**[INTRODUCTION]**

**Types of Systems**

* Soft – difficult to define precisely
  + Depends on viewpoint of person describing it
  + Difficult to agree on boundaries and behaviour
  + Eg, all human activity systems
* Hard – well defined
  + Easy to agree on boundaries
  + Eg, mechanical operation of car

**Problem** **statement** – clear and concise description of the issue(s) that need(s) to be addressed by a problem solving team. Issues, Goals and Objectives, Method, and Scope.

**Requirements** **engineering** – set of activities concerned with identifying and communicating the purpose of a software-intensive system and the context in which it will be used

Requirement analyst must identify the problem/opportunity

* Which problems need to be solved? (problem boundaries)
* Where is the problem? (context/problem domain)
* Whose problem is it? (stakeholders)
* Why does it need solving? (goals)
* How might a software system help? (scenarios)
* When does it need solving? (development constraints)
* What might prevent us solving it? (feasibility/risk)

Typical problems in requirement specifications:

* Noise
* Silence
* Over specification
* Contradiction
* Ambiguity
* Forward reference

Things you want:

* Valid/correct
* Unambiguous
* Complete
* Understandable
* Consistent
* Ranked
* Verifiable
* Modifiable
* Traceable

**Requirements** **Specification** – Purpose

* Communication
* Contractual
* Baseline for evaluating the software
* Baseline for change control

Re**q**uirements **Specification** – Audience

* Customers and users
* Systems analysts
* Developers/programmers
* Testers
* Project managers

Specification should include

* Functionality
* External influences
* Required performance
* Quality attributes
* Design constraints

**Critical** **System** – a computer, electronic or electromechanical system that the failure of which may have serious consequences (human death/injury, failure of goal directed activity, financial loss/failure of business)

* Safety-critical systems
* Mission-critical systems
* Business-critical systems

**Validation** – ensure that the specification of the software system satisfy the requirements of the stakeholders

**Verification** – ensure that the finished product satisfies the specification

**[REQUIREMENTS ANALYSIS]**

Requirements 🡪 Design 🡪 Implementation 🡪 Verification 🡪 Maintenance

**Requirements** – description of something a product must do/quality it must have

* Functional – things the product must do or provide to be useful for users
* Non-functional – qualities the product must have
* Design – how the system will be designed and implemented

**Constraints** – global issues that shape the requirements

**Assumptions** – describe properties of the environment of a software product that the software can’t directly control, but necessary for overall functioning.

**Verifiable** – A requirement should be verifiable.

* Should be a clear way of checking that an implementation can satisfy requirements
* Eg, by defining test cases for that requirement

**PROCESS**

**1. Scope** – provides a high level view of the planned product: Need, Goals, Case, High-level operational concepts, Stakeholders. Purpose:

* Helps keep bigger picture in view
* Ensure all stakeholders have a common view
* Ideally should remain fixed throughout development process

**2. Use Case** – abstractions from scenarios for usage of the system typically describing series of interactions between users and the system to be developed.

* Help people quickly gain an understanding of what is intended
* Convenient way of generating requirements
* Helps to avoid omitting requirements

**3. Interfaces –** points where the system interacts with environment (users, other software components, physical devices)

* Helps to define boundaries of responsibility for the development teams
* Important for defining ways of testing implemented systems

**[Software Modelling and UML]**

**UML** – visual modelling language for specifying, analysing and designing/documenting essential aspects of software systems before construction. 14 types of diagram.

* Structural: show static structure of the system and its parts on different abstraction and implementation levels and how they relate to each other

Eg, class diagram, component diagram

* Behaviour: shows dynamic behaviour of the objects in a system, which can be described as a series of changes to the system over time

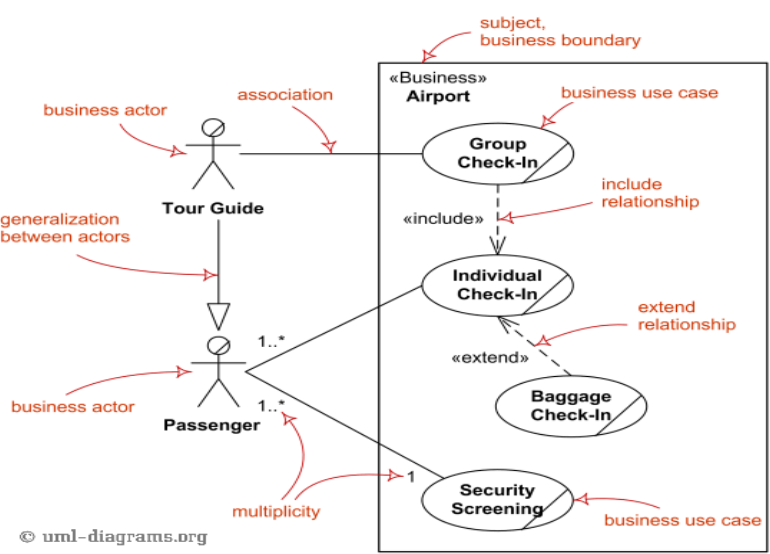
Eg, use case, sequence diagram, state diagram, activity diagram

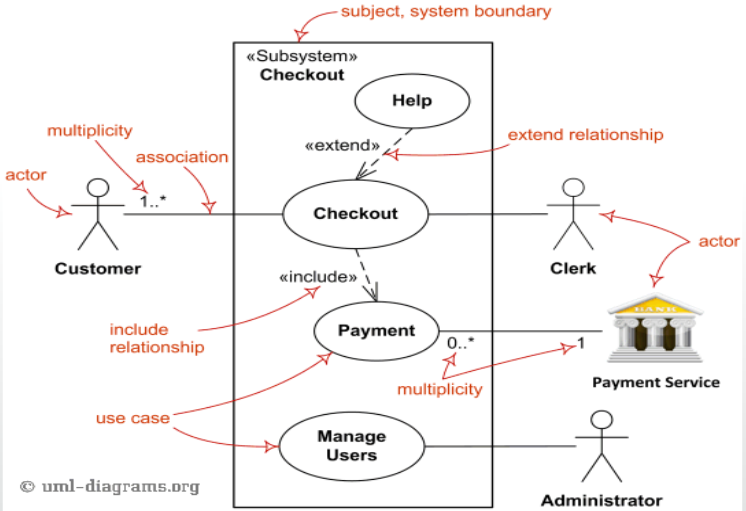
**Use Case Diagrams -** generalised description of how a system will be used**.** Provides an overview of the intended functionality of the system

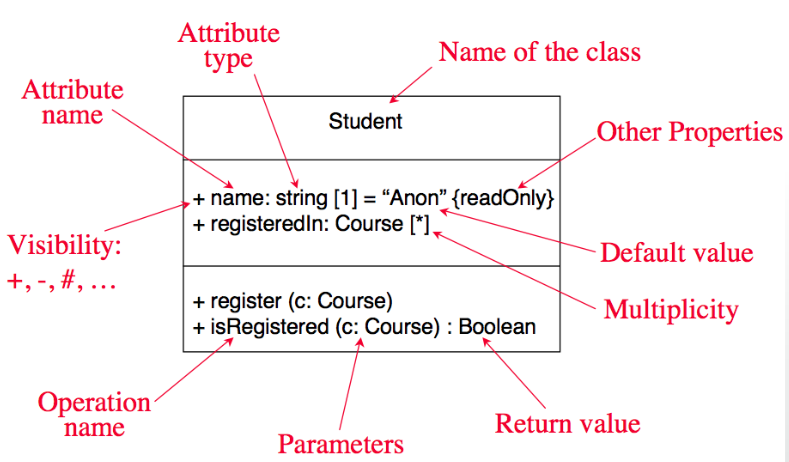
* System boundaries (rectangle)
* Actors (people, eg users)
* Use case: set of scenarios describing interactions between a user and system (ovals)

Relationships:

* Association: communication between an actor and use case
* Generalisations: relationship between one general use case and one specific use case. Arrow towards parent use case
* Include: a particular use case must include another use case to perform function. A dotted arrow from base use class to include use class
* Extend: The extending use class may add behaviour to the base use case. The base class declares ‘extension points’





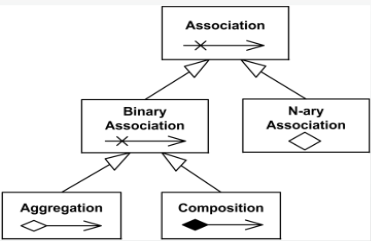
**Class Diagram** - classes describe groups of objects

with similar attributes, common operations,

relationships and meanings

Modifiers used to indicate visibility:

* + Public
* # Protected
* – Private

Relationships:

Directed

* Generalization: an inheritance link indicating one class is a superclass of the other.

Association:

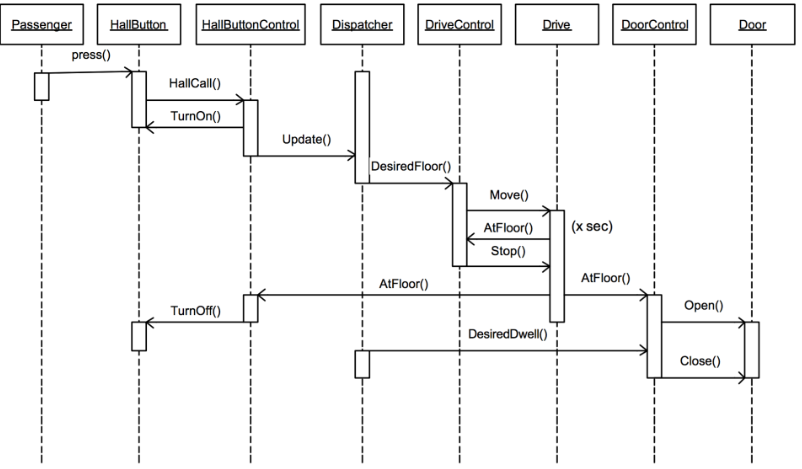
* a relationship between instances of the two classes. Eg, if an instance of one class must know about the other to perform its work.
  + Bi-directional
  + Uni-directional
* Aggregation: association where one class belongs to a collection
* Composition: ‘strong’ form of aggregation. Whole/part relationship.
* easy to discover related data and attributes
* gets a quick picture of important entities
* see whether you have too few/many classes
* see whether the relationships between objects are too complex
* spot indecencies between one class/object and another
* doesn’t discover algorithmic (not data-driven) behaviour
* doesn’t find the flow of steps for objects to solve a given problem
* doesn’t understand the apps overall control flow (event driven/sequential?)

**State Machine Diagram** – graph where nodes correspond to states and directed arcs correspond to transitions labelled with event names. Visualise how objects respond to stimuli

* State: duration of time during which an object is doing an action
* Event: occurs at a point in time and transmits information from one object to another
* Action: occurs in response to an event and cannot be interrupted
* Activity: is an operation with certain duration that can be interrupted by another activity
* Guard: logical condition placed before a transition (true/false)

**Sequence Diagram**

Object interactions

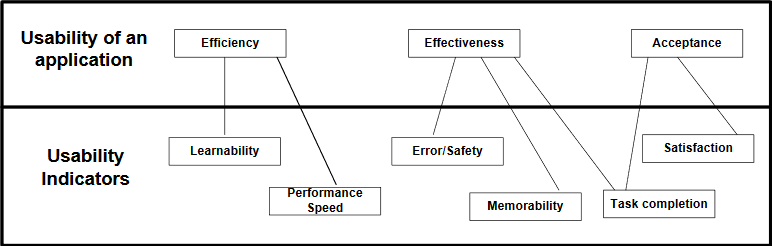
* Self-call – a message that an object sends to itself
* Condition – indicates when a message is sent (only sent is condition is true)

Object life span

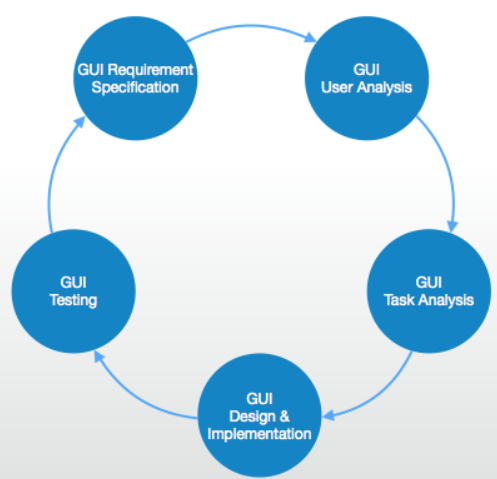
* Lifelines (dotted lines)
* Messages (arrows, direction indication direction of call)
* Activation bar (rectangles)
* Depletion (placing an X on lifeline. Objects life ends at that point)

**[UI DESIGN]**

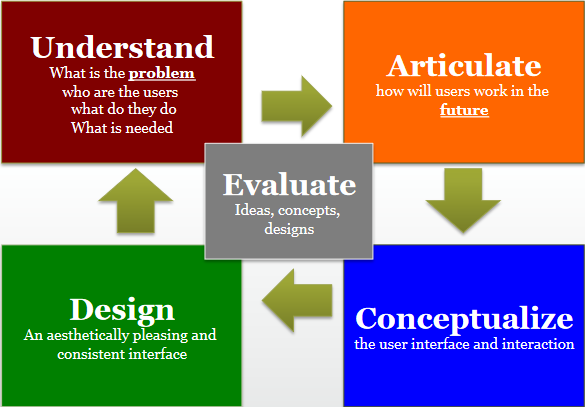
User interface – allows users to accomplish their goals using software without unnecessary effort. Provides appropriate means for using the system in an efficient manner (layout, controls

Effective interaction is determined by and measured using Usability Indicators

UI Design – iterative process including:

* Create prototype of system
* End user testing prototype
* Gathering data from these tests
* Re-designing the interface to address discovered problems

Steps:



Objectives – build an understanding of the design problems and requirements

* Business problems – to identify what we are trying to solve
* User characteristics – what they are like, what do they want, try to avoid, goals, needs
* Use environment – where users use the product/application

Articulate – how will people use the product in the future?

Conceptualize – objectives. Build a clear and shared vision of the product or app

* How UI will appear
* Key user tasks are accounted for
* Select and wireframe the most suitable design

Wire frame diagram – schematic produced displaying where all the agreed features and functions within the website spec will appear

* Cost effective and save time further down the line
* Basic layout composition is planned and agreed to before any complex actual implementation work is undertaken

Design objective – why?

* Create complaint and aesthetically pleasing render of the app wireframe
* Ensure clarity and simplicity at the user interface
* Help new uses quickly master product usage by adding guidance and instruction where needed

Design – methods

* Review and research and conceptual design to understand goals and proposed design
* Skin each wireframe component to comply with corporate guidelines
* Identify and design special components required by the design (icons, special labels, highlight colour, etc)
* Review visuals with ux and documentation

Evaluate – objectives

* Provide constructive and actionable input to specific design questions/issues
* Continuously verify proposed design solutions
* Ensures usability both at the micro and macro level

Evaluate – methods

* User reviews
* Local and remote usability tests
* A/B testing compering design alternatives
* On-line surveys

General UI Design Principles

1. Visibility of system status
2. Match between system and real world
3. User control and freedom (eg, undo and redo)
4. Consistency and standards (in line with users expectations)
5. Error prevention (help avoid mistakes)
6. Recognition rather than recall (rapid and easy learning of system)
7. Flexibility and efficiency of use (eg shortcuts/accelerators for experienced users)
8. Aesthetics and minimalist design (only necessary info)
9. Help users recognise, diagnose and recover from errors (sensible error messages)
10. Help and document

Lab tests and stats

* Number of subjects who can complete the tasks within a certain time
* Length of time required to complete different tasks
* Number of times help functions was needed
* Number of times redo was used and where
* Number of times shortcuts were used

Utility – does the system provide the raw capabilities to allow the user to achieve their goal?

Usability – does the system allow the user to learn and use the raw capabilities easily?

* Learnability
* Efficiency of use
* Error handling
* Acceptability

Usability principles:

1. Do not rely on usability guidelines – always test with users
2. Base UI design on users’ tasks (use case analysis)
3. Ensure sequences of actions to achieve a task are as simple as possible (˅ reading)
4. Ensure user always knows what he or she can and should do next
5. Provide good feedback including effective error messages (inform about progression and location)
6. Ensure the user can always get out, go back, or undo an action
7. Ensure that response time is adequate
8. Use understandable encoding techniques
9. Ensure that the UIs appearance is uncluttered (effective organisation and don’t overload with info)
10. Consider the needs of different groups (locales and people with disabilities, and ensure that the system is usable by both beginners and experts)
11. Provide all necessary help
12. Be consistent

Difficulties and risks

* Users differ widely
* UI implementation technology changes rapidly
* UI design and implementation can often take the majority of work in an app
* Developers often underestimate the weakness of a GUI

GUI frameworks

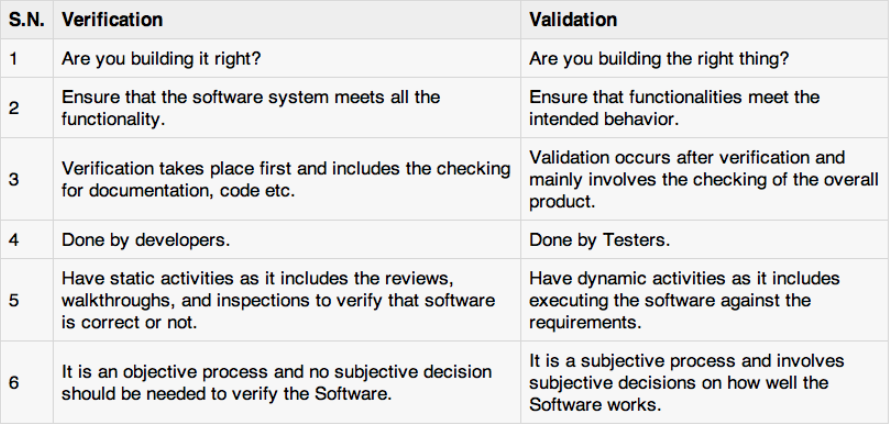
* Java Swing (java)
* Windows Forms (.NET languages, Windows)
* Qt (C++)
* Gtk2 (C/C++)

**[SOFTWARE TESTING]**

Testing – process of evaluating a system/components to check if it satisfies the specific requirements

Eg, executing a program/app with intent of finding software bugs. Involved the identification of bugs/errors/defects in the software without correcting. Done by:

* Software developer (unit testing)
* Software tester
* Project lead
* End user (acceptance testing)



Debugging – identifying, isolating, and fixing problems/bugs. Part of White box/Unit testing

Types of testing:

* Manual: testing without automated tools/scripts
* (Test) automation: writing scripts to test software. Also used for load, performance and stress testing. Used for:
  + Large and critical projects
  + Projects that require the testing the same areas frequently
  + Requirements dont changing much
  + Accessing for applications for load and performance with many virtual users
  + Stable software with respects to manual testing
  + Availability of time

Method of software testing

Black box:

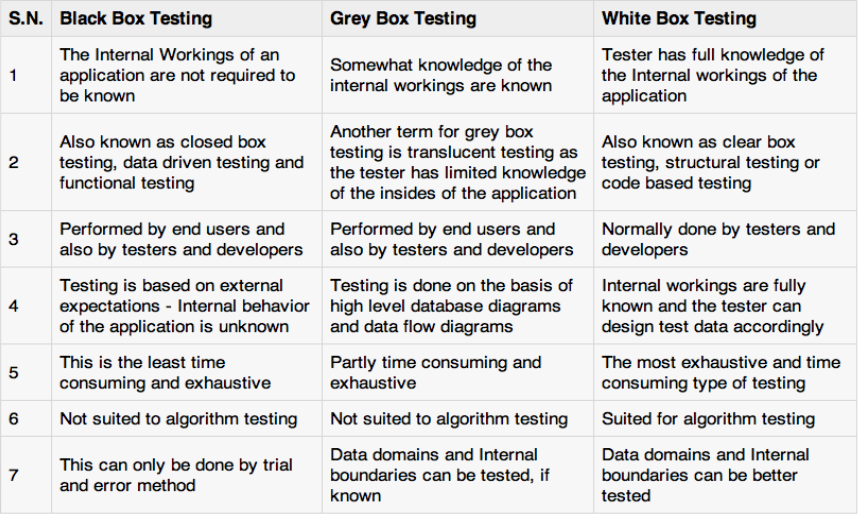
* Testing done without any real knowledge of the interior workings of the app
* Tester will interact with systems UI by providing inputs, and examining outputs, without knowing how and where the inputs are worked upon
* well suited and efficient for large code segments
* code access not required
* clearly separated user’s perspective from the developers perspective through viably defined roles
* large number of moderately skilled testers can test the app with no knowledge of implementation, programming language or operating system
* limited coverage since only a selected number of test scenarios can be performed
* inefficient testing, due to the fact that the tester only has limited knowledge about an application
* Blind coverage, since the tester cannot target specific code segments or error prone areas
* The test cases are difficult to design

White box (glass/open box testing):

* Detailed investigation of internal logic and structure of the code
* Tester must know about the internal workings of the code
* Tester has knowledge of source code, so easy to find out which type of data can help test the app effectively
* Helps to optimise the code
* Extra lines of code can be removed which can bring in hidden defects
* Due to testers knowledge of code, maximum coverage is obtained during test scenario writing
* Due to the fact a skilled tester is needed, its more costly
* Sometimes impossible to look into every nook and corner to find out hidden errors that may create problems as many paths go untested
* Difficult to maintain as use of specialised tools like code analysers and debugging tools are required

Grey box:

* Technique to test the app with limited knowledge of the internal workings of an app
* Offers combined benefits of black box and white box testing where possible
* Don’t rely on source code (interface definition and functional spec)
* Can design excellent test scenarios especially around communication protocols and data type handling
* The test is done from the point of view of the user and not the designer
* Since the access to source code is not available, the ability to go over the code and test coverage is limited
* The tests can be redundant if the software designed has already run a test case
* Testing every possible input stream is unrealistic because it would take an unreasonable amount of time therefore paths with go untested



Functional testing:

* Type of black box testing that’s based on the spec of the software to be tested
* Tested by provided input and then the results are examined that need to conform to the functionality it was intended for
* Conducted on a complete, integrated system to evaluate the systems compliance with its specified requirements

Steps:

1. The determination of the functionality that the intended app is meant to perform
2. The creation of test data based on the spec of app
3. The output based on the test data and spec of the app
4. The writing of test scenarios and the execution of test cases
5. The comparisons of actual and expected results based on executed test cases.

Non-functional testing:

* Performance
  + Load testing
  + Stress testing
* Usability
* Security
* Portability

**Unit** **testing** – performed by developers before the setup is handed over to the testing team. Goal is to isolate each part of the program and show that individual parts are correct in terms of requirements and functionality

**Integration** – testing of combined parts of an app to determine if the function correctly. Bottom-up and top down integration testing

**System** **testing** – next level in the system testing and tests whole system. Performed by a specialised testing team

**Regression** **Testing** – when a change in other places are made, it may affect stuff, so this is to make sure fixing bugs doesn’t break other stuff yo

**User Acceptance testing** – conducted by quality assurance team, to gauge whether the app meets the intended specification and satisfies the clients responsibility. Team will have pre written scenarios. Alpha testing and beta testing

**Documentation** – helps in estimating the testing effort requires, test coverage, requirement tracking/tracing, etc

* Test plan
* Test scenario
* Test case
* Traceability matrix

[insert really boring stuff about what test plans should include but it probably wont come up and im too tired to type it all out]

Test scenario

* One line statement that tells what area in the app will be tested
* Used to ensure all process flows are tested from end to end

Test case

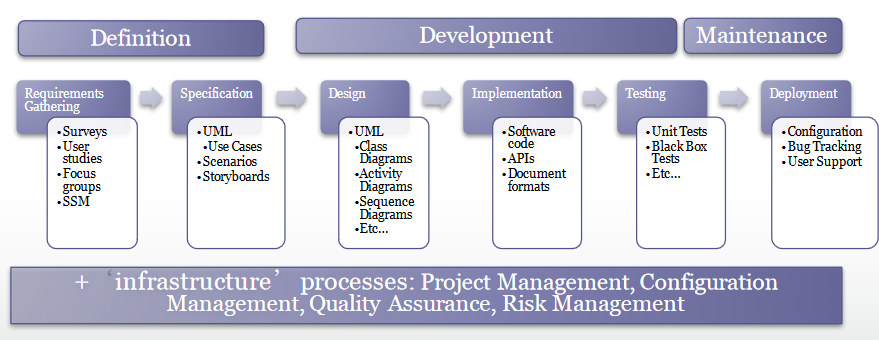
* Involve the set of steps, conditions and inputs which can be used while performing the testing tasks
* Ensure whether the software passes or fails in terms of functionality and other aspects
* Many types of test cases: functional, negative, error, logical test, physical, UI test cases etc

Traceability Matrix – table used to trace the requirements during the software development life cycle. Main goals:

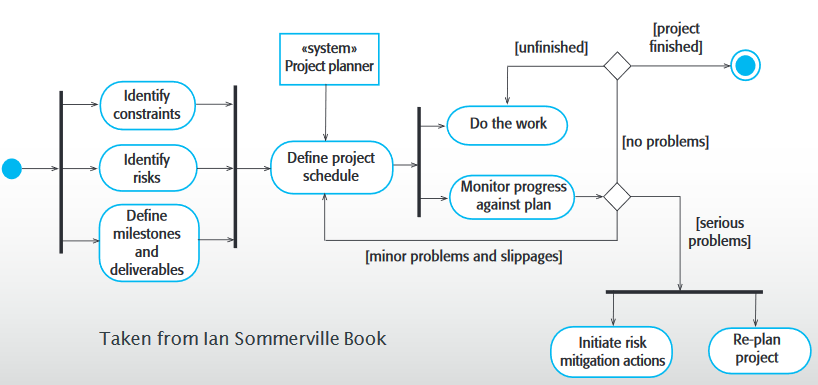
* Make sure software is developed as per the mentioned requirements
* Helps in finding the root cause of bugs
* Helps in tracing the development documents during different phases of SDLC

**[SOFTWARE DEVELOPMENT LIFE CYCLES]**

Software engineering – The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them

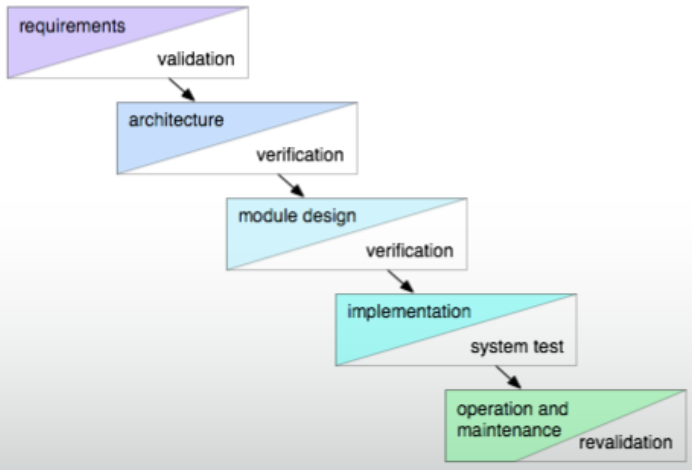
**Process**:

**Process models for software development**

Software process – set of actions required to efficiently transform a user’s need into an effective software solution. Predictable, cost estimates and schedules met with consistency, functional and quality expectations should be met

**Build and fix model** – project is implemented without any spec/design

* Requires less experience to execute or manage
* Suitable for smaller software
* Requires less project planning
* No real means is available of assessing the progress, quality, and risks
* Cost is high, as requires rework until user’s requirements are accomplished
* Informal design of the software as it involves unplanned procedure
* Maintenance of these models is problematic

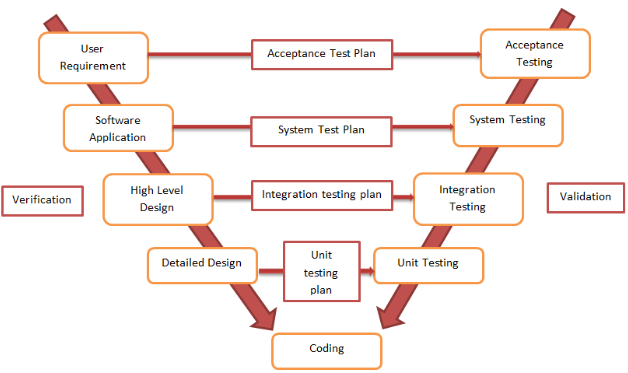
**Simple waterfall model**

* Easy to understand and to use
* Stabilises requirements because no changes are made during development phases
* Allows the organisation to use the staff of one phase in another project when done
* When used wisely, it will produce software with good quality
* Document driven and a huge amount of documentation is developed
* Backtracking cost is high if a problem is found
* False impression about project progress
* Product is only delivered when complete
* All requirements must be known and fully understood early in the life cycle

When to use:

* when staff are lacking experience
* complex (but well-understood) systems
* when quality is more important that cost and schedule

**The V-model** – puts more focus on software testing



* Easy to understand and to use
* The model involves testing early in the life cycle
* Stabilizes requirements because it is developed up front like the waterfall model
* Doesn’t handle requirement changes
* Requirements are tested too late, so the cost of fixing issues related to requirements is high

When to use:

* Suits projects with high reliability needs
* Works well when the requirements are clear and well known

**The prototyping model**

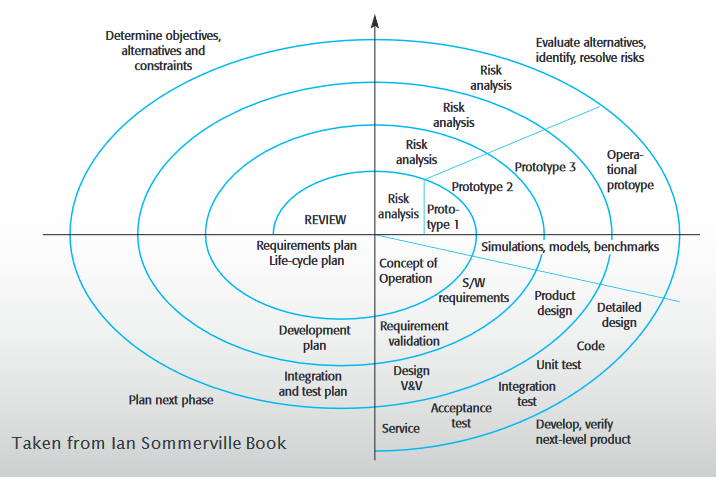
Used for:

* Understanding the requirements for the UI
* Examining feasibility of a proposed design approach
* Exploring system performance issues

Problems:

* Users treat prototype as the solution
* Prototype is only a partial spec

**Spiral Software Process** – introduced as an alternative to the waterfall model and its rigidity in dealing with change of requirements

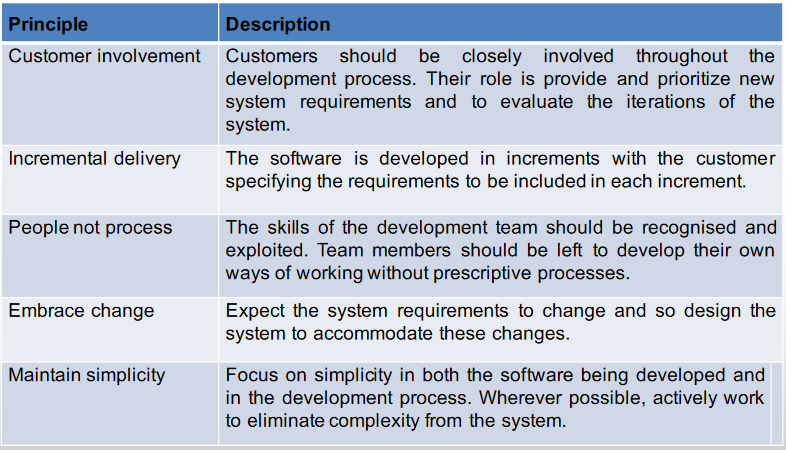
* The spiral model explicitly include risk management and verification and validation
* Allow the user to interact and use the system developed in the earlier phase
* It promotes reuse and increase productivity
* Expensive for small projects or if low risk
* Complex to understand
* Requires good experience in risk assessment
* If not managed correctly the spiral will continue forever due to customer feedback
* Time for planning, risk analysis and resolution and prototyping can be expensive

When to use:

* When prototyping is the appropriate way of development
* When the skills are available to manage and tailor the model as needed
* If the project has medium to high risks (uncertainties, new tech, etc)
* If the project will take a long time (years and is difficult to commit because of the economic changes)
* If the requirements are complex
* Suitable for large projects
* If the project is complex and/or critical (real time/military/health care/space)

**Agile** **methods**

* Dissatisfaction with the overheads involved in software design methods led to the creation of agile methods
  + Focus on code rather than design
  + Based on iterative approach to software development
  + Are intended to deliver working software quickly and evolve this quickly to meet changing requirements
* Aim is to reduce overheads in the software process (eg, by limiting documentation)

Principles of agile methods

Agile vs traditional approaches



Project planning

* Involves breaking down the work into parts and adding these to project team members, anticipate problems that might arise and prepare tentative solutions to problems
* Project plan (created at the start) is used to communicate how the work will be done to the project team and customers and to help assess progress on the project
* Periodically throughout the project you may need to modify your plan in the light of experience gained and information from monitoring the progress of the work

Agile approach (to something)

* Measure progress of small tasks
* Estimate effort for each task
* Only working code counts
* Visualise progress
* Use experience

Burn down chart – plots effort remaining against time