Inf2C - Software Engineering

2018/19

Lecture notes* by s1709906 based on Paul Jackson's lecture slides.

* basically the slides typed up in a single document

Lecture 1 - Requirements Engineering

Kinds of Requirements

- Functional Requirements (services): What the system should do
- Non-functional Requirements (constraints or quality reqs.)
  - How fast it should do it
  - How seldom it should fail
  - What standards it should conform to
  - How easy it is to use
  - Can be workarounds for functional requirements
  - User experience often shaped by non-functional

The distinction between functional and non-functional requirements not always clear-cut.
e.g. Security might initially be a non-functional requirement, but, when requirements refined, it might result in addition of authorisation functionality.

Requirements vs. Design

Requirements try to avoid design (express *what* is desired and not *how* what is desired should be realised)

Stakeholders in requirements

Requirements are usually relevant to multiple stakeholders:

- End users
- Customers paying for software
- Government regulators
- System architects
- Software developers
- Software testers
- etc.

Requirements capture process

Activities: gathering (elicitation), sorting out (analysis), writing down (specification), checking (validation)
* often overlaps, not in strict sequence, and iterated
Why is it critical?

- Faulty requirement capture can have huge knock-on consequences in the later software process activities
  - Major source of project failure according to Standish CHAOS reports
- One motivation for incremental nature of Agile processes

Requirements elicitation sources

Goals: high-level objectives of software
Domain Knowledge: Essential for understanding requirements
Stakeholders: Vital, but they may find expressing requirements difficult
Business rules: e.g. University regulations for course registration
Operational Environment: e.g. concerning timing and performance
Organisational Environment: how does software fit with existing practices?

Requirements elicitation techniques

Interviews

Traditional method: ask them what they want or currently do

Can be challenging: jargon confusing, interviewees omit information obvious to them

Important to: be open-minded (requirements may differ from those pre-conceived), prepare starting questions (e.g. from first-cut proposal for requirements, as they help to focus dialogue)

Scenarios

Typical possible interactions with the system

- Provide context or framework for questions
- Allow 'what if' or 'how would you do this' questions
- Easy for stakeholders to relate to
- Can be captured as user stories and use cases

Prototypes

e.g. screen mock-ups, storyboards, early versions of systems

Advantages: like scenarios, but more 'real'; high quality feedback; often help resolve ambiguities

Disadvantages: expensive

Observation

Suitable if replacing existing system or business process (nuances can make or break a software product)

Immersive method -> expensive!

Advantages: helps with surfacing complex/subtle tasks and processes, finding the nuances that people never tell you

Disadvantages: expensive, not so good if innovating (e.g. capturing requirements for a touchscreen smartphone 15 years ago)

Requirements Analysis

The process of getting to a single consistent set of requirements, classified and prioritised usefully, that will actually be implemented.

Why? This is because requirements elicitation often produces a set of requirements that

- contains conflicts (e.g. one stakeholder wants one-click access to data, another requires password protection)
- Is too large for all requirements to be implemented

Requirements Specification

Requirements can be recorded in various ways, perhaps using:

- very informal means (e.g. handwritten cards, in agile development)
- A document written in careful structured English

  e.g. `3.1.4.4 The system shall... // for an essential feature`

  `3.1.4.5 The system should... // for a desirable feature`
- Use case models with supporting text
- A formal specification in a mathematically-based language

Phrasing of requirements specifications

Different phrasing of requirements needed for different stakeholders

User requirements: high-level, targeting end users and buyers, pre-contract

System requirements: much more detailed, targeting developers, but still reviewed by users, post-contract

Requirements Validation

Checks include: consistency checks, completeness checks, realism checks (can requirements be met using time and money budgets?), verifiability (is it possible to test that each requirement is met?; non-functional requirements must be measurable, i.e. performance must be good)

User Stories

Used in Agile (lightweight) development processes, e.g. Extreme Programming (XP), to document requirements

User stories are brief, written by the customer on an index card.

e.g. *10. User A leaves the office for a short time (vacation, etc.) and assigns his access privileges to user B, so B can take care of A's tasks while A is gone.*

Advantages

- Can really be owned by the customer: more likely to be correct
- Quick to write and change
- Easy to arrange and organise on a table or wall (e.g. could group by priority or risk)

Cons

- May be incomplete, inconsistent
- Only work in conjunction with good access to the customer
- Not suitable to form the basis of a contract

Lecture 2 - Use Cases

Use cases

- An important part of any requirements document for a system is a description of the system's behaviour from the viewpoint of its users.
- Behaviour can be broken down into units, each triggered by some stakeholder.
- **Use cases** are one way of describing these units.

What is a use case?

A **task** involving the system which **has value** for one or more stakeholders.

Actors are stakeholders who take an active part in the task.

Each use case:

- Has a discrete goal **the primary actor** wishes to achieve.
- Includes a description of the sequence of messages exchanged between the system and actors, and actions performed by the system, in order to achieve this goal.

The **primary actor** usually is the one triggering the use case.

Supporting actors may also be involved.

Some stakeholders may be neither primary nor supporting actors.

Actors in use cases

An **actor** can be:

- A human user of the system *in a particular role*
- An external system, which *in some role* interacts with the system

Not a user: a particular *kind* of user. E.g. Bank Customer.

The same human user or external system may interact with the system in more than one role: he/she/it will be (partly) represented by more than one actor (e.g., a bank teller may happen also to be a customer of the bank).

Paths in a use case

Usually a use case describes a core of sequence of steps necessary to achieve its goal.

However might be alternatives, including some handling when all does not go to plan and the goal is not achieved.

Each path through use case called a *use-case instance* or *scenario.*

- One talks about the **main success scenario** and alternate success or **failure scenarios**.

All scenarios in a use-case tied together by common user goal.

Warning: sometimes *scenario* and *use-case* are synonyms.

Example of use-case paths

Goal: Buy a product

Main Success Scenario

1. Customer browses catalogue and selects item that he/she wishes to buy
2. Customer goes to check out
3. Customer fills in shipping information
4. System presents full pricing information
5. Customer fills in credit card information
6. System authorises purchase
7. System confirms sale immediately
8. System sends confirmation email to customer

Extensions: variations on main success scenario

3a : Customer is regular customer

.1 : System displays current shipping and billing information

.2 : Customer may accept or override these defaults, returns to MSS at step 4, but skips step 5

6a : System fails to authorise credit card purchase

.1 : Customer may re-enter credit card information or may cancel

Phrase at the start of an extension is an enabling condition for that extension

Use Case Template (Example Fields)

- **Goal:** what the primary actor wishes to achieve.
- **Summary:** a one or two sentence description of the use case.
- **Stakeholders** and each's **Interest** in the use case.
- **Primary actor**
- **Supporting actors**
- **Trigger**: the event that leads to this use case being performed.
- **Pre-conditions/Assumptions:** what can be assumed to be true when the use case starts.
- **Guarantees:** what the use case ensures at its end
  - Success guarantees
  - Failure guarantees
  - Minimal guarantees
- **Main Success Scenario**
- **Alternative Scenario**

Use cases: connections and scope

A use case:

- Can have different levels of detail
  - e.g. depending on where it is used in development process
- May refer to other use cases
  - To provide further information on particular steps
- May describe different scopes
  - e.g. a system of systems, a single system or a single component of a system

Requirements capture organised by use cases

Use cases can help with requirements capture by providing a structured way to go about it:

1. Identify the actors
2. For each actor, find out
   - What they need from the system
   - Any other interactions they expect to have with the system
   - Which use cases have what priority for them

Good for both requirements specification and iterated requirements elicitation.

Use cases primarily capture functional requirements, but sometimes non-functional requirements are attached to a use case.

Other times, non-functional requirements apply to subsets or all of use-cases.

Uses of use cases in software processes

- Driving design
- Design validation
  - You can walk through how a design realises a use case, checking that the set of classes provides the needed functionality and that the interactions are as expected.
- Testing
  - Use cases can be a good source of system tests.

Possible problems with use cases

- Interactions spelled out may be too detailed, may needlessly constrain design
- May specify supporting actors that are not essential for fulfilling goal of primary actor
  - Does borrowing a book have to involve a librarian?
- Focus on operational nature of system may result in less attention to software architecture and static object structure
- May miss requirements not naturally associated with actors

The Unified Modeling Language

UML is a graphical language for recording aspects of the requirements and design of software systems.

It provides many diagram types; all the diagrams of a system together form a UML model, which must be consistent (in a weak sense).

Mostly tailored to an OO world-view.

Often used just for documentation, but in **model-driven development**, a UML model may be used e.g. to generate and update code and database schemas automatically.

Many tools available to support UML.

Lecture 3 - Software Design and Modelling

What is design?

Design is the **process** of deciding **how** software will **meet requirements.**

Usually excludes detailed coding level.

Outputs of design process

- Models
  - e.g. using UML or Simulink
  - Often graphical
  - Can be executable
- Written documents
  - Important that these record reasons for decisions

(Some) criteria for a good design

- It can meet the known (functional and non-functional) requirements
- It is maintainable (i.e. can be adapted to meet future requirements
- It is straightforward to explain to implementors
- It makes appropriate use of existing technology (e.g. reusable components)

Notice how the human angle in most of these points, and the situation-dependency, e.g.

- Whether an OO design or a functional design is best depends (partly) on whether it is to be implemented by OO programmers or functional programmers
- Different design choices will make different future changes easy - a good design makes the most likely ones easiest

Levels of design

Design occurs at different levels, e.g. someone must decide:

- How is your system split up into subsystems? (high-level, or architectural, design)
- What are the classes in each subsystem? (low-level, or detailed, design)

At each level, decisions needed on

- What are the responsibilities of each component?
- What are the interfaces?
- What messages are exchanged, in what order?

Examples of architectures

- Client-server
- Peer to peer
- Message bus

What is architecture?

The way that components work together.

More precisely, an architectural decision is a decision which affects how components work together.

Pervasive, hence hard to change. Indeed an alternative definition is 'what stays the same' as the system develops, and between related systems.

Classic structural view

Architecture specifies answers to:

- What are the components?
  - Where shall we put the encapsulation barriers? Which decisions do we want to hide inside components, so that we can change them without affecting the rest of the system?
- What are the connectors?
  - How and what do the components really need to communicate? What should be in the interfaces? What protocols should be used?

The component and connector view of architecture is due to Mary Shaw and David Garland - spawned specialist architectural description languages, and influenced UML 2.0.

More examples of architectural decisions

- What language and/or component standard are we using? (C++, Java, CORBA, DCOM, JavaBeans...)
- Is there an appropriate software framework that can be used?
- What conventions do components have about error handling?

Clean architecture helps get reuse of components.

Detailed design

Happens inside a subsystem or component.

Examples:

- System architecture has been settled by a small team written down, and reviewed.
- You are in charge of the detailed design of one subsystem.
- You know what external interfaces you have to work to and what you have to provide.
- Your job is to choose classes and their behaviour that will do that.

Idea: even if you're part of a huge project, your task is now no more difficult than if you were designing a small system.

But: your interfaces are artificial, and this may make them harder to understand/negotiate/adhere to.

Software design principles

Key notions that provide the basis for many different software design approaches and concepts.

Design principles: initial example

Which of these two designs is better?

A.

```
public class AddressBook {
       private LinkedList<Address> theAddresses;
       public void add (Address a) {theAddresses.add(a);}
       // . . . etc. . . .
}
```

B.

```
public class AddressBook extends LinkedList<Address>{
       // no need to write an add method, we inherit it
}
```

A is preferred.

- An `AddressBook` is not conceptually a `LinkedList` so it shouldn't extend it.
- If B chosen, it is much harder to change implementation, e.g. to a more efficient `HashMap` keyed on name.

Design Principles

Cohesion is a measure of the strength of the relationship between pieces of functionality within a component. **High cohesion** is desirable.

Benefits of high cohesion include increased understandability, maintainability and reliability.

Coupling is a measure of the strength of the inter-connections between components. **Low** or **loose** coupling is desirable. Benefits of loose coupling include increased understandability and maintainability.

Abstraction - procedural/functional, data is the creation of a view of some entity that focuses on the information relevant to a particular purpose and ignores the remainder of the information.

e.g. the creation of a sorting procedure or a class for points.

Encapsulation / information hiding is grouping and packaging the elements and internal details of an abstraction and making those details inaccessible.

Separation of interface and implementation is specifying a public interface, know to the clients, separate from the details of how the component is realised.

Decomposition, modularisation is dividing a large system into smaller components with distinct responsibilities and well-defined interfaces.

Modeling

Model - any precise representation of some of the information needed to solve a problem using a computer.

e.g. a model in UML, the Unified Modeling Language

A UML model

- Is represented by a set of diagrams
- But has a structured representation too (stored as XML)
- Must obey the rules of the UML standard
- Has a (fairly) precise meaning
- Can be used informally, e.g. for talking round a whiteboard
- And, increasingly, for generating and synchronising with, code, textual documentation, etc.

Pros and cons of Big Design Up Front (BDUF)

- Often unavoidable in practice
- If done right, simplifies development and saves rework
- But error prone AND wasteful

Alternative (often) is simple design plus refactoring.

XP maxims:

- You ain't gonna need it
- Do the simplest thing that could possibly work!

Lecture 4 - UML Class Diagrams

The Unified Modeling Language

UML is a graphical language for recording aspects of the requirements and design of software systems.

It provides many diagram types; all the diagrams of a system together form a UML model.

Three important types of diagram:

1. Use-case diagram
2. Class diagram
3. Sequence diagram

A class

A class as design entity is an example of a model element: the rectangle and text form an example of a corresponding **presentation element.**

UML explicitly separates concerns of actual symbols used vs meaning.

Allows same class to appear in multiple diagrams, maybe in different formats.

Many other things can be model elements: use cases, actors, associations, generalisation, packages, methods...

Visibility

Can show whether an attribute or operation is

- Public (visible from everywhere) with +
- Private (visible only from inside objects of this class) with -

Or protected (#), package (~) or other language dependent visibility.

Association between classes

An object is an instance of a class.

A link is an instance of an association.

Each link consists of a pair of objects, each an instance of the class at each end of the association.

e.g. `< Copy 3 of War and Peace, War and Peace >`

Classes can be thought of as sets of their objects, and associations as binary relations or sets of links.

Rolenames on associations

Rolenames show the roles that objects play in an association.

e.g. Personal Tutor (PT) —— Student (tutees)

The above association shows that

- Students are tutees of a Personal Tutor
- A Personal Tutor is a PT for some students

Can use visibility notation + - etc on role names too.

Multiplicities of associations

e.g. Personal Tutor (PT, 1) —— Student (tutees, 1..30)

Student (0..*) —— Course (1..*)

For each Personal Tutor there are between 1 and 30 students

For each student there is exactly one Personal Tutor

* for unknown number: each student takes one or more courses.

* 0..* often abbreviated as *

Navigability

Adding an arrow at the end of an association shows that some object of the class at one end can access some object of the class at the other end, e.g. to send a message.

e.g. Student (is taking) <——— Course

Crucial to understanding the coupling of the system.

Direction of navigability has nothing to do with direction in which you read the association name.

Use x near an association end to show non-navigability.

Attributes vs Associations

Attributes and associations show similar information.

Personal Tutor (PT, 1) —— Student (tutees, 1..30)

Could be shown instead as

Personal Tutor { tutees : Student[1..30] }

Student { PT: Personal Tutor }

Use of attribute implies navigability.

Attribute preferred if attribute type is simple, e.g. if it represents a simple value such as a Boolean or a date.

Coding Associations

Personal Tutor (PT, 1) —— Student (tutees, 1..30) could be coded in Java as:

```
Public class PersonalTutor {
       Set<Student> tutees;
       . . .
}
Public class Student {
       PersonalTutor PT;
       . . .
}
```

If we want navigability both ways.

Use `List<Student>` rather than `Set<Student>` if want `Student` to be ordered.

Generalisation

LibraryMember <—— MemberOfStaff

Usually corresponds to implementation with inheritance.

Usually can read *is a:* e.g. Member of Staff *is a* Library Member.

Interfaces

In UML an interface is just a collection of operations, that can be *realised* by a class.

Identifying objects and classes

Simplest and best: look for noun phrases in the system description!

Then abandon things which are:¶

- Redundant
- Outside scope
- Vague
- Attributes
- Operations and Events
- Implementation classes¶

Avoid incorporating premature design decisions into your conceptual level model.

Similarly, can use verb phrases to identify operations and/or associations.

Identifying classes example

Books and Journals: The library contains books and journals. It may have several copies of a given book. Some of the books are for short tern loans only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.

- *Eliminate:* library, short term loan, member of the library, week, time
- *Left with:* item, book, journal, copy (of book), library member, member of staff

Lecture 5 - Software Component Interactions and Sequence Diagrams

Dynamic aspects of design

Suppose that we have decided what classes should be in our system, provisionally. What next? Well, we have to meet the requirements.

In the end, we need to know what operations they have, and what each operation should do.

Two ways of looking at this:

1. Inter-object behaviour: who sends which messages to whom?
2. Intra-object behaviour: what state changes does each object undergo as it receives messages, and how do they affect its behaviour? (UML provides state diagrams, enhanced FSMs)

Thinking about inter-object behaviour

There's no algorithm for constructing a good design. Create one that's good according to the design principles.

1. Your classes should, as far as possible, correspond to domain concepts.
2. The data encapsulated in the classes is usually pretty easy to define using the real world as a model.
3. Then look at the scenarios in the use cases, and work out where to put what operations to get them done.

Can get hard when several objects have to collaborate and it isn't clear which should take overall responsibility.

CRC cards can help.

Interaction diagrams

Describe the *dynamic* interactions between objects in the system, i.e. the pattern of message-passing.

Good for showing how the system realises [part of] a use case.

Particularly useful where the flow of control is complicated, since this can't be deduced from the class model, which is static.

UML has two sorts, *sequence* and *communication* diagrams - we'll mainly talk about sequence diagrams.

OO message-passing terminology

```
class A {
       f() {
              B b = . . . ;
              . . .
              b.g();
              . . .
       }
}

class B {
       g()}
```

Let `a` be some object of type `A`.

- Object `a` sends a (call) message `g` to object `b`
  - An invocation `a.f()` makes a call `b.g()`
- Object `b` sends a reply message to object `a`
  - An invocation `b.g()` finishes and control flow returns to `a.f()`

Developing an interaction diagram

1. Decide exactly what behaviour to model.
2. Check that you know how the system provides the behaviour: are all the necessary classes and relationships in the class model?
3. Name the objects which are involved.
4. Identify the sequence of messages which the objects send to one another.
5. Record this in the syntax of an interaction diagram.

A use case

Scope: A system for keeping track of books owned a library and which books are checked out to whom.

Title: Borrow book

Primary Actor: A book borrower

Description: A book borrower presents a copy of the book to the system along with some ID card. Assuming the borrower has not already checked out some maximum number of books, the system permits the loan, recording who the book is checked out to, and noting that the number of free copies of the book is reduced.

[UML Class and Sequence Diagram]

What is a good interaction pattern?

In designing an interaction, your first aim is obviously to design *some* collection of operations that can work together to achieve the aim.

Next, consider:

- Conceptual coherence: does it make sense for this class to have that operation?
- Maintainability: which aspects might change, and hw hard will it be to change the interaction accordingly?
- Performance: is all the work being done necessary?

Reducing longer-range coupling

The *Law of Demeter* design principle recommends that in response to a message m, an object O should send messages *only* to the following objects:

1. O itself
2. Objects which are sent as arguments to the message m
3. Objects which O creates as a part of its reaction to m
4. Objects which are *directly* accessible from O, that is, using values of attributes of O.

Example: A -> B -> C (where A, B, and C are objects)

Consider an A operation needing to access part of data in C object.

- We have longer range coupling if operation requests reference to C object from B object.
- Better if A's operation calls operation in B designed just to pick out needed data from C.

More complex sequence diagrams

UML provides further notation for e.g.

- Conditional Behaviour
- Iterative Behaviour
- Concurrent Behaviour
- Including one diagram in another

Lecture 6 - Design Patterns

Design Patterns

"Reuse of good ideas."

A pattern is a named, well-understood good solution to a common problem.

- Experienced designers recognise variants on recurring problems and understand how to solve them.
- They communicate their understanding by recording it in design patterns.
- Such patterns then help novices avoid having to find solutions from first principles.

Patterns help novices learn by example to behave more like experts

Patterns: background and use

Idea comes from architecture (Christoper Alexander): e.g. **Window Place:** observe that people need comfortable places to sit, and like being near windows, so make a comfortable seating place at a window.

Similarly, software design patterns address many commonly arising technical problems in software design, particularly OO design.

Patterns also used in: reengineering; project management; configuration management; etc.

Pattern catalogues: for easy reference, and to let designers talk shorthand.

A very simple recurring problem

We often want to be able to model tree-like structures of objects, where an object may be

- A thing without interesting structure, a leaf of the tree, or
- Itself composed of other objects
  - Which in turn might be leaves or might be composed of other objects

Commonly there are operations on these tree structures that make sense, whether the trees are single objects or are composed of multiple objects.

Code interacting with these trees might want to invoke these operations without knowing whether the tree is just a leaf or has some composite structure.

Composite is a *design pattern* which describes a well-understood way of doing this.

Example Situation

A graphics library provides primitive graphics elements like lines, text strings, circles, etc.

A user of the library wants support for operations on elements that are uniform across different kinds.

e.g. Move, draw, change colour, get width

Makes sense to have

- Classes **Line, Text,** etc. for each element kind, and
- A **Graphics** interface or abstract base class that describes the common features of graphics elements.

Further, the user wants to group elements together to form pictures, which can then be treated as a whole.

- e.g. user expects to be able to move a composite picture just as they move primitive elements

And user wants to group pictures and elements together into larger pictures.

e.g. `Using List<Graphics>` for type of pictures is not enough.

Benefits of Composite Pattern

- Can automatically have trees of any depth: don't need to do anything special to let containers (Pictures) contain other containers
- Is easy to add new kinds of Graphics subclasses, including different kind of pictures, because user programs don't have to be altered

A drawback of Composite Pattern

- Code for each operation is spread around the Graphic's subclasses and may cause maintenance issues
- One solution is to code the operation using a single method that walks Graphics object trees and has explicit conditional statements branching on the particular subclasses each Graphics object belongs to.
  - Use `instanceof` operator in Java to test class membership

Another solution is to use the *Visitor* pattern.

- Each operation coded using a single class
- One method provided for the case of each Graphics subclass

Variations on Composite

- Might want to write some new method that walks over a whole `Graphics` tree. e.g. a *tree-map* method
- To support it, need methods in `Graphics` like `numChildren()` and `getChild(int i)`
- `Graphics` then provides default implementations of these methods for the leaf subclasses

Elements of a pattern

A pattern catalogue entry normally includes roughly:

- Name (e.g. Publisher-Subscriber)
- Aliases (e.g. Observer, Dependants)
- Context (in what circumstances can the problem arise?)
- Problem (why won't a naive approach work?)
- Solution (normally a mixture of text and models)
- Consequences (good and bad things about what happens if you use the pattern)

Cautions on pattern use

Patterns are very useful *if you have the problem they're trying to solve.*

But they add complexity, and often e.g. performance penalties too. Exercise discretion.

You'll find the criticism that the GoF (Gang of Four, authors of the first major Design Patterns book) patterns in particular are 'just' getting round the deficiencies of OOPLs. This is true, but misses the point.

Lecture 7 - Construction: High Quality Code for 'programming in the large'

What is high quality code?
High quality code does what it is supposed to and will not have to be thrown away when that changes.

What has this to do with programming in the large?
Why is the quality of code more important on a large project than on a small one?
Fundamentally because other people will have to **read** and **modify** your code, e.g.
- Because of staff movement
- For code reviews
- For debugging following testing
- For maintenance

Even you in a year's time count as 'other people'!

Bracketing
Consistent bracketing (in-line or after-line does not matter, as long as you settle on a convention and follow it throughout the project).

Indentation
Be consistent about indentation. Don't use TABs!

Naming
`customer.add(order);` is much better than `c.add(o);`
Use meaningful names!
A good length for most names is 8-20 characters.
Follow conventions for short names (e.g. `i`, `j`, `k` for loop indexes)

White space
Use white space consistently.
e.g. both `( x * x + y * y )` and `(x*x + y*y)` are fine but stick to your choice throughout the project.

Commenting
Use comments when they're useful.
Avoid comments when they are obvious.
Too many comments is actually a more common serious problem than too few.
Good code in a modern high-level language shouldn't need many *explanatory* comments, and they can cause problems.
"If the code and the comments disagree, both are probably wrong."

How to write good code
Follow your organisation's coding standards: placement of curly brackets, indenting, variable and method naming.
Use meaningful names: for variables methods and classes. If they become out of date, change them.
Avoid cryptic comments: try to make your code so clear that it doesn't need comments.

Good Coding

... is also about:

- Declaration and use of local variables
  - Good to try keeping local variable scope restricted
- How conditional and loop statements
  - With *if-else* statements, put normal frequent case first
  - Use *while*, *do-while, for* and *for/in* loops appropriately
  - Avoid deep nesting
- How code is split among methods
  - Good to try avoiding long methods, over 200 lines
- Defensive programming
  - Using assertions and handling errors appropriately
- Use of OO features and OO design practices (e.g. patterns)
- Use of packages
- Balance structural complexity against code duplication
  - Don't write the same two lines 5 times (why not?) when an easy refactoring would let you write them once, but equally, don't tie the code in knots to avoid writing something twice.
- Remove dead code, unneeded package includes, etc.
- Be clever, but not too clever
  - Remember the next person to modify the code may be less clever than you! Don't use deprecated, obscure or unstable language features unless absolutely necessary.

And much more.

Javadoc

Any software project requires documentation aimed at the *users* of its component parts (e.g. methods, classes, packages).

Documentation held separately from code tends not to get updated, so including such documentation in stylised comments is a good idea.

Javadoc is a tool originally from Sun. Programmer writes doc comments in particular form, and Javadoc produces pretty-printed hyperlinked documentation.

Example:

```
    /**
     * Returns an Image object that can then be painted on the screen.
     * The url argument must specify an absolute {@link URL}. The name
     * argument is a specifier that is relative to the url argument.
     * <p>
     * This method always returns immediately, whether or not the
     * image exists. When this applet attempts to draw the image on
     * the screen, the data will be loaded. The graphics primitives
     * that draw the image will incrementally paint on the screen.
     *
     * @param  url  an absolute URL giving the base location of the image
     * @param  name the location of the image, relative to the url argument
     * @return      the image at the specified URL
     * @see         Image
     */
     public Image getImage(URL url, String name) {
       try {
           return getImage(new URL(url, name));
       } catch (MalformedURLException e) {
           return null;
} }
```

A Common Pattern in Java

```
interface Foo {
...
}
class FooImpl implements Foo {
...
}
```

Why is this so much used?

Use of single implementations of interfaces

All about separating users from implementations.

- Expect that users of `FooImpl` objects always interact with them at type `Foo`.
  - Maybe they are not even aware of `FooImpl` name
  - Alternative means of constructing `FooImpl` objects to new `FooImpl()` can be set up
- Can use access control modifiers (e.g. *private, public*) on `FooImpl` members to define its interface
  - However interface and implementation still mixed together in one class definition
  - This was an early criticism of the OO take on the ADT paradigm
- With distinct interface `Foo`, users see nothing of `FooImpl`

The relevance of object orientation

Construction is intimately connected to design: it is design considerations that motivate using an OO language.

Key need: control complexity and abstract away from detail. This is essential for systems which are large (in any sense).

- **Classes:** grouping of behaviour, conceptual abstraction, hiding of implementation from users
- **Inheritance:** incremental extension without having to know about details
- **Interfaces:** decouple users from implementations

Packages

Recall that Java has packages. Why, and how do you decide what to put in which package?

Packages:

- Allow related pieces of code to be grouped together.
- Are units of **encapsulation**. By default, fields and methods are visible to code in the same package.
- Give a way to organise the **namespace**.
  - They avoid problems if different software components of a large system happen to use the same class names.

Caution: the package 'hierarchy' is not really a hierarchy as far as access restrictions are concerned - the relationship between a package and its sub/superpackages is just like any other package-package relationship.

Lecture 8 - Construction: Version Control and System Building

The problem of systems changing

- Systems are constantly changing through development and use
  - Requirements change and systems evolve to match
  - Bugs found and fixed
  - New hardware and software environments are targeted
- Multiple versions might have to be maintained at each point in time
- Easy to lose track of which changes realised in which version
- Help is needed in managing versions and the processes that produce them

Solution: Software Configuration Management

Common configuration management activities:

- **Version control**
  - Tracking multiple versions
  - Ensuring changes by multiple developers don't interfere
- **System building**
  - Assembling program components, data and libraries
  - Compiling and linking to create executables
- **Change management**
  - Tracking change requests
  - Estimating change difficulty and priority
  - Scheduling changes
- **Release management**
  - Preparing software for external release
  - Tracking released versions

Version Control

The core of configuration management

The idea:

- Keep copies of every version (every change made) of files
- Provide change logs
- Somehow manage situation where several people want to edit the same file
- Provide diffs/deltas between versions

Lock-Modify-Unlock Model

- Editor checks-out a file from a repository
  - Editor locks file
  - Others can check-out, but only for reading
- Editor makes changes
- Editor checks-in modified file to repository
  - Lock is released
  - Changes now viewable by others
  - Others now can make their own changes

RCS

- Old, primitive VC system, much used on Unix that uses Lock-Modify-Unlock model
- Keeps deltas between versions; can restore, compare, etc.
- Can manage multiple branches of development
- Works on single files, not collection of files or directory hierarchies
- Best suited for small projects, where only one person edits at a time

CVS and SVN

CVS is much richer system, (originally) based on RCS.

Subversion (SVN) is very similar, but newer.

Both handle entire directory hierarchies or projects - keep a single master repository for project.

Designed for use by multiple developers working simultaneously.

Copy-Modify-Merge model replaces *Lock-Modify-Unlock*.

Pattern of use for **Copy-Modify-Merge**:

1. Checks out entire project (or subdirectory) and NOT individual files.
2. Edit files.
3. Do *update* to get latest versions of everything from repository.
   - System merges non-overlapping changes
   - User has to resolve overlapping changes - conflicts
4. Check-in version with merges and resolved conflicts

Central repository may be on local filesystem, or remote.

Three-way Merge

A 3-way merge considers how files F2 and F3 are each modifications of some nearest common ancestor file F1. It creates a new file F4 that incorporates into F1 both the changes from F1 to F2 and from F1 to F3. When changes are conflicting, i.e. to the same lines of F1, then both changes are inserted into F4 with extra text making clear the conflicting changes and prompting the developer to resolve them.

A standard usage scenario is when a checked-out version of a file, possibly locally modified, has to be brought into sync with a version of that file that has been modified in a different way, perhaps that is on a different development branch and/or that has been modified by another developer.

Example of a three-way merge :¶

1. Original file:

```
Alpha
Bravo
Charlie
```

2. Tester 1 Edits

```
Alpha
Foxtrot
Charlie
```

3. Tester 2 Edits

```
Delta
Alpha
Echo
Charlie
```

4. Tester 2 commits changes
5. Tester 1 commit fails
6. Tester 1 updates and merge reports conflicts

```
Delta
Alpha
<<<<<<< .mine
Foxtrot
=======
Echo
>>>>>>> .r4
Charlie
```

7. Tester 1 fixes conflicts
8. Tester 1 commits

Distributed version control

e.g. Git, Mercurial, Bazaar

VCS use a single central repository -> you cannot check changes unless you can connect to its host and have permission to check in.

dVCS allow many repositories of the same software to be managed, merged, etc.

Advantages:

- Reduces dependence on single physical node
- Allows people to work (including check in, with log comments etc.) while disconnected
- Much faster VC operations

Disadvantages:

- Much more complicated and harder to understand

Branches

Simplest use of a VCS gives you a single linear sequence of versions of the software.

Sometimes it's essential to have, and modify, two versions of the same item and hence the software: e.g., both

- The version customers are using, which needs bug fixes
- A new version which is under development

Branching supports this: you end up with a tree of versions.

Merging branches (e.g. to roll bug fixes in with new development?)

- Need a 3-way merge between ends of two branches and a common ancestor
- Merge support good in Git and Mercurial
  - Developers with these VCSs use branches a lot more
- Merge support has improved with recent SVN versions

Build tools

To make sure that you recompile one piece of code when another than it depends on changes.

Makefile for a C program

```
OBJS = ppmtomd.o mddata.o photocolcor.o vphotocolcor.o dyesubcolcor.o
ppmtomd: $(OBJS)
	$(CC) -o ppmtomd $(OBJS) $(LDLIBS) -lpnm -lppm -lpgm -lpgm -lpbm -lm
ppmtomd.o: ppmtomd.c mddata.h
	$(CC) $(CDEBUGFLAGS) -W -c ppmtomd.c
mddata.o: mddata.c mddata.h
```

Makefile rule structure

target: dependencies

command1

command2

Running `make` *target* uses *commands* to

- Create *target* from *dependencies* if it does not exist
- Rebuild *target* when any of *dependencies* are newer

Before creating/rebuilidng *target*, `make` recursively considers whether any *dependencies* need creating or rebuilding.

Ant

`make` can be used for Java

However, there is a pure Java build tool called Ant.

Ant *Buildfiles* (typically `build.xml`) are XML files, specifying the same kind of information as `make`.

Part of an Ant buildfile for a Java Program

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name ="Dizzy" default = "run" basedir=".">
<description>
    This is an Ant build script for the Dizzy chemical simulator. [...]
</description>
<!-- Main directories -->
<property name = "source"      location = "${basedir}/src"/> [...]
<!--General classpath for compilation and execution-->
<path id="base.classpath">
  <pathelement location = "${lib}/SBWCore.jar"/> [...]
</path> [...]
<target name = "run" description = "runs Dizzy"
                     depends =" compile, jar">
  <java classname="org.systemsbiology.chem.app.MainApp" fork="true">
    <classpath refid="run.classpath" />
    <arg value="." />
  </java>
</target> [...]
</project>
```

Maven

For making the building of Java projects simpler and more uniform.

Defines default support for:

- Compiling
- Testing (using unit tests)
- Packaging (e.g. as a jar file)
- Integration testing
- Installing (e.g. into local repository)
- Deploying (e.g. into release environment for sharing with other project)
- Generating documentation

Per-platform code configuration

Different operating systems and different computers require code to be written differently (incompatible APIS, etc.). Writing portable code in C (etc.) is hard.

Tools such as GNU Autoconf provide ways to automatically extract information about a system. The developer writes a (possibly complex) configuration file; the user just runs a shell script produced by autoconf.

Canonical way to install Unix software from source:

`./configure; make; make install`

A newer tool is CMake.

Problem is less severe with Java, but still tricky to write code working with all Java dialects.¶

Lecture 9 - Refactoring

The Problem

As code evolves its quality naturally decays

- Initially code implementing a good design
- Changes often local, without full understanding of the context
- With loss of structure, code becomes harder to follow, harder to modify

Refactoring is about restoring good design in a disciplined way

- Expertise on refactoring captured in *refactoring* patterns
- Enables rapid learning
- Enables tool support

Refactoring Definition

Refactoring *(noun)* is a change made to the internal structure of software software to make it easier to understand and cheaper to modify *without changing its observable behaviour.*

Refactor *(verb)* to restructure software by applying a series of refactorings *without changing its observable behaviour.*

Why refactor?

- Helps you program faster in general
  - Makes software easier to understand
    - Your code, by you/others
    - Others' code, by you
  - Helps you make subsequent modifications quicker
  - Helps you find bugs
    - Design becomes clearer and bugs easier to see

When to refactor?

It was once seen as a kind of maintenance to be done when:

- You've inherited legacy code that's a mess
- A new feature is required that necessitates a change in the architecture

But can also be an integral part of the development process.

Agile methodologies (e.g. XP) advocate continual refactoring.

XP maxim: *"Refactor mercilessly!"*

What does refactoring do?

A refactoring is a *small* transformation which preserves correctness.

Examples:

- Add parameter
- Change Bidirectional Association to Unidirectional
- Extract Variable (Introduce Explaining Variable)
- Replace Conditional with Polymorphism

Extract Variable

Change

```
if ((platform.toUpperCase().indexOf(“MAC”) > -1) && (browser.toUpperCase().indexOf(“IE”) > -1) &&
wasInitialized() && resize > 0 ) {
        // do something
}
```

To

```
final boolean isMacOS = platform.toUpperCase().indexOf(“MAC”) > -1;
final boolean isIEBrowser = platform.toUpperCase().indexOf(“IE”) > -1;
final boolean wasResized = resize > 0;
if (isMacOS && isIEBrowser && wasInitialized() && wasResized) {
        // do something
}
```

Replace Conditional with Polymorphism (inheritance)

Change

```
double getSpeed() {
        switch (_type) {
                case EUROPEAN:
                        return getBaseSpeed();
                case AFRICAN:
                        return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
                case NORWEGIAN_BLUE:
                        return (_isNailed) ? 0 : getBaseSpeed(_voltage);
        }
        throw new RuntimeException (“Should be unreachable”);
}
```

To

```
class Bird {
        getSpeed();
}
class European extends Bird {
        getSpeed();
}
class African extends Bird {
        getSpeed();
}
class NorwegianBlue extends Bird {
        getSpeed();
}
```

Eclipse Refactoring

Eclipse has a built-in refactoring tool.

Can be grouped in three broad classes.

1. Renaming and physical reorganisation

A variety of simple changes.

For example:

- Rename Java elements (classes, fields, methods, local variables)
  - On class rename, import directives updated
  - On field rename, getter and setter methods also renamed
- Move classes between packages
- Eclipse applies these changes semantically (much better than syntactic search-and-replace)

2. **Modifying class relationships**

   Heavier weight changes. Less used, but seriously useful when they are used. e.g.,
   - Move methods or fields up and down a class inheritance hierarchy
   - Extract an interface from a class
   - Turn an anonymous class into a nested class

3. **Intra-class refactorings**

   Rearranging code within a class to improve readability, etc. e.g.,
   - Extract Method: pull code block into new method
     - Good for shortening method or making block reusable
     - Also can extract local variables and constants
   - Change the type of a method parameter or return value

Safe refactoring

How do you know refactoring hasn't changed/broken something?

Perhaps somebody has proved that a refactoring operation is safe.

More realistically: *test, refactor, test*

This works better the more tests you have: ideally, unit tests for every class

Bad smells in code

- Duplicated code
- Long method
- Large class
- Long parameter list
- Lazy class
- Long message chains

Lecture 10 - Verification, Validation and Testing

Verification, Validation and Testing

All techniques for improving product quality, e.g. by eliminating bugs (including design bugs)

Verification: are we building the software right? Does software meet requirements? (HOW)

Validation: are we building the right software? Does software meet customer's expectations? (WHAT)

Testing is a useful (but not the only) technique for both.

Other techniques useful for verification: static analysis, reviews/inspections/walkthroughs

Other techniques useful for validation: prototyping/early release

Essence of testing

- Generating stimulus for component
- Collecting outputs from component
- Checking if actual outputs are as expected

Often hard to fully test a component in isolation (component test env. constructed using *mock objects)*

Test automation and Junit

Automation of test is essential, particularly when tests must be re-run frequently.

JUnit is a framework for automated testing of Java programs.

Assertions

Junit provides a library of *assert statements* to use in test code for checking output of the program being tested. Checks can also be spread throughout the program code itself.

Example:

Suppose `i` is an integer variable, and we are writing a bit of code where we 'know' that `i` must be even (because of a function that is declared earlier on in the code). We can add a statement:

```
assert i % 2 == 0
```

to check this - if `i` is odd, an `AssertionError` exception is raised.

Such assertion checking can (and should, for release) be switched off.

Preconditions, Postconditions, Invariants

Common types of assertions:

- **Method Precondition:** a condition that must be true when a method is invoked.
- **Method Postcondition:** a condition that the method guarantees to be true when it finishes.
- **Class Invariant:** a condition that should always be true of objects of the given class. What does always mean? In all *client-visible* states: that is, whenever the object is not executing one of its methods.

Java Modeling Language

JML provides:

- A richer language for writing conditions than Java boolean expressions.
  e.g. qunatifiers `\forall` and `\exists`
- Special comment syntax for common assertion types.
  - Preconditions: `//@ requires x > 0;`
  - Postconditions: `//@ ensures \result % 2 == 0;`
  - Invariants: `//@ invariant name.length <= 8;`
  - General assertions: `//@ assert i + j = 12;`

Example:

```
//@ requires x >= 0.0
/*@ ensures JMLDouble
  @           .approximatelyEqualTo
  @           (x, \result * \result, eps);
  @*/
public static double sqrt(double x) {
       /*. . .*/
}
```

Assertion-related tool support

The tools jmlc, jmlrac etc. compile and run JML-annotated Java code into bytecode with specific runtime assertion checking.

Can also use assertions on inputs to constrain random generation of input data.

- *QuickCheck*: Originally Haskell, now Java and other languages

"Bug"

Or more precisely,

1. Mistake: a human action that produces a fault
2. Fault: an incorrect step, process, or data definition in a computer program, a.k.a. defect
3. Error: a difference between some computed value and the correct value
4. Failure: the software or whole system failing to deliver some service it is expected to deliver

Faults do not necessarily lead to errors, errors do not necessarily lead to failures.

Kinds of Tests

Testing approaches:

- **black box** - purely specification based
  - How it happens does not matter, but what happens matter
  - Don't see what's inside
  - Just input/output
- **white box** - also considers software structure
  - Consider internal structure

Kinds of tests:

- **Module (or unit) tests**: for each class in OO software
- **Integration tests**: test components interact properly
- **System tests**: check if functional and non-functional requirements met
- **Acceptance tests**: in customer environment (validation)
- **Stress tests**: look for graceful degradation, not catastrophe
- **Performance tests**: performance is often a non-functional requirement (e.g. in real-time systems)
- **Regression tests**: more like a testing methodology, repeated full tests after each modification

How to test

Desirable that tests are:

- Repeatable
- Documented (both the tests and the results)
- Precise
- Done on configuration controlled software

Ideally, test spec. should be written at the same time as the requirements spec.

- Tests and requirement features can be cross-referenced
- Use cases can suggest tests

Helps to **ensure testability** of requirements!

Test-First Development (TFD)

The motivating observation: tests implicitly define interface and specification of behaviour for the functionality being developed.

Basic idea:

- Write tests **before** writing the code they apply to
- Run tests as code is written

As a consequence:

- Bugs found at earliest possible point
- Bug location is relatively easy

Further advantages:

- **Clarifies requirements**: trying to write a test often reveals that you don't completely understand exactly what the code *should* do.
  - Discover issues more quickly than if coding first
  - Makes coding easier
- **Avoids poor ambiguity resolution**: if coding first, ambiguities might be resolved based on what's easiest to code. This can lead to user-hostile software.
- **Ensures adequate time for test writing**: if coding first, testing time might be squeezed or eliminated. That way lies madness.

Test-Driven Development (TDD)

A subtly different term, covers the way that in Extreme Programming detailed tests *replace* a written specification.

Evolving tests when a new bug is identified

Assume an implementation passes all current tests.

What if a new bug is identified by customer or by code review?

A good discipline is:

1. Fix or create a test to catch the bug.
2. Check that the test fails.
3. Fix the bug.
4. Run the test that should catch this bug: check it passes.
5. Rerun *all* the tests, in case your fix 'broke' something else.

Limitations of Testing

- Writing tests is time-consuming
- Coverage almost always limited: may happen not to exercise a bug
- Difficult/impossible to emulate live environment perfectly
  - e.g. *race conditions* that appear under real load conditions can be hard to find by testing
- Can only test executable things, mainly code, or certain kinds of model - not high level design or requirements.

Reviews/walkthroughs/inspections

One complementary approach is to get a group of people to look for problems.

This can:

- Find bugs that are hard to find by testing
- Discover when requirements have been misunderstood
- Spot unmaintainable code
- Work on non-executable things
  - e.g. requirements specification, UML model, test plan, user docs.

Of course the author(s) of each artefact should be looking for such problems - but it can help to have outside views too.

For our purposes, reviews/walkthroughs/inspections are all the same; there are different versions with different rules.

Reviews (Key Points)

- A review is a meeting of a few people
- Which reviews one specific artefact (e.g. design document, or defined body of code)
- For which specific entry criteria have been passed (e.g. the code compiles)
- Participants study the artefact before the meeting
- Someone, usually the main author of the artefact, presents it and answers questions
- The meeting does not try to fix problems, just identify them
- The meeting has a fixed time limit

Static Analysis

Inspecting code to determine properties of it *without* running it.

Testing is *dynamic analysis* (requires you to run the code).

Type-checking during compilation is a basic kind of static analysis.

Tools vary in what problems they address, e.g.

- Runtime exception issues (e.g. null pointer exceptions, array index out of bounds)
- Correctness of pre/post-condition specification of methods
- Concurrency of bugs (e.g. race conditions)

Trade-offs

When more complicated properties checked, tools generally

- Can analyse only smaller programs
- Are less automated (e.g. annotations required)

As tools are more automated and designed to work on larger programs, they often cannot

- Guarantee every problem flagged is a real error (false negatives)
- Find every error (false positives)

Need for annotations

Some tools require annotations, others do not.

- Sometimes annotations can be inferred automatically
  - e.g. checks for runtime exception conditions
- And sometimes program structure provides sufficient information for checking these annotations

Example:

```
int[] arr = {2, 3, 5, 7, 11};
assert arr != null;
for (int i = 0; i < arr.length; i++) {
       assert 0 <= i && i < arr.length;
       sum += arr[i];
}
```

There is a huge benefit to not requiring annotations but the possible checks without annotations are limited.

Static analysis tools for Java

OpenJML project provides support for both dynamic and static analysis of JML-annotated programs.

SpotBugs (FindBugs) is relatively widely used: looks for *bug patterns*, code idioms that are often errors.

ThreadSafe from the Informatics spinout *Contemplate* focuses on finding concurrency bugs.

Infer from FaceBook applies lightweight static analysis techniques that scale to 10^6 + LOC. Finds e.g. concurrency and null pointer exception issues.

Lecture 11 - Software Deployment and Maintenance

What is deployment?
Getting software out of the hands of the developers into the hands of the users.

Key issues around deployment
- **Business processes.** Most large software systems require the customer to change the way they work. Has this been properly thought through?
- **Training.** No point in deploying software if the customers can't use it.
- **Deployment itself.** How to physically get the software installed?
- **Equipment.** Is the customer's hardware up to the job?
- **Expertise.** Does the customer have the IT expertise to install the software?
- **Integration** with *other* systems of the customer.

Deployment itself
Many people will sell you tools to help deploy software. Such systems help you to:
- Package the software
- Make it available (nowadays over Internet of on DVD)
- Give the customer turn-key installers, which will:
  - Check he system for missing dependencies or drivers etc.
  - Install the software on the system
  - Set up any necessary licence managers

Maintenance
The process of changing a system after it has been delivered.
Kinds:
- Fixing bugs and vulnerabilities: not only in code, but also design and requirements
- Adapting to new platforms and software environments: e.g. new hardware, new OSes, new support software
- Supporting new features and requirements: necessary as operating environments change and in response to competitive pressures

Maintenance challenges
- Popularity of maintenance work
  - Unpopular - seen as less skilled, can involve obsolete languages
- Often a new team has to understand the software
- Development and maintenance often separate contracts
  - De-incentives developers paying attention to maintainability
- How software structure changes over time
  - Structure degrades, making maintenance harder
  - Not only code impacted, also other software aspects, e.g. user documentation
- Working with obsolete compilers, OSes, hardware

Software evolution and release management

Discipline in the evolution of software is (at least) as important as in its development.

- Gather change requirements: new features, adapting to system/business change, bug reports
- Evaluate each; produce proposed list of changes
- Go through normal development cycle to implement changes - ensuring that you understand the software, which may be non-trivial
- Issue new release

Unfortunately, emergencies happen, and things have to be done with urgency. If at all possible, go through the normal process afterwards.

Re-engineering

The process of taking an old or unmaintainable system and transforming it until it's maintainable. This *may* be considerably less risky and much cheaper than re-implementing from scratch.

Re-engineering may involve:

- **Source code translation** e.g. from obsolete language, or assembly, to modern language
- **Reverse engineering** i.e. analysing the program, possibly in the absence of source code
- **Structure improvement**, especially modularisation, architectural refactoring
- **Data re-engineering**, reformatting and cleaning up data
- **Adding adaptor interfaces** to users and newer other software

Issues:

- What is the specification?
- Which bugs do you deliberately preserve?

Lecture 12 - Software Development Processes: from the waterfall to the Unified Process

What is a process?

A set of activities and a way of organising/coordinating those activities in order to produce a software system.

Activities:

- Software specification (goals, functionality and constraints)
- Software development (producing software to meet specifications, design, construction)
- Software validation (does software do what the customer wants?)
- Software evolution (adapting software to changing customer needs)

Process ingredients

Processes are about:

- Ordering activities
- Outcomes of activities
- Arrangement of people and resources to carry out activities
- Planning in advance of execution, predicting time/cost/resources
- Monitoring
- Risk reduction
- Enabling their own adaptation

Processes are complex and creative

Process models are ideals; in practice, mix and match.

The waterfall model

Requirements -> Design -> Implementation -> Verification -> Maintenance

Advantages:

- Better than no process at all - makes clear that requirements must be analysed, software must be tested, etc.

Disadvantages:

- Inflexible and unrealistic - in practice, you cannot follow it: e.g. verification will show up problems with requirements capture
- Slow and expensive - in an attempt to avoid problems later, end up "gold plating" early phases, e.g. designing something elaborate enough to support the requirements you suspect you've missed, so that functionality for them can be added in coding without revisiting Requirements

Domains of use for waterfall-like models

Embedded systems: software must work with specific hardware: can't change software functionality later

Safety critical systems: safety and security analysis of whole system is needed up front before implementation

Some very large systems: allows for independent development of subsystems

Spiral Model

Successive loops involve more advanced activities:

- Checking feasibility
- Gathering requirements
- Design development
- Integration and test

Phases of single loop

1. **Objective setting:** plans drawn up, risks identified
2. **Risk assessment and reduction:** consider alternatives. e.g. go for prototype and analyse uncertain requirement
3. **Development and validation**
4. **Planning:** project reviewed and decisions made about continuing

A key innovation is prominent role of risk.

Advantages:

- Risk plays a prominent role, which is crucial
- Iterative approach is more suitable (than the waterfall) to the real world
- Steps are clearly identified

Disadvantages:

- Loops are still *sequential*, but in real-world it's more of a back-and-forth, and there is more concurrency (e.g. realising that a requirement has been misunderstood during development forces you to revise your requirements)
- Steps are not as elaborate as UP for instance; no guidelines on how to proceed with validation, with business modelling, and how the importance of each varies through time

Unified Process

One cycle consists of phases:

- **Inception** ends with commitment from the project sponsor to go ahead: business case for the project and its basic feasibility and scope known.
- **Elaboration** ends with
  - Basic architecture of the system in place
  - A plan for construction agreed
  - All significant risks identified
  - Major risks understood enough not to be too worried
- **Construction** (definitely iterative) ends with a beta-release system
- **Transaction** is the process of introducing the system to its users

Iteration

- Process for one product will have several cycles
- Each instance of a phase might have several iterations

Workflows: 9 Activities

6 Engineering workflows:

- Business modelling
- Requirements
- Analysis and design
- Implementation
- Test
- Deployment

3 Supporting workflows:

- Configuration and change management
- Project management
- Environment (e.g. process and tools)

UP Best Practices

Six fundamental best practices:

1. **Develop software iteratively.** Customer prioritisation, best first.
2. **Manage requirements.** Explicit documentation, analyse impact before adopting.
3. **Use component-based architectures.** Promote systematic reuse.
4. **Visually model software.** UML.
5. **Verify software quality.** Testing, checking coding standards, etc.
6. **Control change to software.** Configuration management.

Personal Software Process

- Identify those large-system software methods and practices that can be used by individuals
- Define the subset of those methods and practices that can be applied while developing small programs
- Structure those methods and practices so they can be gradually introduced
- Provide exercises suitable for practicing these methods in an educational setting

PSP provides a ladder of gradually more sophisticated practices.

Explicit phases of development, e.g. separate design from coding.

Lots of forms to fill in, e.g. time recording log, defect recording log.

Aim is to provide numerical data adequate for identifying weak areas and tracking improvements in process and in own skills.

Where does PSP fit in?

PSP is a relatively high ceremony process, aimed at individuals and small projects.

It is often used as a training process by people who expect to end up using a high ceremony process - such as UP - on large projects.

TSP, Team Software Process, is an intermediate.

Agile processes such as Extreme Programming take a very different approach - owing partly to deep philosophical differences, partly to different context assumptions.

A process maturity model such as CMMI (from SEI) can be used to help choose how to improve a software development process so as to fit the actual needs of the organisation.

Lecture 13 - Extreme Programming, an agile software development process

Agile processes

Able to react quickly to change

The Agile Manifesto:

> *We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
>
> ***Individuals and interactions*** *over processes and tools*
>
> ***Working software*** *over comprehensive documentation*
>
> ***Customer collaboration*** *over contract negotiation*
>
> ***Responding to change*** *over following a plan*
>
> *That is, while there is value in the items on the right, we value the items on the left more*

12 Principles of Agile

- Customer satisfaction by rapid delivery of useful software
- Welcome changing requirements, even late in development
- Working software is delivered frequently (weeks rather than months)
- Working software is the principal measure of progress
- Sustainable development, able to maintain a constant pace
- Close, daily co-operation between business people and developers
- Face-to-face conversation is the best form of communication (co-location)
- Projects are built around motivated individuals, who should be given the right support and trusted to get job done
- Continuous attention to technical excellence and good design
- Simplicity - the art of maximising the amount of work not done - is essential
- Best requirements and designs from self-organising teams
- Regular reflection on process and tuning of behaviour

Extreme Programming

> *"A humanistic discipline of software development, based on values of communication, simplicity, feedback, and courage."*

Example risks and the XP responses

- **Schedule slips:** short iterations give frequent feedback; features prioritised
- **Project cancelled after many slips:** customer chooses smallest release with biggest value
- **Release has so many defects that it is never used:** tests written with both unit-level and customer perspectives
- **System degrades after release:** frequent rerunning of tests maintains quality
- **Business misunderstood:** customer representative embedded in development team
- **System rich in unimportant features:** only highest priority tasks addressed
- **Staff turnover:** programmers estimate task times; new programmers nurtured

XP classification of software development activities

Coding: central. Includes understanding, communicating, learning

Testing: embodying requirements, assessing quality, driving coding

Listening: understanding the customer, communicating efficiently

Designing: creating structure, organising system logic

XP Practices¶

- The Planning Game
- Small releases
- Metaphor
- Simple design
- Testing
- Refactoring
- Pair programming
- Collective ownership
- Continuous integration
- 40-hour week
- On-site customer
- Coding standards

The Planning Game

Goal: maximise value and control development costs and risk

Pieces: story cards

Players:

- Customer understands scope, priority, business needs for releases: sorts cards by priority
- Developers estimate risk and effort: sorts cards by risk, split cards if more than 2-4 weeks

Phases: exploration, commitment, steer

On-site customer

A customer - someone capable of making the business's decisions in the planning game - sits with the development team always ready to:

- Clarify
- Write functional tests
- Make small-scale priority and scope decisions

Customer maybe does their normal work when not needed to interact with the development team.

Small releases

Release as frequently as is possible whilst still adding some business value in each release.

This ensures:

- That you get feedback as soon as possible
- Lets the customer have the most essential functionality ASAP

Every week or every month

Outside XP releases commonly every 6 months or longer

Metaphor

- About an easily-communicated overarching view of system. (e.g. desktop)
- Encompasses concept of software architecture
- Provides a sense of cohesion
- Often suggests a consistent vocabulary

Continuous Integration

Code is integrated, debugged, and tested in full system build at most a few hours or one day after being written.

- Maintains a working system at all times
- Responsibility for integration failures easy to trace
- If integration difficult, maybe new feature was not understood well, so integration could be abandoned

Simple design

Motto: *do the simplest thing that could possibly work.* Don't design for tomorrow: you might not need it!

Testing

Any program feature without an automated test simply doesn't exist.

Test everything that could break.

Programmers write **unit tests**: use a good automated testing framework (e.g. JUnit) to minimise the effort of writing running and checking tests.

Customers (with developer help) write functional tests.

Refactoring

Vital for XP because of the way it dives almost straight into coding.

Later redesign is essential.

A maxim for not getting buried in refactoring is "Three strikes and you refactor."

Consider removing code duplication:

1. The first time you need some piece of code you just write it.
2. The second time, you curse but probably duplicate it anyway.
3. The third time, you refactor and use the shared code.

i.e. do refactorings that you *know* are beneficial.

Note: you have to know about the duplication and have 'permission' to fix it... ownership in common.

Pair programming

All production code is written by two people at one machine. You pair with different people on the team and take each role at different times.

> *There are two roles in each pair. The one with the keyboard and the mouse, is coding. The other partner is thinking more strategically about:*
>
> - *Is this whole approach going to work?*
> - *What are some other test cases that might not work yet?*
> - *Is there some way to simplify the whole system so the current problem just disappears?*

Collective ownership

i.e. you don't have 'your modules' which no one else is allowed to touch.

If a pair sees a way to improve the design of the whole system they don't need anyone else's permission to go ahead and make all the necessary changes.

Of course a good configuration management tool is vital.

Coding standards

The whole team adheres to a single set of conventions about how code is written (in order to make pair programming and collective ownership work).

Sustainable pace

a.k.a. **40-hour-week**, but this means not 60, rather than not 35!

People need to be fresh, creative, careful, and confident to work effectively in the way XP prescribes.

There might be a week coming up to deadlines when people had to work more than this, but there should't be two consecutive such weeks.

Mixing and Matching: yes or no?

Can you just use some of the XP practices?

Maybe... but they are *very* interrelated, so it's dangerous.

If you do collective ownership but not coding standards, *the code will end up a mess.*

If you do simple design but not refactoring, *you'll get stuck!*

XP practices support each other in many ways.

Where is XP applicable?

The scope of situations in which XP is appropriate is somewhat controversial. Two examples:

- There are documented cases where it has worked well for development in-house of custom software for a given organisation (e.g. Chrysler)
- A decade ago it seemed clear that it wouldn't work for Microsoft: big releases were an essential part of the business; even the frequency of updates they did used to annoy people. Now we have automated updates to OSes, and Microsoft is a Gold Sponsor of an Agile conference

XP does need: team in one place, customer on site, etc. 'Agile' is broader. Other Agile processes include Scrum and DSDM.

Relating different processes

Agile home ground	Plan-driven home ground	Formal methods
Low criticalility	High criticality	Extreme criticality
Senior developers	Junior developers	Senior developers
Requirements change often	Requirements do not change often	Limited requirements, limited features
Small number of developers	Large number of developers	Requirements that can be modeled
Culture that responds to change	Culture that demands order	Extreme quality

Lecture 14 - How to improve the quality of your processes

Why improve a process?

- To manage (reduce, predict and plan for) risks
- Bad things that might happen to you

Risk kinds

Risk may be categorised by kind of impact they have

- **Project risks** affect schedule or resources
  - e.g. staff loss, management change, missing equipment
- **Product risks** affect software quality or performance
  - e.g. changing requirements, delays in requirements analysis
- **Business risks** affect the software developer or buyer
  - e.g. mis-estimation of cost, competitor gets to market first

Categorisation not exclusive.

Planning for risks

Identify the risks early, in various possible categories.

Analyse each identified risk. Is it minor, major, serious, fatal? What is the chance of it happening?

Plan how to cope with each risk. Can it be avoided by reducing the probability of occurrence? Can you plan to minimise the effect if it does happen? What is your contingency plan if it does happen?

Example:

- The lead designer on the human interface team leaves.
- (Bespoke software) The customer changes the requirements to adapt to to a new company acquisition.
- (Shrinkwrap software) A competitor releases a product that already has a key feature of the product you are developing.

Software quality

Anything that the customer cares about.

Approaches to improving quality

May focus on:

1. The software product itself - verification, validation, testing, code/design reviews, inspections, walkthroughs are product-focused approaches to quality improvement.
2. The process by which the software is produced.

Process focus

Advantages:

- Potential to improve *all* products
- Possibility of certifying a whole organisation
- Some important things, especially planning (and thus time-to-market, cost, etc.), are hard to approach in any other way

Disadvantages:

- Done badly, can easily prove very costly with low benefits: easy to spend time and money without improving *any* product

Centres of process-focused QA

Possible directions of influence, each with own philosophy.

1. Organisation -> Project -> Individual

   Organisation's management decrees, influencing projects.

   Project managers direct individuals into desired behaviour.

2. Organisation <- Project <- Individual

   Individuals introduce improvements which are rolled up and out to the rest of the project.

   Project improvements spread to the rest of the organisation.

Making these centres work together productively depends on the software engineering culture of the organisation and attitude to work of the individual.

Areas and terminology

Quality planning: how will you ensure that this project delivers a high quality product?

Quality metrics: what measurements must you make in order to tell whether what you're doing is making the difference you intend?

Quality improvement: what can you learn from this project to help you plan and run the next one better?

Quality control: how can you ensure and prove that your quality plan was followed?

Quality assurance (QA): an umbrella term for the whole field.

Standard QA models

Two examples with different aims:

- **CMMI (Capability Maturity Model Integration)** from Carnegie Mellon's Software Engineering Institute, a development of the very popular Capability Maturity Model (CMM). Crucial idea here is that maturity increases: quality planning, control, and **improvement** framework.
- **ISO9000** - family of quality standards. Less emphasis on improvement: quality **control** framework.

These can be complementary (resolution of a historical argument about which should be used).

Standards

There are many types of standard. Coding standards tell you how to format code. Documentation standards tell you how to organise documentation. Quality standards tell you how to organise quality control.

ISO 9001 is an international standard for quality assurance. It specifies **how** to specify documents and procedures that a company should follow in its quality control. It **does not** specify or require any level of product quality.

CMMI

The integrated Capability Maturity Model is a successor to the influential CMM. It provides a (generic, specialisable to software) description of process areas, goals associated with each area, and practices that may achieve goals. Organisations are assessed at a maturity level according to **how** they achieve goals and follow practices. Levels are:

1. **Initial:** goals may be met, but by heroics.
2. **Managed:** documented plans, resource management, monitoring, etc.
3. **Defined:** organisation defines process framework. Measurements collected for use in improvement.
4. **Qualitatively managed:** use measurement and statistical etc. methods during process.
5. **Optimising:** process improvement happens, driven by quantitative measures.

Total Quality Management

One of the inspirations of CMMI.

Plan, Do, Check, Act - the Deming cycle.

TQM embodies the idea that improving quality is everyone's job - not just that of the QA department.

Typical principles:

1. Quality can and must be managed.
2. Everyone has a customer to delight.
3. Processes, not the people, are the problem.
4. Every employee is responsible for quality.
5. Problems must be prevented, not just fixed.
6. Quality must be measured so it can be controlled.
7. Quality improvements must be continuous.
8. Quality goals must be based on customer requirements.

Six Sigma - "TQM on steroids"

Bottom line

Things only get better when those involved:

- Have enough information to tell what's wrong
- Think intelligently about it
- Plan how to improve
- Actually make sure the plan happens
- Check whether it worked

Many ways to achieve this, many ways to fail. Not specific to software.

Cost estimation

Before starting, or bidding for, any significant project, need to know (estimate) how much it will cost.

Many things are factors in this estimation: but the main factors are software size and complexity, and engineer productivity.

Productivity = (software size x complexity) / time required

Lines of Code (LoC)

A simple measure of software size.

Not very meaningful by itself. What is a line of code?

How many lines of Haskell correspond to how many lines of Java correspond to how many lines of C? What about library routines?

Still widely used; alternatives like 'function points' exist.

Estimation

Metrics are all very well, but how do you estimate them for software that doesn't exist yet? Some approaches:

- **Algorithmic cost modelling:** develop (from past data) a model relating size/complexity to ultimate cost.
- **Expert consensus:** get a bunch of expert estimates. Compare, discuss, repeat until convergence.
- **Analogy:** relate the cost to that of similar completed projects.
- **Available effort**
- **What the customer will pay**

COCOMO

Constructive Cost Model is a long-standing algorithmic model, publicly available, well supported, and widely used. Basic idea:

$$\texttt{Effort} = \texttt{A} \times \texttt{Size}^{\texttt{B}} \times \texttt{M}$$

where

- 'Effort' is measured in person months
- A is a constant, dependent on kind of software and developing organisation
- B typically ranges from 1 to 1.5
- M is a *multiplier*. Product of 15 factors, effort adjustment each typically ranges from 0.9 to 1.4, that are derived from ratings of attributes such as *required reliability, required time to market,* and *software engineer capability.*

Getting good values for A, B, and M is highly non-trivial.

Considerable data available. Versions/sub-models/tweaks available to account for factors like reuse, generated code, etc.

Why do projects almost always slip

Relative to human intuitions of how long should they take?

(This is why we need something like COCOMO).

The Rational Planning of (Software) Projects by Mark C. Paul discusses the effects of three features of human feature:

- "People tend to be risk-averse when there is a potential of loss"
- "People are unduly optimistic in their plans and forecasts"
- "People prefer to use intuitive judgements rather than (quantitative) models"

It goes on to discuss how a framework like the CMM can help.

Gantt charts

An example of a project planning tool, to help with scheduling.

Divide project into tasks, with milestones at the end. Analyse (e.g. in graphical network) dependencies between tasks. Now lay these out as bars running across time, respecting dependencies. This reveals the critical path of tasks for the project. Optionally, show permissible slippage with shaded bars.

Project tracking

The project manager needs to decide how and what to track. Example:

- How much time each person spent on each task, and when?
- Just total effort expended?
- Something in between?

Aim is to find a happy medium between having too little information to tell whether things are OK, and so much that it's very time-consuming to manage the information. Ideally want meaningful info!

Tools are available and helpful, especially for big projects, but they don't create the data or decide what to do as a result.

Revising the project plan

As the project goes on, estimates have to be revised in the light of progress, unforeseen problems, etc. Typically a large project will replan once a week.

Depending on the circumstances slippages may mean

- Reallocating resource
- Cutting functionality
- Asking the customer for more money
- Losing profit

Lecture 15 - Non-functional requirements, Metrics, and Reliability

NFRs

Concern the whole system, not just the software.

- Ways the system needs to be related to other systems and versions of itself
  - Flexibility, maintainability, reusability, portability
- Properties of the system in use (including things the system must not do or allow to be done)
  - usability, dependability (safety, reliability, availability, resilience), efficiency (performance, resource usage), security (integrity, confidentiality, availability), scalability

NFRs in the development process

Must be identified along with functional requirements - at the end is too late (requires special care in agile developments)

Often tied up with architectural decisions: hard to modify late. For example, if you choose the AppFuse framework for your system, you are locked into its security model.

How much of an NFR is needed? Often essential to

- *Quantify* the requirement
- Have a way to *measure* the system - 'metrics'

Useful metrics should ideally be

- Measurable - e.g not someone's opinion of how complex something was
- Specified with a precision - i.e. a range in which measured values have to fall
- MEANINGFUL! - there must be some reason to believe that numbers for the metric have something to do with something we care about

Reliability

A key non-functional requirement in many systems.

How does one specify reliability?

Several ways - most appropriate depends on the nature of the system.

POFOD

Probability of failure on demand is the probability that the system will fail when service is requested.

- Mainly useful for systems that provide emergency or safety services.
- e.g. the emergency shutdown in a nuclear power plant will never be used - but if it is, it shouldn't fail
  - 'New' Sizewell B Primary Protection System specified 0.0001 - and achieved 0.001 in testing

How to evaluate? Repeated tests in simulation. Expensive?

ROCOF

Rate of occurrence of failure is the number per unit time of failures (unexpected behaviour). 'Time' may mean elapsed time, processing time, number of transactions, etc.

- Mainly used for systems providing regular service, where failure is significant.
- E.g. banking systems
  - VisaNet processes over 10^9 transactions/day. Failure rate is not published, but probably (much) less than 10^{-5} failures/transaction.

MTTF and MTBF

MTTF is Mean Time To Failure.

MTBF is Mean Time Between Failures.

- Both mainly used where a single client uses the system for a long time.
  - e.g CAD systems - or indeed desktop PCs.
- MTTF used when system is non-repairable.
  - Popular metric for hardware components.
- MTBF used when system can recover from failures.
  - e.g. used for OS crashes.

Note: ***mean* alone is often insufficient to be decisive!**

- Variation matters
  - Whilst the component whose MTTF is shown with the red curve (pointier graph) is more predictable to fail after MTTF whilst the blue curve (less pointy graph with variations) indicates that the failure might occur much before (or much after) the MTTF.
  - An ideal curve would be sharp and pointy.

Availability

The proportion of time that the system is 'available for use'. Often quoted as 'five nines', meaning 0.99999, 'four nines', etc.

Appropriate for systems offering a continuous service, where customers expect it to be there all the time. ('Five nines' is achieved by large data processing systems (e.g. VisaNet) - running on IBM mainframes, not PCs!)

Difference between availability and ROCOF

Availability takes the time to recover from a failure into consideration too.

Availability gives a better overall picture.

For instance, your ROCOF might be once in a year, bit if it takes you 3 months t recover, that's not high availability.

Metrics summary

Metric	Appropriate when...	Example
POFOD	System is rarely used	Airliner oxygen masks
ROCOF	Regular service, failure significant	Online site, correct customer charge
MTTF and MTBF	Single client, extended use	Disks, desktop OS crash
Availability	Continuous service	Online site up

Lecture 16 - IP and Licensing

Intellectual Property

IP is a monopoly right the exploit an intangible product of human thought or labour.

Modern intellectual property rights

Fall into several broad classes.

- **Copyright** applies to literary or artistic works.
- **Patents** apply to inventions of things or processes.
- **Design rights** apply to the design of products.

None of these were invented with software in mind.

Patents

Arose to protect inventors of physical objects or processes (e.g. steam engine)

Stronger than copyright: stops other people from using/making object, even if they invented it independently.

Duration is shorter - 17 years.

Controversy over what is patentable. Originally, physical things and processes for making physical things.

Extended to 'embodied programs'. Can an algorithm be patented? In U.S. yes, in Europe no (roughly speaking).

In U.S. 'business processes' are patentable. e.g. Amazon has patents on 'one-click' shopping, and on the idea of customers reviewing products!

Patent systems appear unable to cope with the issues of IT patents.

Copyright

The original IP.

- Restricts the ability to copy, adapt, etc. artistic or literary works.
- But no artistic or literary merit required. e.g. a lecture is a literary work for copyright purposes
- Currently subsits (on literature) for life of author plus 70 years.
- The protected rights may be assigned in whole or in part, or licensed in whole or in part, with or without restrictions. (Certain exceptions for moral rights)

Generally, it is accepted that source code is subject to copyright.

Object code and machine code are 'adaptations or translations' of source code, and hence protected.

Running a program necessarily involves copying it!

Scope of copyright

Slogan: copyright protects the expression of an idea, not the idea itself.

Largely by judicial interpretation, this has been strengthened. Characters and settings, plots and stories, have been held to be copyrightable.

What of reverse engineering? Generally held that clean-room reverse engineering does not breach copyright - the function is not protected, the code is.

Almost all code (compiled or not) is licensed to the user within the framework of copyright.

Standard commercial licenses

Most commercial software is licensed either per computer or per user. License grants permission to run program; usually specifically forbids decompilation etc.

- Per computer: usual model for desktop PC software. Easy to arrange, check, and charge for.
- Per user: usual model for more expensive software, particularly SE tools, scientific tools, etc. Licenses may be tied to individuals, or may 'float', using a license server to control total number in use. (e.g. Matlab)
- Site: used to access a website. (e..g Office365)

Shareware/Freemium

- Source not available
- Basic version free
  - May be limited in functionality or lifetime
- Pay fancier versions or extra features

Very common now for apps and games.

Fuller versions can be expensive: 1000GBP+ (e.g. Video editing software)

Free software/Open source

Open source: the source code must be available free of charge and open for review and modification by users. Redistribution (even of modified versions) must be allowed.

Often called **FLOSS (Free/Libre Open Source Software)**, *Free* emphasising freedom to redistribute, not that the source code is available at no lost.

There are many widely used licenses: the BSD License, the X11 (MIT) License, the Mozilla Public License, the Artistic License.

GNU Licenses

The General Public License (GPL)

Goes beyond earlier 'free' licenses.

The first *copyleft* license - key property that modifications or adaptations of GPL-ed software, *including* software using any GPL libraries, must be licensed under the GPL and no other license. ('The viral nature of the GPL.')

The GPL does not force you to distribute your modifications; and does not prevent you charging fees for providing your software; and does not prevent you charging for support.

The GPL is incompatible with many other licenses; it is legally impossible to combine incompatible software with GPL-ed software.

Appropriate to use when you want your software to be freely accessible, and you don't want anyone who keeps their software closed to be able to use yours.

The Lesser General Public License (LGPL)

Places *copyleft* restrictions on code, but does not apply these restrictions to other software that merely link with the code.

So proprietary applications can link with LGPL-ed code and not have to be distributed freely, even though the LGPL-ed code is distributed freely.

The LGPL is mostly used for software libraries.

Appropriate to use when you want your software to be freely accessible, but you are happy for it to be included in software that is not freely accessible.

The choice of licensing model

The range of models varies from 'keep it secret and charge the earth' to 'give it away'. The choice depends on many things:

- Philosophy: some consider restricting software to be unethical.
- Legal constraints: you may have used other software that restricts your choices.
- Business constraints: is the software your core business, or do you use it to leverage other (chargeable) activities?
- Support: do you want to get the users to contribute?

Lecture 17 - Ethics

ACM/IEEE Software Engineering Code of Ethics

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession.

In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. **PUBLIC** - software engineers shall act consistently with the public interest.
2. **CLIENT AND EMPLOYER** - software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. **PRODUCT** - software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. **MANAGEMENT** - software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
5. **PROFESSION** - software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
6. **COLLEAGUES** - software engineers shall be fair to and supportive of their colleagues.
7. **SELF** - software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Case Study: Therac-25

Generally when software contributes to bad outcomes the situation is complex, and allocation of blame is difficult. e.g. Something else also had to go wrong to cause the bad outcome (Ariane 5) or it isn't clear what role software played (Chinook FADEC).

The Therac-25 incident is one of relatively few cases where it is clear that software errors - failures of the software engineering process - caused people's deaths.

What happened?

Therac-25 was a medical liner accelerator - it could produce beams of electrons or X-rays to treat tumours.

Earlier versions used software as a convenience, on top of hardware that could stand alone. In particular, they had hardware safety controls.

Key danger: the machine's 'raw' electron beam is harmful. It must be treated to produce either a 'safe' electron beam or a 'safe' X-ray beam. This involves a turntable being correctly positioned. **It wasn't.**

1. **Original safety feature:** the machine's settings were made manually, then reentered at a terminal. The computer checked for a match. Operators thought it inefficient to reenter data. Replaced by a 'hitting return' version.
2. **2.1 Reports of errors were frequent**
   2.2 Operators were told in training that the machine was safe
   2.3 Error messages were cryptic.
   2.4 It was possible to respond to an error with a 'proceed'
   So operators did ignore and override error reports.
3. **Unprotected shared data - race conditions.**
   Result: malfunctions that appeared only on particular, fast editing sequences involving the up-arrow key.

The Consequences

Between 1985 and 1987 at least 6 people were seriously over-treated.
At least 3 died.

Root causes

- **Software was regarded as safe, compared to hardware.**
- The manufacturer made unjustified and wildly optimistic claims about how unlikely errors were, which misled clinicians.
- Poor software specification.
- Lack of defensive design.
- Poor software testing.
- Incident reporting was not systematic.

Improving practice

Technical fixes, e.g.: there should have been an independent, simple, verifiable double-check of the safety of what the machine was about to do.

Process fix, e.g.: software should have been regarded as risky, and specified and verified accordingly.

Ethical angle

The ACM/IEEE code of ethics was violated, for example because:

1. **PUBLIC** - AECL (manufacturer of the Therac-25) employees failed *"1.06. Be fair and avoid deception in all statements, particularly public ones, concerning software or related documents, methods and tools."* e.g. by claiming it was impossible for Therac-25 to overdose patients.
2. **PRODUCT** - they failed *"3.10. Ensure adequate testing, debugging, and review of software and related documents on which they work."* Testing was severely inadequate.
3. **PROFESSION** - they failed *"6.07. Be accurate in stating the characteristics of software on which they work, avoiding not only false claims but also claims that might reasonable be supposed to be speculative, vacuous, deceptive, misleading, or doubtful."*