1 - Git

Git is a **version control system** for managing code which works through the use of repositories. Users interact with 2 repositories, the **local repository** and the **remote repository**. The local repository is the working directory for the git project and keeps all work done on the local machine on a project. The remote repository is a shared repository, where multiple local repositories may be attributed to it and is where code for a project is stored and accessed. To initialise a new git working repository, use the **Git Clone** command.

Git conflicts occur when you try to push code to a repository but you are missing commits in your local repository. In order to avoid **Git Conflicts**,the general workflow is: pull, commit, pull, push. Diff tools can be used to deal with Git conflicts. **Branches** are used to create a new target destination for work commits, and are generated from a specified point in the source code timeline. Branches can be brought back into the source code using merging. There are three main types of merging:

- Reconciling - Where the code in the branch is simply merged with the most up to date code in the source (master) branch
- Linearization - This is where commits in the source are restructured such that all the commits from the source and branch are amalgamated into one branch with commits ordered by time.
- Rebasing - Commits are reconciled by applying one set of commits on top of the other in the destination branch.

Git **Pull requests** let everyone know you've finished a piece of work and are literally requesting them to pull it and review the code.

2 - Software Processes

There are certain software processes that should be followed when engineering software, in order to ensure projects are completed in a timely manner using proportionate resources. These will be covered in the coming chapters. There are certain implications when engineering systems, and these are:

- Repeatable - Should be able to re-use tools and techniques across projects
- Responsible - Should take account of best practices and ethics
- Systematic - Should be planned, documented and analyzed
- Measurable - Should have objective support both for the products and the projects themselves.

There are generally 6 main stages in software engineering:

Analyzing, Planning, Designing, Implementing, Deploying and maintaining.

3 - Software Lifecycles

The following are lifecycles implemented in both companies and on individual projects.

Build and Fix

Build anything then wing it. Might work for smaller projects but generally never works for big projects. There's no real plan, deficiencies propagate, and as there are no documents there's no maintenance.

Waterfall Model

The stages are: Requirements -> Specify -> Design -> Implement -> Deploy -> Operate -> Retire/ Maintenance

Each stage gets verified before proceeding to the next stage, faults cause the process to roll back a stage or more, and maintenance can cause changes at some point, but then the same process continues from that point. However, Faults discovered near the end are extremely expensive to fix, and there is no need to show the clients anything but documentation until the end. This lifecycle also begs the question, "why weren't all the problems found in design?"

In conclusion, this lifecycle is adequate for something you really know how to build.

Incremental Model

Very similar, except there is a stage added before design called architect where the framework of the system is created. The idea is then to iterate through the steps Design through Deploy quickly, adding new features to the product every iteration.

Rapid Prototyping Model

Very similar to the incremental model, however a prototype is produced before the specify stage to ensure everyone is on the same wavelength. Prototypes consist of screenshots, key algorithms and no fine details or finesse. Prototypes must always be dumped and never be used as a building block for the real development. The reason being they are built without architecture, lashed together in a hurry and features are added ad lib.

Spiral Model

This model consists of four stages that development progresses through a number of times until the product is completed. These are:

- Planning
    - Requirements document specifications and are gathered and used to determine the outcome of the current spiral
- Risk Analysis
    - Processes are undertaken to identify risks and alternate solutions to avoid costly risks. A prototype can be produced at the end of the risk analysis phase, but should never be used in production code.
- Engineering
    - This is the phase where code is actually written for the current spiral, and tested towards the end of the phase.
- Evaluation
    - This allows the customer to evaluate the output of the project to date and review before the project continues to the next spiral.

Spiral model incorporates a bit of all the previous models, there is a lot of emphasis on risk management and it is big on re-use, not just in code but in all stages.

Win-Win Spiral

This software process model is the essence of the spiral model with added functionality. In the following, the first 3 stages are Win-Win Spiral Extensions, whereas the final 4 belong to the original spiral model.

- Identify Next Levels Stakeholders
- Identify Stakeholders win conditions
- Reconcile Win conditions, establish next level objectives
- Evaluate product and process alternatives, Resolve Risks
- Define next level of product and process - including partitions
- Validate product and process definitions
- Review, Commit

4 - Agile Processes

Agile Processes are lightweight working processes as opposed to software lifecycle models as outlined in the previous chapter. They give priority to:
- Working Software
- Customer Collaboration
- Reactiveness to Change

Two main Agile Processes we'll be looking at are extreme programming (XP) and Scrum.

Extreme Programming (XP)

There are four main stages in XP: Planning, Designing, Coding and Testing.

Planning

Release planning in XP creates the schedule. Frequent small releases are scheduled and the project is divided into iterations. Iteration planning starts each iteration, and user stories are written for each iteration. User stories are short scenarios written in non technical syntax by the client which describe the requirements of the iteration.

Designing

When designing in XP, **simplicity** is key. A **system metaphor** is used (short story used by everyone in involved which describes the system functions), **Class Responsibilities Collaboration** (CBC) cards are used in design sessions. **Spike solutions** are used to reduce risks (Short programs that highlight possible solutions to technical and design problems), **No functionality** is added early and unnecessary infrastructure is never added. **Refactor** whenever and wherever, only if necessary.

Coding

When Coding in XP, the Customer is always available, Code must be written to agreed standards, Test First, All production code is pair programmed, Only one pair integrates code at a time, Integrate Often, Use Collective Code ownership, Leave Optimisation until Last, no Overtime.

Pair programming is done using two people, one keyboard approach. Generates better, cleaner code and allows for faster knowledge transfer.

Testing

All code must have unit tests, all code must pass unit tests before deployment, tests are created when a bug is found and acceptance tests are run often and the score is published.

Scrum

In scrum project development, series of work is done in sprints. A sprint is a set period of time in which specific work has to be completed. Work is split up into three sections:

- Product Backlog
- Sprint Backlog
    - This is work to be completed during the current sprint

- Working increment of the Software

Scrum Planning

The plan is updated with every daily scrum (Work done during a day in a sprint), where there are continuous updates through the day associated with continuous commits. The design team is updated with every sprint.

Sprint Planning

The team decides what the scope of the sprint is and what the final product at the end of the sprint should be. In sprint planning you select the functionality for the sprint and identify the tasks for each functionality.

5 - Planning

User Stories

- Three key Words when writing User Stories
  - When
  - What
  - Why

*“In the role of a **Student**, I want to **See** the list of courses available for my program”*

*“**When** registering to a course **I want to see** the list of requisites for the course, **so I can** select the other courses to take.”*

*“**Mentors can** change students to a new program **from** an administrator view”*

- Create a Student
- Create a Course
- Create a Program
- Add requisites to course
- Associate Student to Program
- Associate program to Course
- List all courses for a program
- Associate Course as Requisite of another Course
- Define the types of Requisites
- List all requisites for a course
- Create a mentor
- Associate Student to a mentor
- Login with permissions (Student / Admin)

- Mentor can change Students program

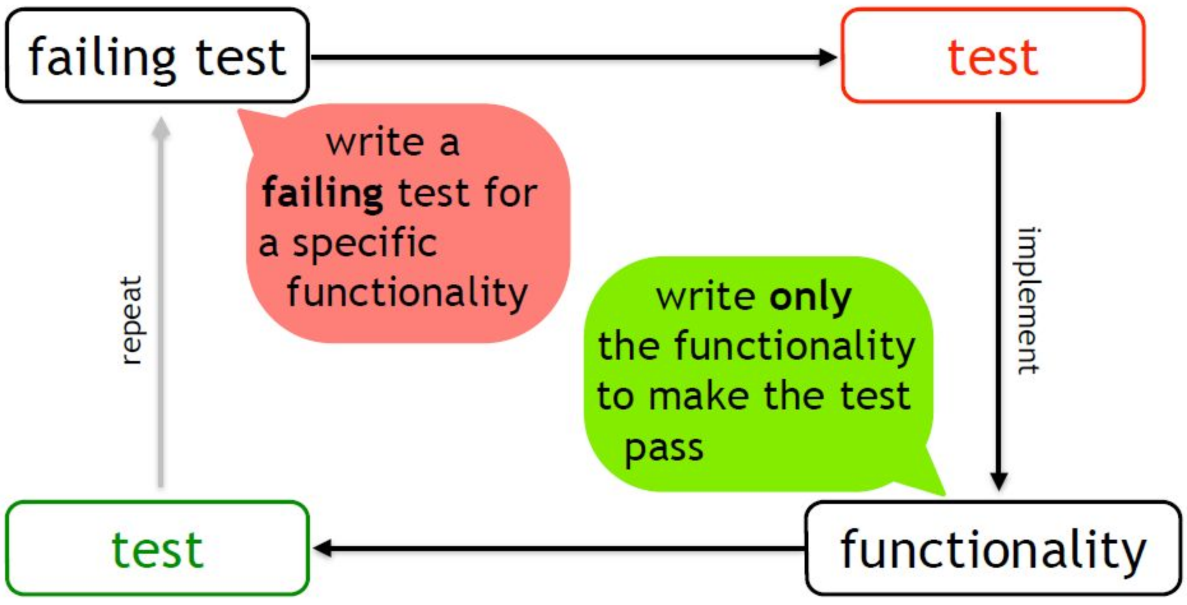
Planning Poker

This a task complexity estimation game. Once user stories are selected they are ranked by all team members. The stages are as follows:

1. The Story is clarified, avoiding any time references
2. Every Participant votes their estimate
3. Low and High estimations explain themselves
4. Repeat until consensus is reached

6 - Test Driven Development

This is important to better understand the code requirements and write better code. The process is:


failing test
test
write a **failing** test for a specific functionality
implement
write **only** the functionality to make the test pass
functionality
test
repeat

Unit Testing

Unit testing is a single, isolated test in a well-defined scope. The dependencies of the test target are ignored. All methods that do something within the isolated code fragment are tested, with the expected result being the behaviour defined in the method.

Integration Testing

This tests integration between multiple modules in the system. Dependencies are **explicit** and test Data is introduced. The aim of integration testing is to test the integration between modules.

Regression Testing

The purpose of this is to test if changes break existing code, and to look for side-effects to newly added functionality. We test changed Artifacts, and Run test suites capturing modified components (a.k.a automate unit testing).

Smoke Testing

This tests that everything works in general using isolated tests. We test basic test cases for each component, Start building components together and perform unit and integration testing.

System Testing

This is ensuring everything is working with the final configuration, it tests that the system configuration works in the deployment environments specification.

Assertions

Assertions are predicates that test a condition that should hold in a program. They test for equality, existence, logic values and identity.

Conclusion

Pro	**Con**
Useful for Continuous Integration	TDD Often sacrifices Refactoring
Drives the development of a system	**Does not** replace any other quality assurance mechanism
Reduces Debugging Time	

7 - UML

No Design Without Analysis

Analysis:

- Extract the main actors/entities in the system
- Extract characteristics for the entities
- Extract behaviour for the entities

Design:

- Create abstractions of the entities
- Capture Entities properties and behaviour
- Identify interactions between entities

Importance of planning before Design

- Danger of over-committing to a technology too early in the project
- OO may not always be the best solution
- There are usually many solutions to a problem

Conversations about requirements must take place with customer at a problem based level, to determine the right system to meet their needs.

Analysis Vs Design

Analysis is about understanding the problem
Design is about simulating and manipulating the problem

Structured graphical notations are usually used when analysing a problem so that users can intuitively follow conversation, while software team intuitively understand how to proceed.

OO Analysis and Design

The heart of Object Oriented problem solving is the construction of a model. The object abstracts the essential details of the underlying problem and stores it in a simple environment.

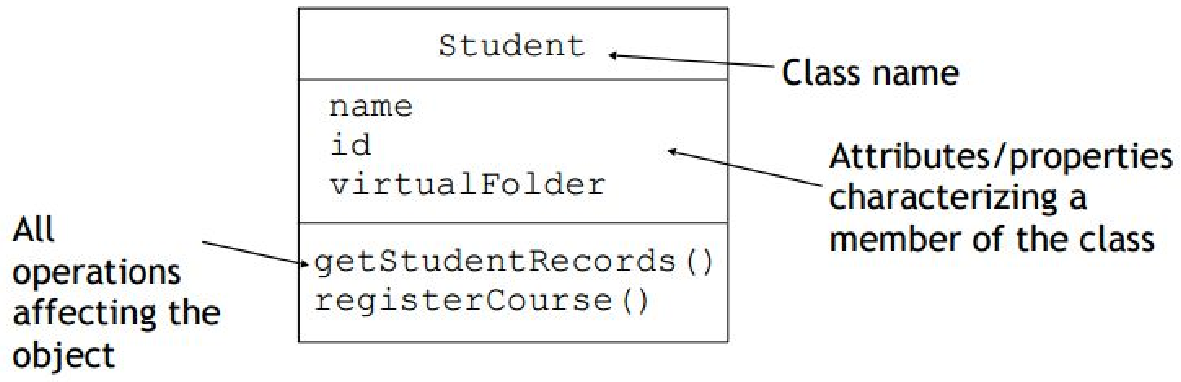
OO Analysis is Real world entities represented as objects.
OO Design is decomposing a system into objects.

Object Orientation

This is managing the complexity of a problem using decomposition. It breaks the problem into smaller problems, and then addresses each of these sub-problems - A literal “Divide and Conquer” approach.

8 - Class Diagrams

These are a description of common “problem domain” objects. It describes a system by showing its classes and their relations with each other. They are static, meaning they display what interacts with each other but not what happens when they do interact. Class Structure is as follows:


Student
name
id
virtualFolder
getStudentRecords()
registerCourse()
Class name
Attributes/properties characterizing a member of the class
All operations affecting the object

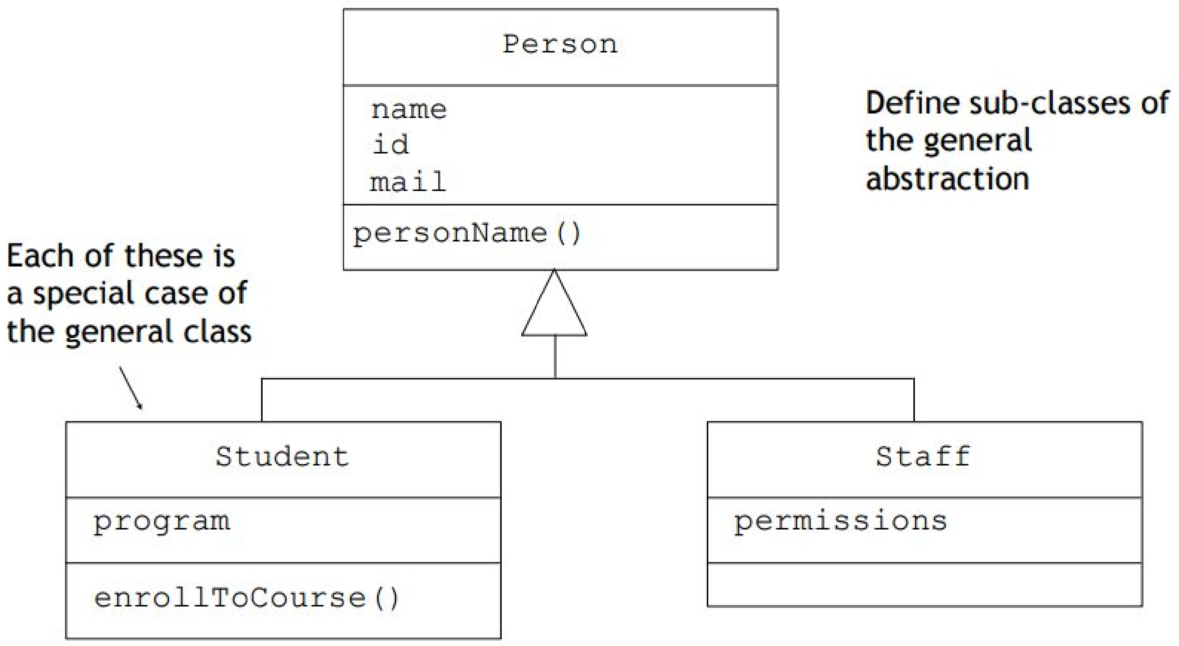
Relations in uml class Diagrams include:

1. Generalization Relation
2. Realization Relation
3. Association Relation
4. Dependency Relation
5. Aggregation Relation
6. Containment Relation

Generalization Relation

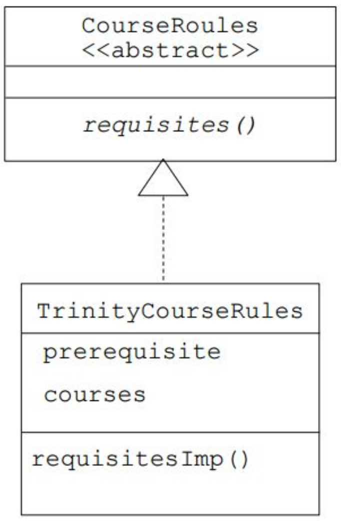
This defines subclasses of the general abstraction, with subclasses being individual general classes.

Consider a class Person. Person can be further classified as a Student or Professor in a University login system. Subclasses of the Person Class inherit all the methods of the Person class.


Person
name
id
mail
personName()
Define sub-classes of the general abstraction
Each of these is a special case of the general class
Student
program
enrollToCourse()
Staff
permissions

Realization Relation

This indicates one class implements behaviour specified by another class. It is the definition of an **abstract** class, that **realizes** the implementation of another class.

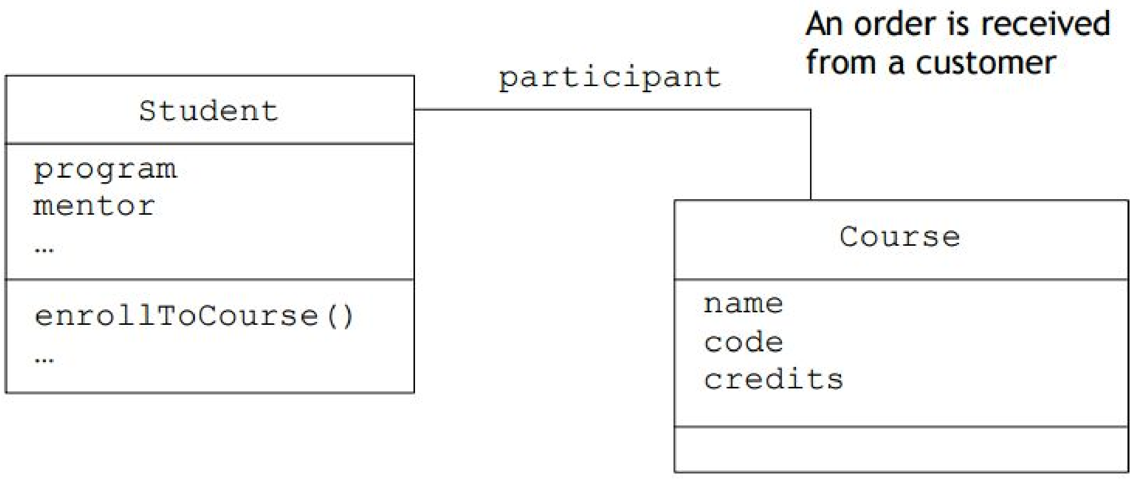


<u>**Generalization vs Realization**</u>

- Generalization should be used to avoid code repetition
- Realization should be used when the implementation of an API is not fixed
- Realization should be used when objects borrow multiple behaviour

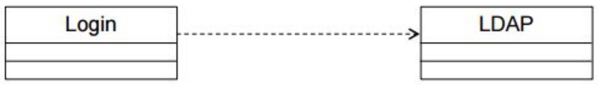
Association Relation

This relation expresses relations between classes in the problem domain (Such as a student enrolled in Course)

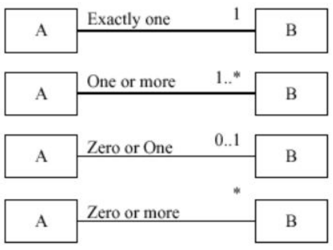


Dependency Relation

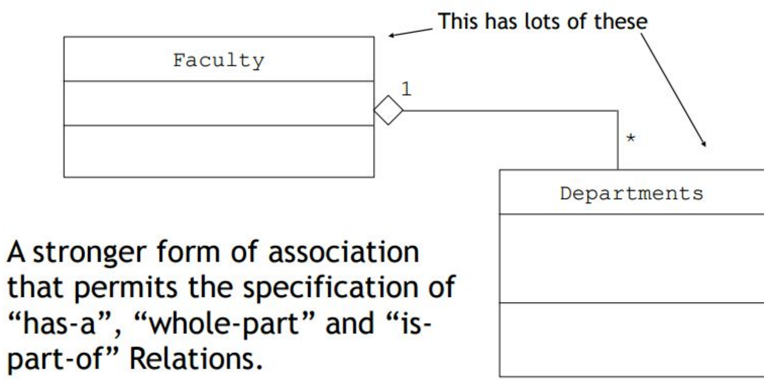
Dependency is a using relation that states a change in the specification of one class may affect the functionality of another class.



Class Associations Multiplicities

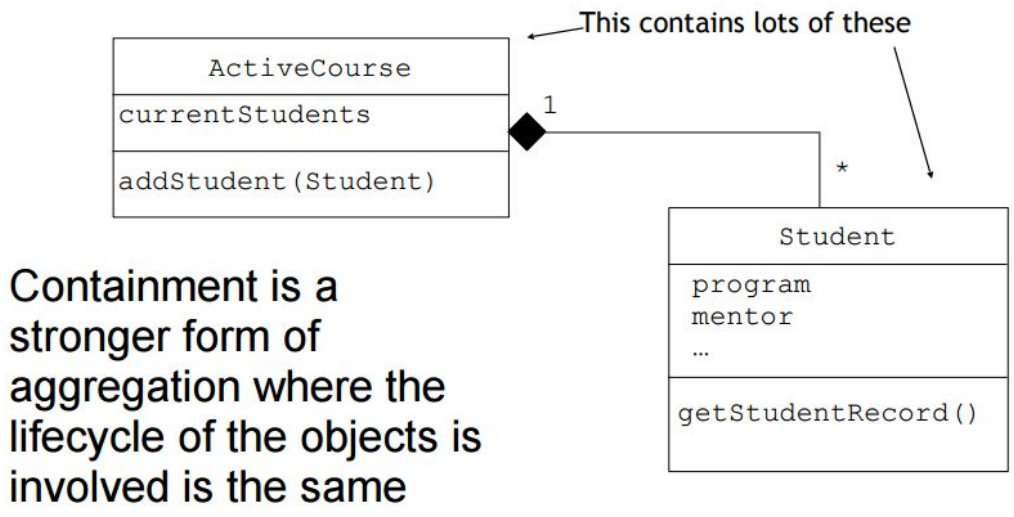


Aggregation Relation



A stronger form of association that permits the specification of “has-a”, “whole-part” and “is-part-of” Relations.

Containment Relation



Containment is a stronger form of aggregation where the lifecycle of the objects is involved is the same

Encapsulation

This is a metric of how “together” an object is and how well it works together with other objects

Coupling

This is a metric of interconnectedness of components, based on how many connections exist between components.

10 - Architecture Views

View	**Monolithic Application**	**Distributed Systems**
Logic View	Class and Sequence Diagrams	Class and Sequence Diagrams, State Charts
Development View	none	Component Diagrams
Process View	none	Activity, Sequence Diagrams
Deployment View	none	Deployment Diagram
Scenarios	Case Diagrams	Use Case, Activity Diagram

11 - Design Patterns

1. Dynamic Proxies

A proxy is a wrapper that adds functionality to another class without changing the code in that class. Its usage examples include:

- Adding security to an existing object (proxy decides when client can access an object)
- Simplifying an API
- Add a thread-safe feature to the object
- Changing parameters when calling functions through a proxy

A dynamic proxy allows you to pass multiple interfaces to a single proxy. This simplifies code and allows the proxy to grant/deny access to different interfaces depending on certain conditions. The interfaces can also be added dynamically.

2. Interpreter

This is based on the interpreter problem, where a user inputs natural language and the interpreter must correctly break down the inputted command, ignoring superfluous words and manipulating the input to return the requested result. This can be done using a simple scripting language, however it is inefficient to implement. The context for the grammar grows with every rule, and the syntax is very strict.

3. Adapter

The adapter pattern allows the interface of an existing class to be used from another interface. It is often used to make existing classes work with others without modifying their source code, for example calling @Override in java.

4. Aspects (Not a pattern)

This was developed as a way to address concerns about code which jumped from area to area in a file. The aim was to improve modularity and structure of code. An aspect is used to implement a feature used multiple times across methods, classes and object models. It essentially just groups code blocks together.

12 - Project Postmortem

This is the analysis of completed project data (estimated times vs real times, defects etc), It uses collected data to refine future estimates, documents the results of completed project analysis and the project itself. It also provides insight on the effectiveness of a project.

13 - Risk Management

In software engineering, a risk is something that, if it happens, will cause the project a problem. Risks include:

- Computer crashes or thefts
- Staff illnesses or resignations
- Unexpected complexities
- Unplanned changes
- Office or building fire

Risk Engineering

The approach to risk engineering is as follows:

Risk Identification	List all the risks which would affect the outcome of the project
Risk Estimation	Their probability, and their impact if they happen
Risk Evaluation	Rank risks and formulate strategies to deal with them

Risk Planning	Draw up contingency plans
Risk Control	React to problems and minimize their impact on the project
Risk Monitoring	Re-assess everything as the project progresses - risk impacts sometimes vary with time

Risk Exposure refers to how much of a company's assets are tied to this risk. If the risk occurs, what impact on revenue will it have?

Reducing Risks

- Prevent, Duck, and cover
  - Avoid circumstances which make risks more likely
  - Take active steps to reduce likelihood of risk
  - Plan a project to stay away from unacceptably risky areas
- Pass the Buck
  - Insurance companies are a form of risk management outsourcing
  - Get the customer to share some of the risks
- Contingency Planning
  - Plans within plans within plans
  - Spare money, and slack time in the schedule

14 - Software Engineering Requirements

Deliverables/artifacts per development phase

1. Requirements Elicitation
   a. Requirements document (elicitation, analysis, specification, validation)
2. Analysis
   a. Risk Analysis Document, Estimation (resources, time) for all phases
3. Planning
   a. Plan of development **for the project/cycle**
4. Design
   a. Design artifacts (object/class model, sequence diagrams, use case diagrams) **for the project/cycle**
5. Implementation
   a. Code and documentation for the project/cycle
   b. Data Collection Point:
      i. Number of Lines of Code (LOC)
      ii. Complexity
      iii. Number of Entities
6. Support

    a. Tests design, test code and documentation
    b. Data Collection Point:
        i. Bugs Introduced
        ii. Bugs Removed
7. Postmortem
    a. Analysis of collected data, description of artefacts produced during the **project/cycle**

The Scrum process fits into these software engineering requirements as follows:

Requirements	**Scrum**
Requirements Elicitation	
Analysis	User Stories
Planning	Release Planning
Design	Architectural Spike
Implementation	Iteration
Support	Acceptance Test
Postmortem	Release

Architectural Spike

These are basically spike solutions from Extreme Programming. The goal is to increase the reliability of a user story estimation by writing a quick prototype to highlight what can be done to solve a story and possible problems. Architectural spikes should be carried out during a sprint with as few developers as required, and should be given equal priority to other tasks being carried out in the sprint.

Iteration

This is when the new functionality designed for a cycle is implemented. Information about time and effort is gathered and estimations for future cycles are refined.

15 - Requirements Engineering

Requirements engineering is the most important phase of the development process. This is where the project is created. It's the process of establishing and specifying what the customers require from the system as well as its operational **environment** and **constraints** (The problem domain).

Requirements

<u>Requirements are descriptions of the services and constraints of the system.</u> Requirements describe the problems to be solved rather than how to actually solve them. They basically describe the effects the clients want to see in the problem domain.

Phrasing is important in requirements document, as can be seen from these examples:

- "To register a student the system must verify that all the courses in the list of requisites are contained in the list of courses the student has passed."
    - Wrong
- "A student can only be registered on a course if it has completed all the requisites for the course, or has special permission from the mentor"
    - Correct

Requirements are usually presented as abstract statements of the system, but they need to be as specific as possible, without going into the solution domain.

Requirements can serve a dual function:
- They may be the basis **for** a contract, and therefore must be open to interpretation, malleable and subject to change.
- They may be the basis **of** a contract, and must be defined in detail.

Types of Requirements

- User Requirements
    - Statements in natural language plus diagrams of the services the system provides, and its operational constraints.
    - This is written for the customer
- System Requirements
    - A structured document setting out detailed **descriptions of the system's functions**, services and operational constraints. **Defines what should be implemented**, so it may be part of a contract between client and contractor.

System Stakeholders

A system stakeholder is any person or organisation who is affected by the system in some way and so has a legitimate interest in the system. Stakeholder types include:

- End Users
- System Managers
- System Owners
- External Stakeholders

Functional Requirements

Functional Requirements are requirements that can be satisfied by the application of appropriate services in the system. They are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. They can also state what the system should not do.

Functional User Requirements may be high level statements of what the system should do.

Functional System Requirements should describe the system behaviour in detail.

Non Functional Requirements

<u>Non Functional Requirements are constraints on how the system is built, not how the system works.</u>

These are constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards etc. They can also be constraints on the system design and costs, or domain requirements. They often apply to the system as a whole rather than individual features or services.

Process Requirements may also be specified mandating a particular IDE, programming language or development method.

Non-functional requirements may be more critical than functional requirements. If these are not met the system may be useless. There are 3 non-functional requirements specifications:

- Product Requirements
  - Requirements which specify that the product must behave in a particular way
- Organizational Requirements
  - Requirements which are a consequence of organizational policies and procedures e.g., process standards used, implementation requirements
- External Requirements
  - Requirements which arise from factors which are external to the system and its development process.

Performance Requirements

These are the performance parameters of the system services. These characteristics can usually be met through functionality, but may not be the core logic of the system.

Requirements Imprecision

Problems arise when functional requirements are not precisely stated. Ambiguous requirements may be interpreted in different ways by developers and users. In principle, requirements should be both complete and consistent.

<u>Complete Requirements means they should include descriptions of all facilities required.</u>

<u>Consistent Requirements means there should be no conflicts or contradictions in the descriptions of the system facilities.</u>
In practice, because of system and environmental complexity, it is impossible to produce a complete and consistent requirements document.

16 - Requirements Engineering Process

Notes are long as balls. Might get back to it later.
https://www.scss.tcd.ie/~cardozon/courses/CS3012/documents/w9l1-3.pdf

17 - Code Principles

Read the notes, bit too messy to summarise.
https://www.scss.tcd.ie/~cardozon/courses/CS3012/documents/w10l1-3.pdf

18 - Software Maintenance

It is important to maintain the quality of a system as you iterate functionality into it. <u>Software Maintenance is the adjustment of software lifecycles to cope with change</u>.There are four main software maintenance categories:

1. Corrective
    a. Modifications driven by defects found in the code
2. Adaptive
    a. Modifications to match the ever changing environment

3. Perfective
    a. Modifications to improve the performance, changeability, or efficiency of the system
4. Preventive
    a. This is long term. Applications deteriorate due to increasing complexity introduced in corrective, adaptive, or perfective changes. Continuous cosmetic improvements are required to avoid systems' decay.

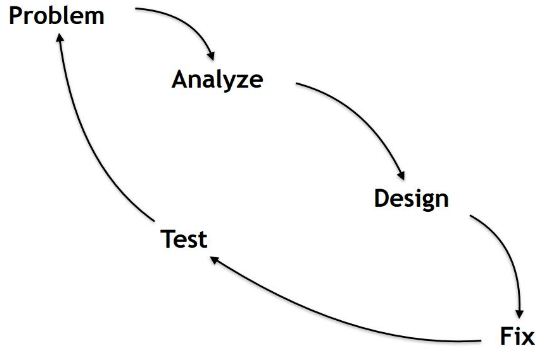
Like with Software Engineering, there are processes for maintenance. The process is:

- Determine maintenance objective
    - Based on the maintenance categories
- Program Understanding
    - Analyzing the program and extent of the required changes
    - Complexity, Documentation, self-descriptiveness
- Generate Maintenance Proposal
    - Proposal for the maintenance activity. How is the system going to change and **why?**
    - Extensibility (How long is it gonna last before it needs more maintenance)
- Account for ripple effect
    - Accounting for the consequence of the modifications in other system components
    - Stability
- Testing
    - Ensure the system is at least as reliable as before
    - Testability

Maintenance Models

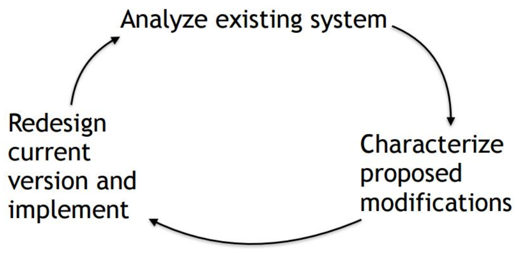
Quick Fix

This is an ad hoc approach, mostly used for putting out small fires. It is a **corrective** model, applicable to continuously running projects, projects with fixed requirements, and for rapid prototyping lifecycles.



Iterative Enhancement Model

This model is based on the existence of an iterative software process model (such as the spiral model)


Analyze existing system
Characterize proposed modifications
Redesign current version and implement

- Analysis of Existing System
    - Inner workings of the system
    - Identity points of modification
- Characterization of proposed modifications
    - Specification of the modification
- Redesign and implement the system to incorporate the modification

This model can be applied to any of the maintenance categories, and is a model to foster software reuse. It can integrate with other maintenance models in the cycle, but is applicable only to iterative development models and this process can only be executed if there is existing documentation.

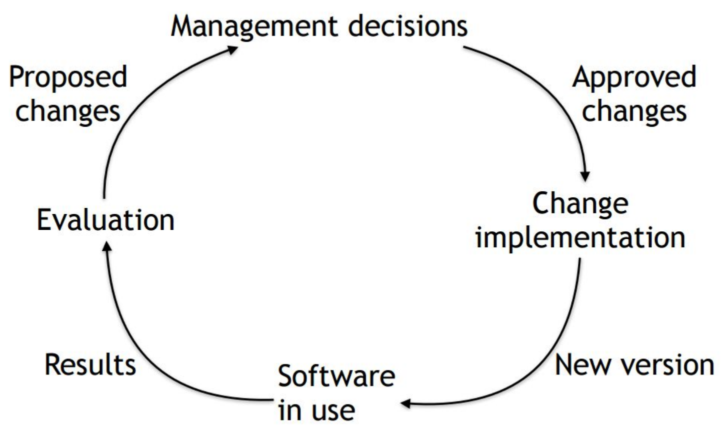
Reuse-Oriented Model

1. Identify the parts of the old system that are candidates for reuse
2. Understand old system parts
3. Modify old system parts to map to the new requirements
4. Integrate the modified parts into the new system

This model is to be used in adaptive or perfective maintenance tasks. It maps to all software lifecycles and can start at any phase of the lifecycle.

Boehm’s Model

Boehm proposed a model for the maintenance process based upon the economic models and principles. The Boehm model represents the maintenance process as a closed loop cycle.



Boehms model is:

- An Economic Biased model
- A model to be used in corrective, adaptive, or perfective maintenance tasks
- Maps to iterative life cycles
- Attached to early stages of the cycle

Maintenance Effort

Maintenance can be classified into two sub divisions, Non-discretionary maintenance and Discretionary Maintenance.

- **Non-discretionary Maintenance**
    - Emergency fixes
    - Debugging
    - Changes to input data
    - Changes to hardware
- **Discretionary Maintenance**
    - Enhancements for Users
    - Documentation improvements
    - Efficiency Improvements

Measuring Effort

Apparently you can measure effort with a formula. The formula is:

$$M = P + Ke^{(c-d)}$$

Where:

- M = Total Effort Expended
- P = Productive Effort (analysis, design, implementation, testing, and evaluation
    - Usually given as person months (PM)
- K = Empirically determined effort constant

- c = Complexity measure
- d = Familiarity Degree

This formula probably requires a lot of background knowledge on the project and team to get a somewhat accurate result - it’s hardly going to be useful if someone inexperienced is just pulling figures out their ass.

Annual Maintenance Effort (AME)

AME = ACT * SDE * EAF

Where:

- SDE = Software Development Factor
- EAF = Effort adjustment Factor
- ACT = Annual Change Traffic