

SOFTWARE ENGINEERING

01/05/15

ATTRIBUTES OF GOOD SOFTWARE: Maintainability, Capability and Security, Efficiency, Acceptability

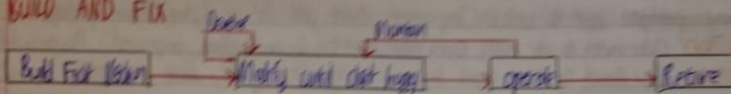
FUNDAMENTAL SFT ENG. ACTIVITIES: Specification, development, validation, evolution

ISSUES AFFECTING SOFTWARE: Heterogeneity, Rapid and Small Change, Security and Fault

HOW LONG EACH ACTIVITY TAKES: Requirements (2%), Specification (5%), Design (15%), Coding (30%), Testing (7%), Integration (12%), Maintenance (39%)

STAGES: Domain Analysis, Requirements, Specification, Architecture, Design, Implementation, Integration, Operation + maintenance

BUILD AND FIX

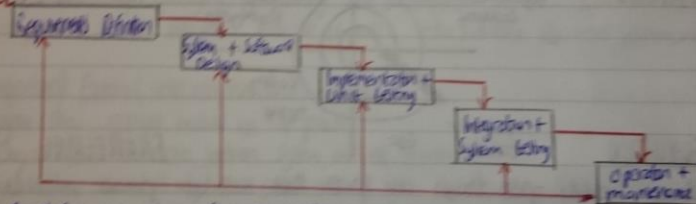


- build anything, then fix it.

ADVANTAGE: Less project planning, Less expensive, Equal, Good for Small Program

DISADVANTAGE: High Cost, Maintenance Problems, Not good for Robust System, No Documentation

WATERFALL



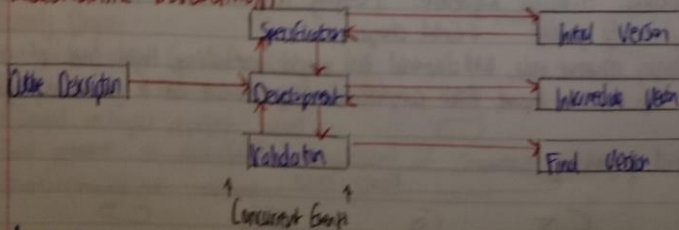
- Don't know exactly what the price after previous phase completion, Faults could null back of stage or more

ADVANTAGE: Easy to manage, Use little resource, Works well for small projects with clear requirements

DISADVANTAGE: Going back a price costly, High Risk and Uncertainty, No working Software until the end.

When To Use: Requirements well known, clear, fixed. Short project. → Objective software, Well-known Product Software

INCREMENTAL DEVELOPMENT



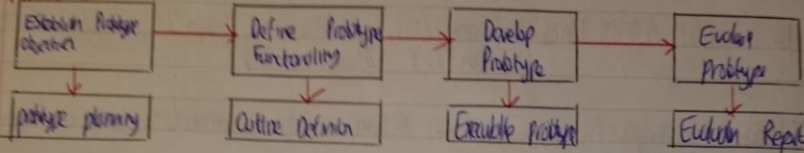
- Agile Method, Develop piece by piece, Each increment adds some of functionality required by client

ADVANTAGES: Iterative software product guide, Flexible-fall costly, Customer can respond to each build
DISADVANTAGES: Need good planning + design, price not viable, Higher costs than other models

When to use: Need to get product to market early, new tech is being used, some high risk features

(RAMP)

PROTOTYPE



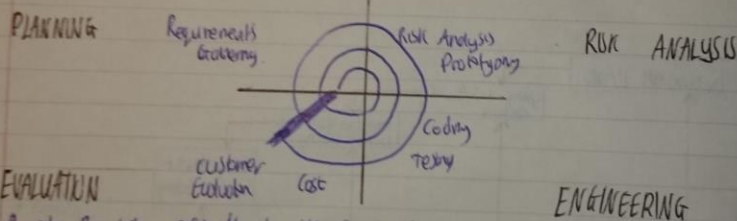
-Throwaway prototype built to understand requirements. Client gets feel of system from prototype

ADVANTAGES: closer match to user needs, Quick Error Detection, Quick Feedback, Improved Maintainability

DISADVANTAGES: Slow Process, Too much Client Involvement, High Skill required, Lack of emphasis on Documentation

When to use: System needs alot of interaction with end users → Online system, Web Interface, Minimal Training

SPIRAL (NOT IN HIS NOTES → MAY NOT BE ASKED??)



EVALUATION

ENGINEERING

-Project repeatedly passes through the spiral. Model built upon previous model(s).

ADVANTAGES: High amount of Risk Analysis, Strong Documentation Control, Highly customisable, Early software produced

DISADVANTAGES: Costly, Requires Specific Expertise, Not easily re-used

When to use: When cost + risk reduction important, High Risk project, Complex Requirements, Significant changes expected

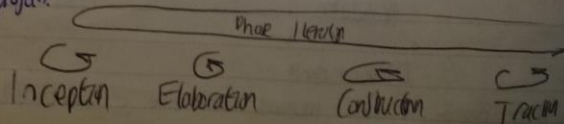
RATIONAL UNIFIED PROCESS (RUP)

- 1-Develop iteratively. 2-Manage Requirement. 3-Employ component-based Architecture 4-Model software visually
- 5-Continuously verify quality 6-Control changes

ADVANTAGES: Changing requirement easy, well documented and complete methodology, Higher level of re-use

DISADVANTAGES: High Expertise Required, Process complex and disorganised, (can not be re-used on high tech projects)

When to use: Small projects



SOFTWARE ENGINEERING

3

15/15

LIFE CYCLES

EXTREME PROGRAMMING (XP)

- 5 steps → Planning, Managing, Designing, Coding, Testing
- Small releases, Simple design, code re-use, paired programming, continuous integration, customer onsite

ADVANTAGES: Fast, Simple, Low Risk, Very Flexible, Robustness - Power of simplicity

DISADVANTAGES: Project plan not explicit, no documentation, Long term effectiveness unproven

When to use: High risk project, Customer Available, small/medium project, Limited planning time

O-O ANALYSIS AND UML

O-O Analysis: Identify objects, methods, relationships. Real World entities represent objects

O-O Design: Decomposing a system into objects

CLASS RESPONSIBILITIES, COLLABORATORS CARDS (CRC)

Goal: Understand the domain of objects

STEPS: Brainstorm candidate objects, create initial CRC cards, Come up with scenarios, Refine CRC cards (role play, scenarios), Why? (identified objects + responsibilities), understand interaction, record of design activity, group session - non tech people

Class	None
Responsibilities	Collaborator

Unified Modelling Language (UML)

- Language for visualizing, specifying, constructing and documenting software systems
- + useful for all phases in software lifecycle. + Re-use of parts - No substitute for real communication

Models (Perspectives): External, interaction, structural, behavioural

Activity Diagram: Show activities involved in a process

Use Case Diagram: Show interaction between system and its environment

Sequence Diagram: Show interaction between objects and the system and between system components

Class Diagram: Show the object classes in system and associations between classes

State Diagram: Show how system reacts to internal and external events

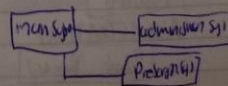
Context Model

- Defines the boundary between the system and its environment, showing the entities that interact with it.
- Show that environment includes several other automated systems
- Do not show type of relationship between systems in environment and system that is being specified

Activity Diagram

- Start process indicated by filled circle
- End process by filled circle
- Rounded rectangles represent actions
- Bars represent the start (split) or end (join) of concurrent activities
- Diamond represent decision

Example



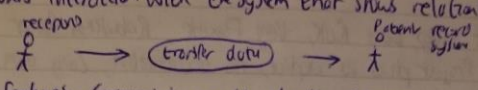
↓ ↓

Interaction Models (UML / Sequence Diagram)

- Model user interaction, System to System interaction, component interaction

Use Case

- Representation of a users interaction with the system that shows relationship between user and different use cases.

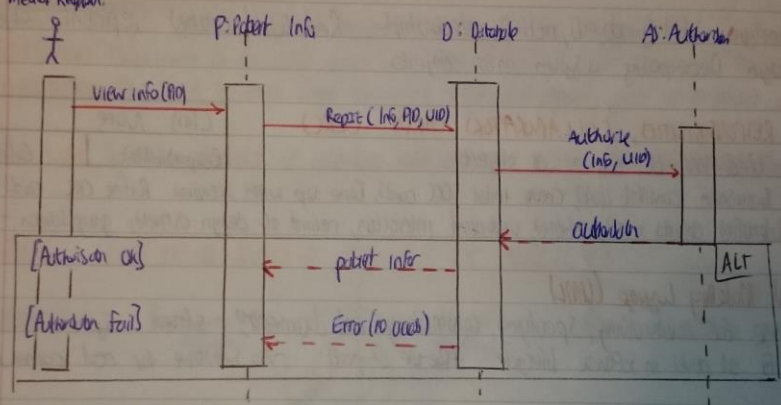


- Helpful: Determining features, Communicating with clients, Generating test case.

Sequence Diagram

- Shows how processes operate with one another - Interaction between actual and objects in system.

Medet Request



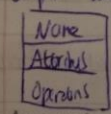
- Actors + objects listed along top. Lifelines are arrows.
- Alt used with condition in square bracket.

STRUCTURAL MODELS

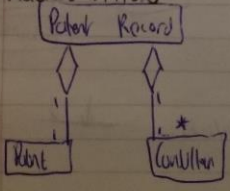
- Display organization of system in terms of components that make up system and their relationship.

Class Diagram

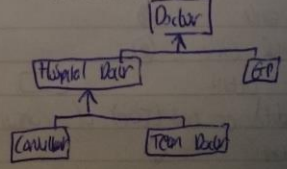
- Overview of system showing its classes and relationship among them. (Class diagrams are static).
- Simple way: Patient * --- Patient Record 1:1 relationship * for indefinite



AGGREGATION



GENERALIZATION



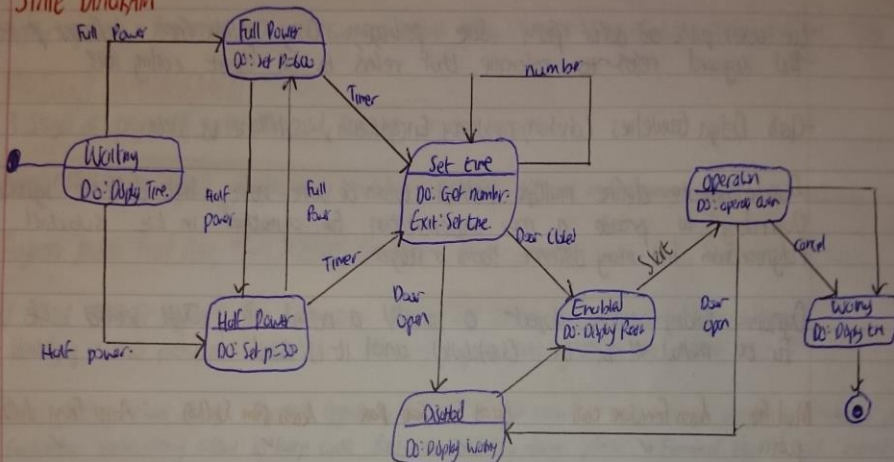
SOFTWARE ENGINEERING

UML

Behavioural Models -

- Model of dynamic behaviour of system as it's executing. Data and events
- Data Driven - Shows flow of data through system - Not supported by UML.
- Event Driven - Shows system response to external and internal events

STATE DIAGRAM



DOMAIN ANALYSIS

O-O Approach why? Solution early modified, simpler solution, improved Readability, Reusability

Domain Analysis: Identification, analysis, and specification of common, reusable capabilities within specific application domain

Domain Well defined set of characteristics that accurately, normally and completely describe a family of problems for which computer application solution are being sought

OO analysis focused on the features and functionality of a single system to be generated, domain analysis focuses on the common and various factors across a family of systems

OBJECT ORIENTED PROGRAMMING OO, INHERITANCE

Object: consist of state and related behaviour

Benefit of Objects: Modularity: code write and maintain independently. Information hiding
Code reuse, Pluggability and debugging ease

Class Name	Constructors:	ClassName (parameterName: parameterType)
Data Fields:	Data Fields:	dataName: dataType
Constructors and Method	Method:	methodName (parameterName: parameterType): returnType
	+ sign	indicates public
	- sign	indicates private
	~ sign	indicates static

- cannot perform action on variable which is not initialized
- Instance variable and methods can be used only from instance method, not from static method because static variables and methods don't belong to a particular object.
- To prevent direct modifications of data field, should declare the data field private using the keyword known as data field encapsulation or getters and setters
- Can access public and default from same package. Only public from different packages
- ~~The~~ keyword refers to reference that refers to an object calling itself.
- Class Design Guidelines: Cohesion, consistency, Encapsulation, Instance vs. Static

Overloading: to define multiple methods with the same name but different signature
 Overriding: to provide a new implementation for a method in the subclass
 Polymorphism: how many different forms or stages

Dynamic Binding - when object o invokes a method m, JVM searches the implementation for the method m in the class/classes until it is found

Modifier	Access from same class	Access from same package	Access from subclass	Access from different package
public	✓	✓	✓	✓
protected	✓	✓	✓	
(default)	✓	✓	✗	✗
private	✓	✗	✗	✗

REQUIREMENTS

Process of finding out, analyzing, documenting and checking these services and constraints - Requirements engineering

Functional and non-functional requirements

Functional - search ability, login ability etc.

Completeness and Consistency

Non-Functional - performance, security, availability

Main elements in Non-Functional Requirements: Product, Organizational and external requirements

Approaches for Specification: Informal, Structural, Formal, Semi-Formal

Requirements Validation: Consistency, Completeness, Realism and Verifiability

DEBUGGING AND TESTING

- Test for Specification against users' requirements - validation, verification
- Include hypothesis for test.
- Systematic process
- Correctness, reliability, robustness, performance, usability, utility
- Execution and non-execution method
- Large Sub, Small Sub or walk through code

SOFTWARE ENGINEERING

7

2/05/2015

White box Testing

Test a module using the knowledge of its internal

- Try to trip algorithm, allows more detailed test code

Unit testing Integration testing

Black box

- test a module using only the knowledge in its documentation

Acceptance and system testing

System engineering \rightarrow Requirements \rightarrow Design \rightarrow Construction \rightarrow Unit testing \rightarrow Integration testing \rightarrow Validation testing \rightarrow System testing

PROJECT MANAGEMENT

3 stages of project life cycle: Proposal, Set-up, Monitor project

Costs: Effort costs (labor), hardware including maintenance, travel and training costs

Payment models: Fixed Price, Time/Materials, Cost-Plus, Fixed price per unit.

4 Critical tasks for supplier: Time Estimation. Planning and scheduling within contractual constraints. Identifying critical paths, points and risks. Monitoring projects - detecting and correcting problems

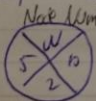
Factors Affecting software Prices: Market opportunity - low price to get started. Cost estimate uncertainty - hike up price. Contractual terms - may allow to keep code. Requirement volatility - lower price. Financial health - get paid!

Plan normally covers: Introduction, Project Organization, Risk Analysis, Hardware/software requirements, Work breakdown, Project schedule, Monitoring and reporting mechanisms

Project Scheduling: Bar Charts or Gantt Charts, Activity Networks

Project Activities: Duration, Effort Estimate, Deadline. A defined endpoint

Earliest start time



Latest start time

Path with the least slack time is the critical path

ESTIMATION TECHNIQUES

- Parkinson's law - "work expands so as to fill the time available for its completion."
- Brooks's law - "Adding manpower to a late project makes it later." "9 women can't make a baby in a month."
- Hofstadter's law - "It always takes longer than you expect, even when you take into account Hofstadter's law."

Major software cost estimation techniques (Bottom): Algorithmic models. Expert judgement. Analogies.
Punishment - whatever is available. Price-bid-win. Top-Down. Bottom up.

Constructive cost model - COCOMO

Function Points: 1. Count number of inputs/outputs/structures 2. Classify in terms of their coverage complexity
3. Multiply by the factors to get total number of FPs. 4. Multiply by programming language to get the cost.

COCOMO Applies to: Organic (Small team, good exp.) Semi-detached (medium team, mixed exp.) Embedded-high constraint

Basic COCOMO

- Computes software development effort and cost as a function of program size.
- Good for quick estimate of cost. - Doesn't account for hardware constraints, experience, use of modern tools...

Intermediate COCOMO's

- Computes software development effort as a function of program size and a set of "cost drivers" that include subjective assessments of product, hardware, personnel and project attributes.
- Project Attributes: Product, Hardware, Personnel and project attributes

Detailed COCOMO

- Incorporates all characteristics of the intermediate versions with an assessment of the cost drivers' impact on each step (analysis, design etc.) of the software engineering process.
- 5 phases each with specific calls:
 - Plan and requirements
 - System Design
 - Detailed Design
 - Module code and Test
 - Integration and test