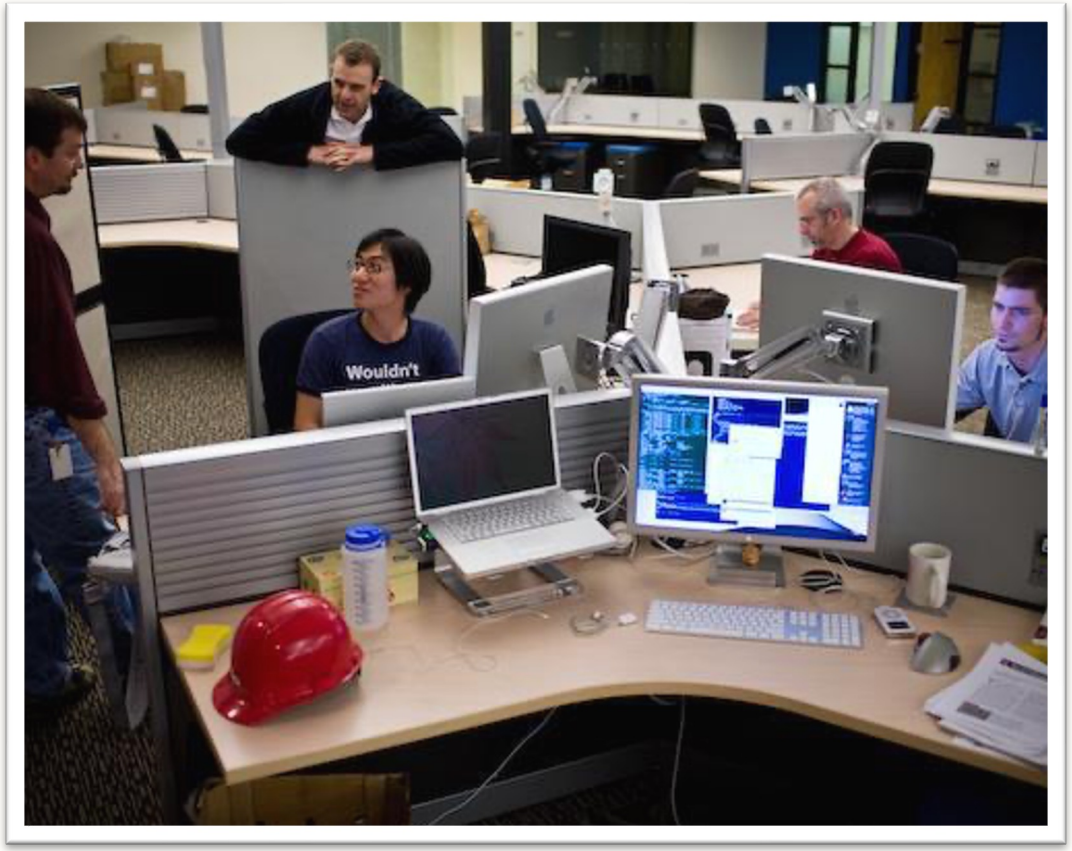
- Curiosity: 2.5 million lines
- CERN: 3 million lines (+50 million in projects)
  - Large hadron collider (particle accelerator)
- Windows: 50 million lines
- The code that guided the Apollo 11 module in 1969: 100,000 lines

- Of course, it is not just lines of code
  - 4000 people worked on the code base of Windows 10



Your New Job

- You've just started in a small company as a developer in a small team
- The company has just acquired a new software system
- You have been tasked with understanding and working with the code
- What do you do?



Your New Job

- Some suggestions from last year's cohort
  - Try to understand the code
    - How do you do it?
  - Look for the documentation
    - Let's hope the person who has written the code base has done the documentation!
  - Talk to team members
    - Scary thought? Need to learn to communicate
    - This is one of the key objectives of the group project!

What is involved in Software Maintenance?

- Understanding the client

- Understanding the code
- Refactoring the code
- Extending the code

- Working as a team
- Managing client expectations
- Managing maintenance process

What is involved in Software Maintenance?

- Understanding the client

- Understanding the code (1a/b)
- Refactoring the code
- Extending the code (2)

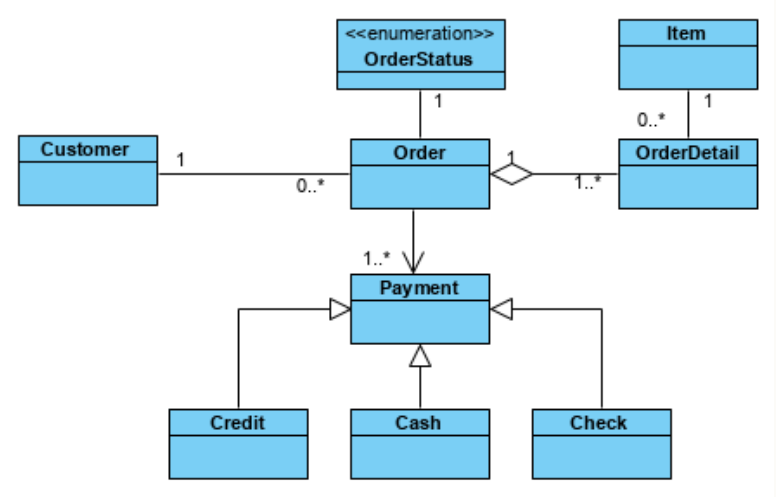
- Working as a team
- Managing client expectations
- Managing maintenance process (3)

1a. Making Sense of System Structure

- There could be hundreds or thousands of source code files in a project
- Program comprehension accounts for 50% of the total effort expended throughout the life cycle of a software system (Tripathy and Naik 2015).
- How to make sense of them?
  - It makes sense to look at relationships between classes
  - To do this we can use visualisation techniques
  - Understanding OOP is critical here

1a. Making Sense of System Structure

- Class Diagrams
  - Show a set of classes and their relationships
  - Addresses static design view of a system

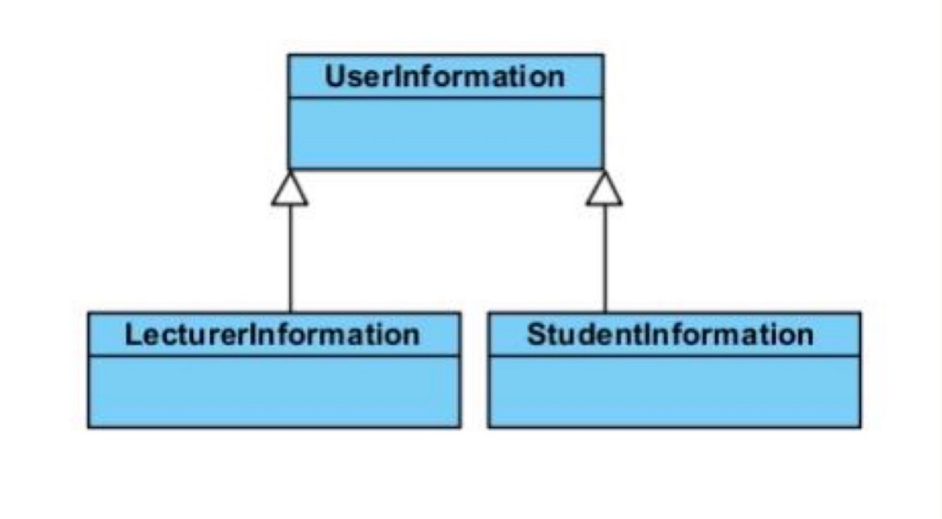


  - Classes
    - Blueprints (templates) for objects
    - Contain data/information and perform operations

What is in the three compartments?

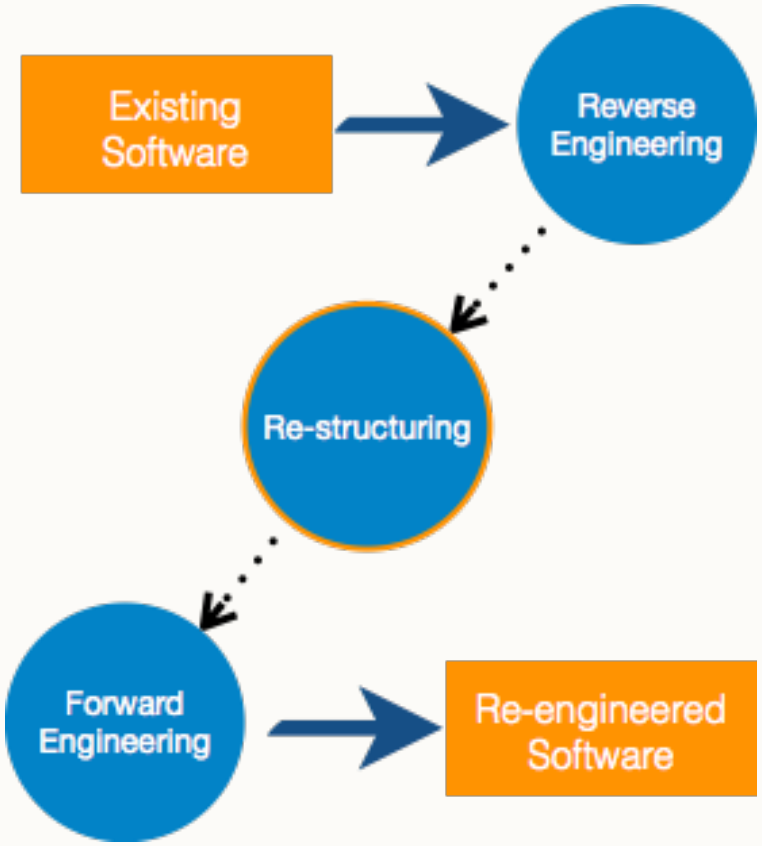
1a. Making Sense of System Structure

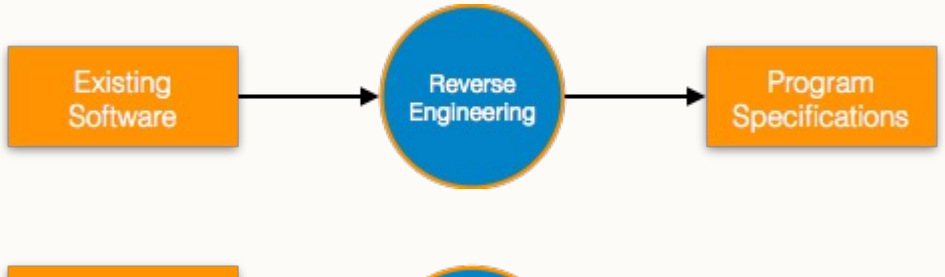
- Class diagrams offer a great way of showing inheritance and polymorphism

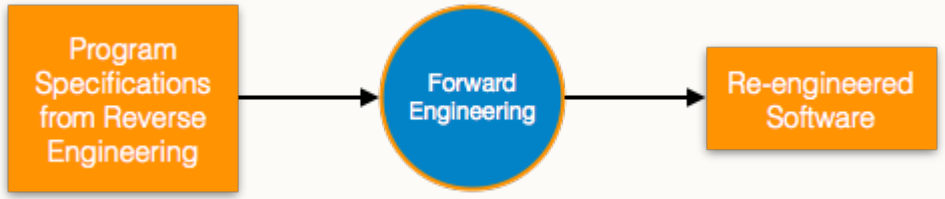


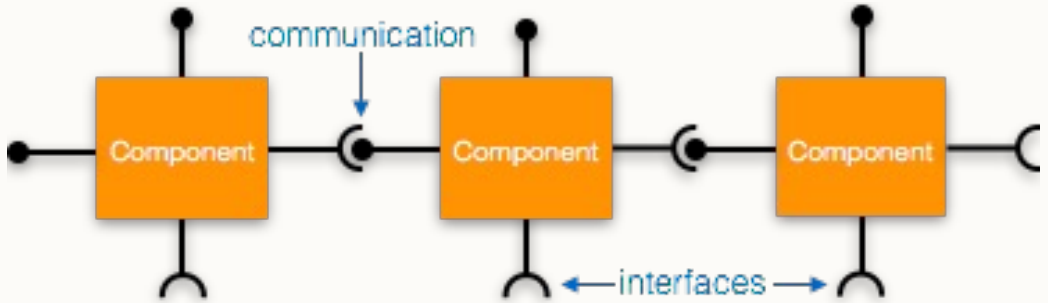
Which of the two do we see in this diagram?

1a. Reverse Engineering









https://sceweb.sce.uhcl.edu/helm/WEBPAGES-SoftwareEngineering/myfiles/TableContents/Module-13/software_maintenance_overview.html

1a. Reverse Engineering

- Transform these classes into a simple class diagram and do some maintenance

```java
package ZooSystem;

public class Piranha extends Fish{
}
```

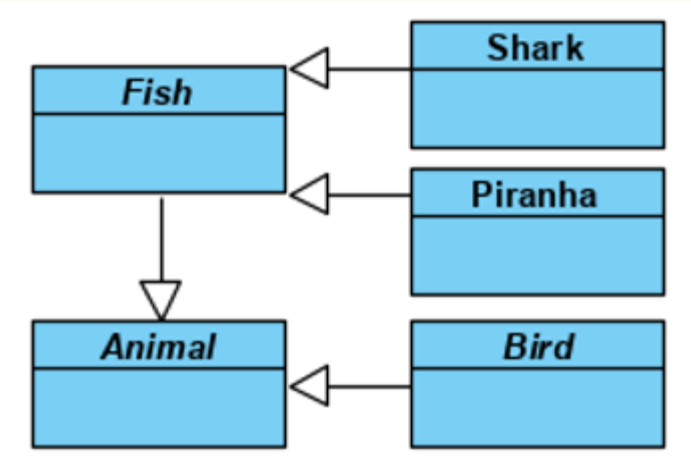
```java
package ZooSystem;

public abstract class Bird extends Animal{
}
```

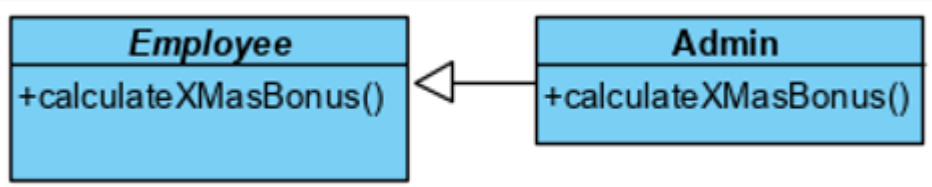
```java
package ZooSystem;

public class Shark extends Fish{
}
```

```java
package ZooSystem;

public class Admin extends Employee{

    @Override
    public int calculateChristmasBonus(){
        int bonus=(int)((double)getSalary()*0.08);
        return bonus;
    }

    public Admin(String name){
        super();
        setEmployeeName(name);
    }
}
```

1a. Reverse Engineering

- Simple class diagram (only including relevant methods)





1a. UML in Practice

- Class diagrams can help sometimes (but not always)
- They summarise the content of classes and relationships between them in the simplest possible way
- Does it make sense to build a detailed *class diagram* for an entire BIG project?
  - There are other methods to do this and other UML techniques to look at software at a higher level (e.g. deployment diagrams)
- Code can be generated directly from UML for maximum consistency

1a. UML in Pract

- Caution! This is still a relatively small number of classes
- Use the diagrams sensibly, where they can help
- In this case using a high-level class diagram would be advisable

1b. Understanding the code itself

- As well as understanding the structure, we need to understand what the code actually does
- We need to understand the code itself to make the right decisions for producing robust and maintainable code

1b. Understanding the code itself

- Let's consider adding a method to set an employee phone number
  - Which class should it go in?
    - In Admin, or the abstract class Employee?
  - Should it be public or private?
  - What variable will you use?
    - What type?
  - What if another subtype of Employee already has a telephone method?
    - And what if one doesn't have a fixed telephone (**mobile number is share by a number of employees**?)

```
1   package ZooSystem;
2
3   public class Admin extends Employee{
4
5       @Override
6       public int calculateChristmasBonus(){
7           int bonus=(int)((double)getSalary()*0.08);
8           return bonus;
9       }
10
11      public Admin(String name){
12          super();
13          setEmployeeName(name);
14      }
15  }
```

1b. Following conventions vs being smart

- What does this C snippet do? Could you extend it easily?

```
float [redacted]( float number )
{
        long i;
        float x2, y;
        const float threehalfs = 1.5F;

        x2 = number * 0.5F;
        y  = number;
        i  = * ( long * ) &y;
        i  = 0x5f3759df - ( i >> 1 );
        y  = * ( float * ) &i;
        y  = y * ( threehalfs - ( x2 * y * y ) );
//      y  = y * ( threehalfs - ( x2 * y * y ) );

        return y;
}
```

- No comments
- Poor variable names
- Obscure coding
- Redundant commenting out

Actually, it's Quake3's Fast InvSqrt()

Tonight, get out your last piece of Java coursework from Y1, and see if you still understand what you did 6 month ago

Not sure about this.

1b. Following conventions vs being smart

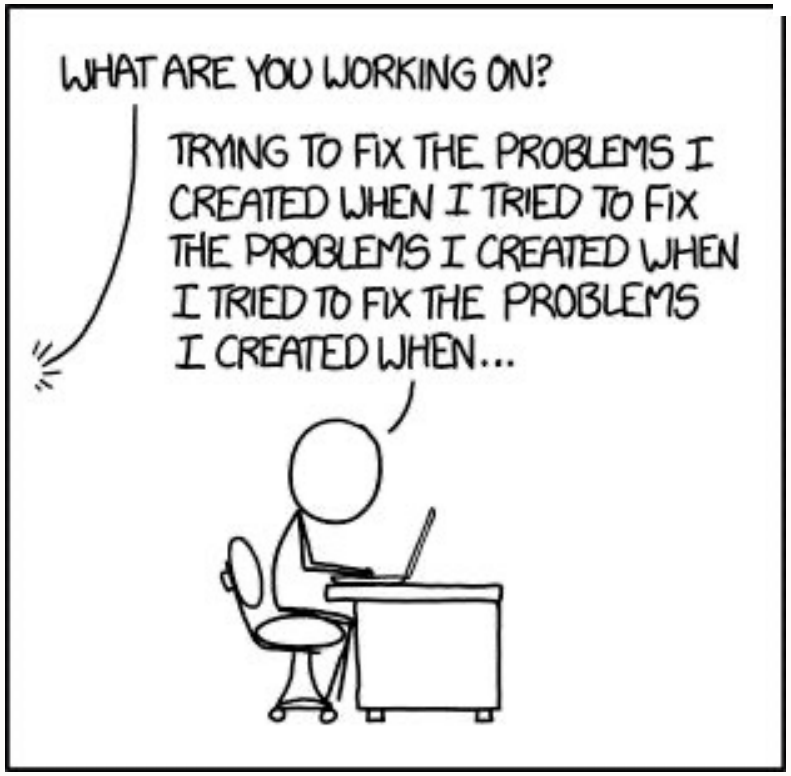
- Actually, there were some comments – *how helpful are these?*

```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;                       // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );               // what the f**k?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed

    return y;
}
```

2. Extending Existing Code

- When you change existing code you risk breaking it
  - How do you know you haven't broken it?
  - What kind of testing do you need?

- Regression testing
  - A type of software testing to confirm that a recent program or code change has not adversely affected existing features
  - Running all your unit tests again



3. Software Maintenance Management

- This involves the management of people and not the management of conditional statements or linked lists
- The most under-appreciated part of our course, yet deep understanding of these processes often leads to successful future employment

- You are no longer the "Lone Coder" – from now on we encourage you to think like a pro and write your code as if someone else will need it
- You have looked at Git and version control in Y1; we will be going over it again ... and again ... and again ... until you use it by habit.
- Understanding how open source and software licensing works

3. Software Maintenance Management

- Use established processes to manage the design and implementation of changes

- Our Industrial Steering Group wants you to learn about how to actually perform agile development ... not just to know the word
  - Team meetings and a design process – e.g. Scrum
  - Team communication
  - Source code management (e.g. GIT)
  - Server-side testing (continuous integration server)
- This means talking to other people …

Some final remarks

Polls

- See what they understood about the mod

Questions?


why?
how?
WHAT?
WHERE?
WHO?
WHEN?
WHY?
HOW?
when?
WHAT?
WHO?
WHY?
when?
how?
how?
WHEN?
who?
WHEN?
WHERE?
HOW?
what?
what?
WHEN?
where?
why?
WHO?
WHAT?
how?
WHY?
WHEN?
how?
WHAT?
when?
where?
HOW?
WHEN?
WHAT?

References

- Visser et al (2016) Building Maintainable Software - Java Edition
- Tripathy and Naik (2015) Software Evolution and Maintenance

- These slides have been adapted and reused from 2022/2023 – Thank you Peer and Robert (Bob)