SOFTWARE ENGINEERING

9/04/15

OVERVIEW

Software engineering is an engineering discipline that is concerned with all aspects of software production

Attributes of good software:
- Maintainability
- Dependability and security
- Efficiency
- Acceptability

Software Process Activities
- Software specification
- Software development
- Software validation
- Software evolution

Issues affecting software:
- Heterogeneity - systems having to work on different platforms
- Business and social change - evolving and updating
- Security and trust

Examples of Software Engineering Diversity:
- Stand alone applications - Systems running on local computer
- Interactive transaction-based applications - execute on remote computers, accessed by PC
- Embedded Control Systems - Control and manage hardware devices
- Batch processing Systems - Business Systems designed to process data in large batches
- Entertainment Systems
- Systems for modelling and simulation
- Data collection Systems
- System of Systems

What is involved?

Requirements	2%
Specification	5%
Design	6%
Coding	5%
Testing	7%
Integration	8%
Maintenance	67%

Planning
Cost
Expertise
Time Software Packs
Documentation
Maintenance

SOFTWARE ENGINEERING

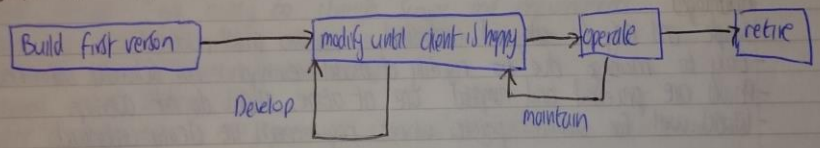
9/04/15

LIFECYCLES

Stages:
Domain Analysis - wider business context of system
Requirements - what client wants
Specification - what client will get
Architecture - Full system context
Design - Component, Structure, Interface
Implementation - Realise each element of design, Unit testing
Integration - Verification and validation (acceptance testing)
Operation and maintenance - Bug detection and fix, new Features, users etc

Software Process Category:
Plan Driven - Process activities are planned in advance
Agile - Planning is incremental.

Build and Fix

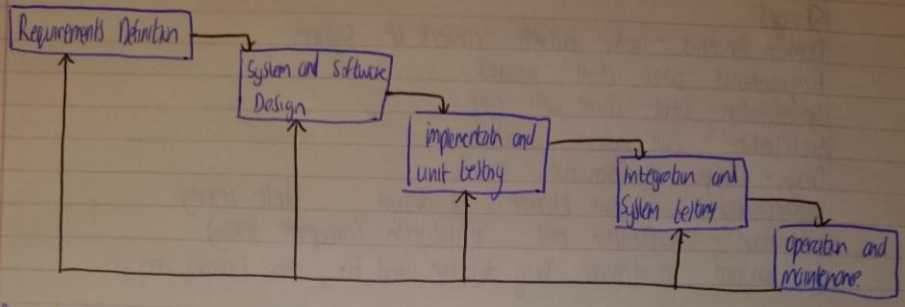


- Build anything, then Fix it.
- May work for small projects, not big one
- No real game plan.
- Deficiences propogate
- No Documents = no maintenance

Advantages
- Require less experience to execute of manage other than ability to program
- Suitable for smaller software
- Require less project planning

Disadvantages
- No real means is available of assessing the progress, quality and risks.
- Cost is high as required rework until user spec met
- Informal Design of software as it involves unplanned procedure
- Maintenance of these models problematic

Waterfall Model:



- Plan Driven process - plan and schedule work
- Following phase should not begin before completion of previous
- Each stage gets verified before proves proceeds
- Faults cause process to roll back a stage or more

Advantages:

- Simple and easy to use and understand, requires small resources
- Easy to manage due to rigidity of model - each phase has deliverables and review process
- Phases are processed and completed one at a time. Phases do not overlap.
- Works well for smaller projects where requirements are clearly understood

Disadvantages:

- Once in testing stage, had to go back and change something poorly thought of in concept stage
- No working software is produced until late during the life cycle
- High amounts of risk and uncertainty
- Not a good model for complex and object oriented projects
- Poor model for long and ongoing models
- Going back a phase costly
- Schedule not adhered to

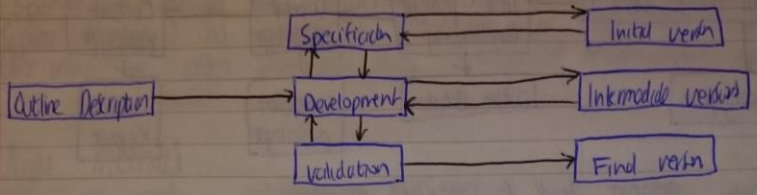
When To Use:

- Requirements are well known, clear and fixed
- Product definition is stable
- Technology is understood
- No ambiguous requirements
- Project is short
- Database software, Development of network protocol software

SOFTWARE ENGINEERING

LIFECYCLES

Incremental Development:



- Develop piece by piece
- Agile approach
- Each increment or version of system incorporates some of functionality needed by the customer.

Advantages:

- General working software quick and early during software life cycle
- More flexible - less costly to change scope and requirements.
- Easier to test and debug during a smaller iteration
- Customer can respond to each build
- Lowers initial delivery cost
- Easier to manage risk as risky pieces are identified and handled at each iteration

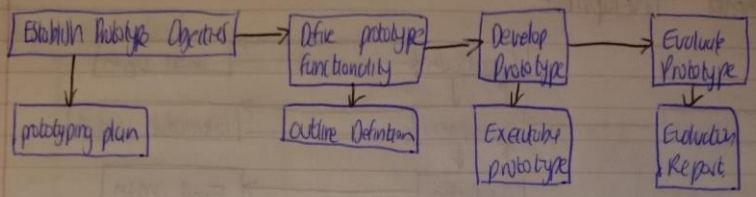
Disadvantages:

- Needs good planning and design.
- Needs clear and complete definition of whole system before it can be broken down and built incrementally.
- Total cost higher than waterfall. Process not visible

When To Use:

- When requirements of complete system are clearly defined and understood
- Need to get product to market early.
- A new technology is being used
- Resources with needed skill set are not available
- There are some high risk features and goals

PROTOTYPE



- Throwaway prototype built to understand requirements
- Client gets feel of system from prototype
- Prototype - An initial version of a system used to demonstrate concepts and try out design options

Advantages:

- Closer match to users' need
- Errors detected much earlier
- Quicker user feedback is available leading to better solutions
- Missing functionality can be easily identified
- Improved design quality
- Improved maintainability

Disadvantages:

- Slow process - Difficult to manage and schedule
- Possibly too much client involvement
- May increase complexity of system as scope of system may expand beyond original plans
- Incomplete or inadequate problem analysis
- Prototype structure usually degraded through rapid change

When To Use:

- When desired system needs to have a lot of interaction with end users
- Online systems, web interfaces, ease of use, minimal training
- Good human computer interface systems

SOFTWARE ENGINEERING

09/04/14

LIFECYCLES

Rapid Software Development:
- Specification, design and implementation are interleaved (no detailed system spec)
- System Developed as a series of prototypes with stakeholders included in version evaluations
- User interface are often developed using an integrated development environment IDE and graphical toolkit

Agile Methods
- Focus on code rather than the design
- Based on an iterative approach to software development.
- Intended to deliver working software quickly and evolve this quickly to meet challenging requirements
- Results in small incremental release with each release building on previous functionality

Principles:
- Individuals and interactions over process(s) and tools
- Working Software over comprehensive Documentation
- Customer Collaboration over contract negotiation
- Responding to change over following a plan

Advantages
- Customer satisfaction by rapid, continuous delivery of useful software
- People and interactions are emphasised rather than process and tools
- Customers, developers and testers constantly interact
- Continuous attention to technical excellence and good design.
- Regular adaptation to changing circumstances
- Even late changes in requirements are welcomed.

Disadvantages:
- Difficult to assess the effort required at the beginning of software development life cycle
- Lack of emphasis on necessary designing and documentation.
- Project can go off track if client not clear what final outcome they want.
- Only senior programmers are capable of taking the kind of decisions required during the development process, hence it has no place for newbie programmers, unless combined with experienced resource
- Maintaining simplicity required extra work

When To Use:
- Changes are needed to be implemented. New changes can be implemented at very little cost because of frequency of new increments that are produced
- Implement a new feature, developers need to lose only the work of a few days, or hours to roll back and implement
- Very limited planning required to get project started
- Quick time to market.

Extrene Programming.
- Incremental planning
- Small releases
- Simple Design
- test first development
- Refactoring - code re use
- Pair programming - working in pairs
- Collective ownership - everyone works in different parts, switch around
- Continuous integration
- Sustainable pace
- On Site customer.
- User requirements expressed as Scenarios or Stories → broken down into Thele tasks are basis of schedule and cost estimates

Planning, Managing, Designing, Coding, Testing

Planning:
- User Stories
- Release planning creates the release schedule
- Make frequent small releases -
- Projects divided into iterations

Managing
- Open work space → Communication
- Set sustainable pace
- Stand up meeting at start of day
- Move people around
- Fix XP when it breaks

Designing
- Simplicity - TUBE Testable, understandable, Browseable, Explainable
- Use class responsibility and collaboration cards to design system as a team

Coding
- Customer is always available
- Coding standard
- Code the unit test first
- Majority of production code is paired programmed
- Only one pair integrate code at a time
- Integrate often
- Use collective code ownership

SOFTWARE ENGINEERING

09/04/15. LIFECYCLES

Extreme Programming Continued:

Testing:
- All code must have unit tests
- All code must pass all unit tests
- Acceptance tests are run often and score is published

Advantages:
- Robustness - power of simplicity
- Resilience
- Cost saving
- Lesser Risks
- Very flexible
- Focused on developing the right system

Disadvantages:
- Assumes constant involvement of customer with technical expertise
- Project phases not explicit
- Suitable for "Smallish" projects
- No requirement to produce documentation (maintenance?)
- Long term effectiveness still unproven

When To USE AGILE:

- Product development where a software company is developing a small or medium sized product for sale
- Custom system development within an organisation, where there is a clear commitment from the customer to become involved in the development process and where there are not alot of external rules and regulations that affect the software
- Because of their focus on small, tightly-integrated teams, there are problems in scaling agile methods to large systems
- Very limited planning time required to get project started.

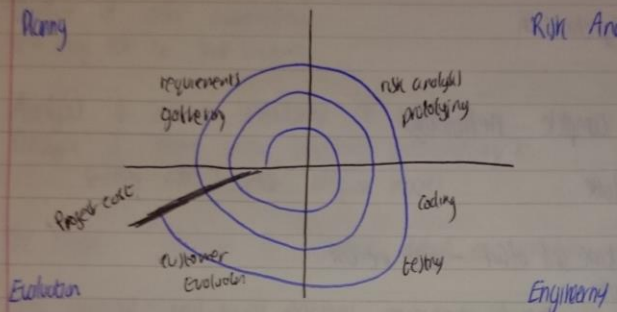
SOFTWARE ENGINEERING

14/04/16

LIFECYCLES

Spiral:

- Spiral model similar to incremental model, with more emphasis on risk analysis
- 4 phases: Planning, Risk Analysis, Engineering and Evaluation
- A software project repeatedly passes through these phases in iterations (spirals)
- Baseline spiral, starting in planning phase, requirements are gathered and risk is assessed
- Subsequent spiral build on baseline spiral



Advantages:

- High amount of risk analysis; avoidance of risk maximised
- Good for large and mission driven projects
- Strong approval and documentation control
- Additional functionality can be added at a later date
- Software is produced early in software life cycle
- Highly customizable

Disadvantages:

- Can be costly
- Highly customized ⇒ not easily reused
- Risk analysis required highly specific expertise
- Project's success highly dependant on the risk analysis phase
- Doesn't work well for smaller projects
- Not suitable for low risk projects

When To Use:

- When cost and risk evaluation important
- For medium to high risk projects
- Long term project commitment unwise because of potential changes to economic priorities
- Users are unsure of their needs
- Requirements are complex
- New product line
- Significant changes are expected (research and exploration)

Rational Unified Process

Best practices:
- Develop Iteratively - keep evolving
- Manage Requirements - documentation
- Use components
- Model Visually - UML Diagrams
- Verify Quality
- Control change - Synchronization

Advantages:
- Well documented and complete methodology
- Open and Published
- Training readily available
- Changing Requirements
- Reduced integration time and effort - code re use
- Higher level of reuse

Disadvantages
- Team members need to be experts in their field to develop software using this method
- Developmental process is too complex and disorganized
- On cutting edge projects which use new technology, the reuse of components will not be possible Hence the time saving ~~costs~~ are not possible
- Integration throughout project, on big project ends in confusion and mac use

SOFTWARE ENGINEERING

10/04/15 O-O Analysis and UML

O-O Analysis
Identify the fundamental objects, methods and relationships in Pac Man

OBJECTS: pac man, ghost, dot, board, game
METHODS: move, change direction, eat, die, finish board.
RELATIONSHIPS: pac-man dots, pacman-ghost (may change), pacman-cherry, board-game

Why not straight to design:
- Danger of over committing
- O-O may not be best solution

Analysis is about understanding the world
Design is more about simulating and manipulating it
Seeking the simplest adequate model

The model abstracts the essential details of underlying problem from its usually complex real world

O-O analysis - real world entities represent objects
O-O Design - decomposing a system into objects

Class Responsibilities, Collaborators Card) (CRC) Card.

Class Name	
Responsibilities	Collaborators
-	-
-	-
-	-
-	-

Goal: understand the domain as objects
- Do analysis is language independent
- Force developer to "think" in objects

STEPS: Brainstorm candidate classes
- Create initial class-responsibility-collaborator cards
- Come up with scenarios of use in the domain
- Use scenario and role playing to refine CRC cards

Why CRC?
- help identify objects and their responsibilities
- help understand how objects interact.
- Cards form a useful record of design actions
- Work well in group situations and are understandable by non-technical stakeholders

UML

Unified Modelling Language (UML)
- General purpose modelling language designed to provide a standard way to visualize the design of a system
- FOR: Specifying, visualizing, constructing, documenting, business modelling, communication

+ Useful for all phases in the Software lifecycle
+ Re use of some of the phases eg. the test cases
- no substitute for real communication between team members

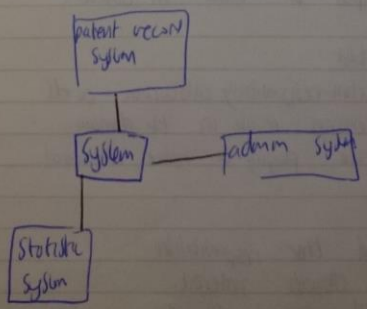
System Modelling
→ External: perspective where you model the context or environment of a system
→ Interaction: perspective where you model interactions between a system and its environment or between the components of a system
→ Structural: where you model the organization of a system or structure of data processed by system
→ Behavioural: model the dynamic behaviour of the system and how it responds to events

UML Diagrams
Activity Diagrams: show the activities involved in a process or in data processing
Use case Diagrams: Show the interactions between a system and its environment
Sequence Diagrams: Show interaction between actors and the system and between system components
Class Diagrams: show the object classes in the system and associations between these classes
State Diagram: Show how the system reacts to internal and external events

- means of facilitating discussion
- Documenting an existing system
- As a system description that can be used to generate a system implementation

CONTEXT MODELS

- At early stage of spec, should decide on system boundaries
- work with system stakeholders to decide on functionality
- Definition of context and dependencies that system has on its environment

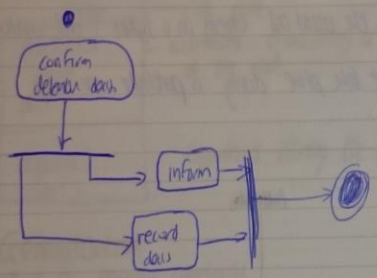


SOFTWARE ENGINEERING.

10/04/15 O-O ANALYSIS AND UML

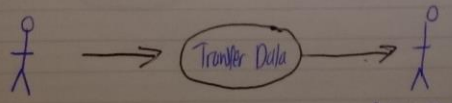
Advantages/Disadvantage of Context Model:
- Normally show that the environment included several other automated system
- Do not show types of relationship between the systems in environment and system being specified

ACTIVITY DIAGRAM
- Show the activities that make up a system process and flows of one activity to another.
→ Start process indicated by a filled circle
→ End is circle inside circle
→ Rectangle with round corners are activities (a specific sub process)
→ Arrows are flow between Activity
→ A solid bar is used to indicate activity coordination
→ When flow of one activity leads to a solid bar, then all of the activities must be completed before process ahead
→ When flow from solid bar leads to a number of activities, they may be executed in parallel



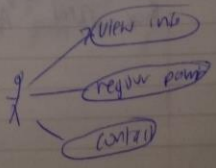
Interaction Models
- Help identify user requirements
- Highlight communication problems which may arise
- Helps understand if proposed system structure is likely to deliver required system performance and dependability

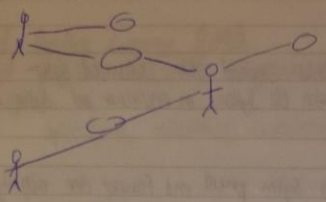
USE CASE
- Is a requirement discovery technique
- Identifies the actors involved in an interaction and names of type of interaction
- Documented using a high level use cases:
  - Actors in process
  - Each class of interaction represented as a named ellipse
  - Lines link the actors with the interaction



Medical Receptionist

Patient Record System



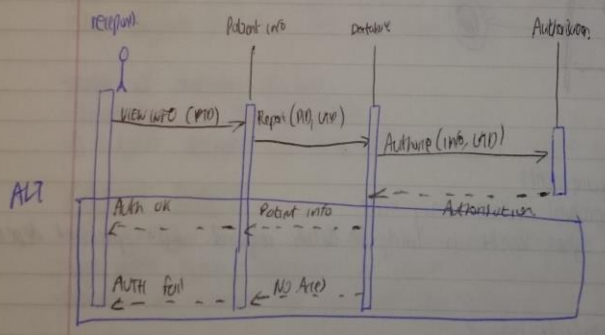


Offer a simple overview

USE CASES
- Determining features
- Communicating with clients
- Generating test cases

SEQUENCE DIAGRAMS
- Used to model interaction between the actor and objects in a system and interaction between objects themselves
- Shows the sequence of interaction that take place during a particular use case

Example:



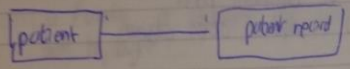
Notation
- Read from top to bottom
- Box named Alt used with conditions

INFORMATION SYSTEMS

OO ANALYSIS And UML

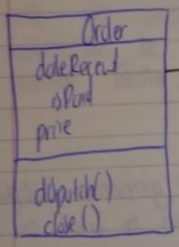
Structural Models
- Display the organization of a system in terms of components that make up that system and their relationships
- May be static or dynamic models

Class Diagram
- Gives overview by showing its classes and relationships among them
- Static - display what interacts but not what happens when they interact.
- In UML can be expressed at different levels of detail
- Used for object oriented system - show classes in system and associations between classes.



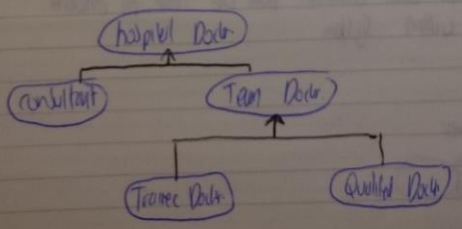
* indicates multiple number of objects

Example:

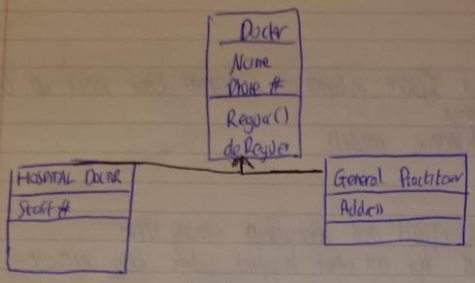


- class name
- attributes (characteristics of members of the class)
- Operations offered by the object.

Generalisation
- Used to manage complexity.
- Rather than learn the detailed characteristics of every entity, we place these entities in more general classes and learn characteristics of these classes
- Useful to examine classes in a system to see if there is scope for generalisation
- means that common info maintained in one place only.
- easy to make changes → all in one place
- Implemented using inheritance mechanism



Generalisation shown as an arrowhead pointing up to the more general class



lower level are sub classes inherit the attributes and operations from their superclass

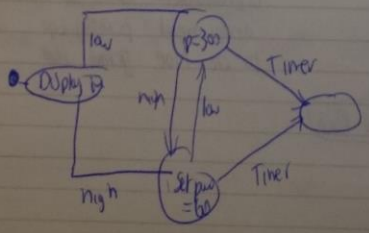
Aggregation:
- Objects in real world often composed of different parts
- One object (represented with a diamond) is composed of other objects

Behavioural Models
- Model dynamic behaviour of system as it executes
- Happens when system respond to stimuli
  - Data has to be processed by system
  - Event

Data Driven Modelling
- Show sequence of actions involved in processing input data and generating output.
- Particularly useful during analysis of requirements as they can be used to show end to end processing in a system

Event Driven Modelling
- Shows how system responds to external and internal events
- Based on assumption that a system has a finite number of states, events cause transition
- Useful for real time systems
- Supported by UML as state Diagrams

STATE DIAGRAM
- Show system states and events that cause transition from one state to another.
- Do not show flow of data within system



SOFTWARE ENGINEERING

10/04/15 DOMAIN ANALYSIS

Why O-O approach?
- Solution easy modified / and or extended
- Simpler solution
- Improved readability
- REUSABILITY

Domain Analysis - The identification, analysis, and specification of common, reusable capabilities within a specific application domain
Key method for reaching systematic software reuse

Domain - A well defined set of characteristics that accurately, normally and completely describe a family of problems for which computer application solutions are being and will be sought

While OO analysis is focused on the features and functionality of a single system to be generated, a domain analysis focus on the common and variant feature across a family of system

SOFTWARE ENGINEERING

11/04/15

PROJECT MANAGEMENT

Proget planning:
- Proposal Stage when bidding for contract. Have enough resources? Price?
- Startup phase: who works on proj, how to break project down.
- Periodically through proj - when plan is changed/modified

Estimating Costs:
- Effort costs - cost of paying software engineers and managers
- Hardware and software cost including maintenance
- Travel and training costs

Payment Method
- Fixed price
- Time and materials - agreed price per person/month
- Cost plus - what it costs to build plus a possible bonus
- Fixed price per unit - agreed price for each part of a multi part delivery

Contract:
- customer - know what you're getting
- Supplier - know what building, deadline, budget

4 Critical Tasks:
- Estimate how long it will take
- Planning and Scheduling: Make things within contracted constraints
- Identifying critical paths, points and risks to be watched carefully
- Monitoring how things progress, detecting and correcting problems

Software Pricing:
- Market opportunity - low price to break into market
- Cost estimate uncertainty - if unsure of price may raise it above normal profit
- Contractual terms - may allow ownership of code and re-use
- Requirements Volatility - If requirements likely to change, → lower price
- Financial Health - may lower price to gain contract. Cash flow more important than profit.

Plan Based Development
- Development process is planned in detail
- Managers use the plan to support project decision making and as a way of measuring progress

+ Early planning allows org issues (staff) to be taken into account and potential problems and dependencies are discovered before project starts

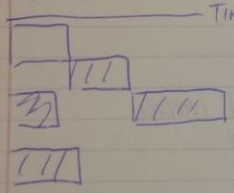
- many early decisions have to be revised because of changes to the environment in which software is being developed

Project Plans:
- Introduction - Objectives and constraints
- Project Organisation - Development team, people and roles
- Risk Analysis - Risks, likelihood and risk reduction strategy
- Hardware and Software Requirements
- Work Breakdown - Break into activities, milestones and deliverables for each activity
- Project Schedule - Dependencies, estimated time
- Monitoring and Reporting Mechanisms

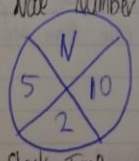
Project Scheduling
- How work in a project will be organised into separate tasks, when and how tasks executed
- Calendar time to complete each task, effort required
- Resources needed for each task, storage, load etc.

Project Activities
- Duration in days
- Effort estimate, number of people days
- Deadline
- Defined endpoint

Gantt Chart



Critical Path
- Network model of activities indicating the scheduling constraints and durations



SOFTWARE ENGINEERING

11/04/15. ESTIMATION TECHNIQUES

Parkinsons law 1995 - "work expands so as to fill the time available for its completion"

Brooks law 1975 - "Adding manpower to a late Software project makes it later" "9 women can't make a baby in one month"

Hofstadter's law 1979 "It always takes longer than you expect, even when you take into account Hofstadter's law"

Major Software Cost Estimation Techniques (According to Boehm)
- Algorithmic - Function calculated cost from list of variables
- Expert Judgement -
- Analogy: Comparing current with similar past project
- Parkinson - project costs whatever is available
- Price-to-win - price necessary to win contract
- Top down - total cost calculated then split into more detailed breakdown
- Bottom up - cost for each component calculated then totaled

Experience Based Techniques:
- Rely on managers experience of past projects and actual effort expended in their projects
- Limitation is that a new software may not have much in common with previous projects

Formula-Based:
- Use a mathematical formula to predict costs based on estimates of project size, type of software being developed, and other team, process and product factors

FUNCTION POINTS
- Count number of inputs, outputs, sources
- Classify them in terms of their average complexities
- Multiply by the factors and sum to get total number of FP's
- Multiply by a programming language specific number to get KLOC Lines of Source Code

Constructive Cost Model COCOMO
- BASIC - computes software development effort and cost as a function of program size
- INTERMEDIATE - computes software development effort as function of program size and a set of "cost drivers" that include subjective assessments of product hardware, personnel and project attributes
- DETAILED - incorporates all characteristics of intermediate with an assessment of the cost driver's impact on each step (analysis, design, etc) of the process

Applied to three classes of Software projects
ORGANIC - small team with good experience working with less than rigid requirements
SEMI-DETACHED - medium teams with mixed experience working with mix of rigid and non rigid requirements
EMBEDDED - developed with a set of tight constraints

Computing Basic COCOMO

Effort Applied (E) = $a_b(KLOC)^{b_b}$ [person-months]

Development Time (D) = c_b (Effort Applied)^{d_b} [months]

People required (P) = E/D [count]

Where KLOC is estimated number of deliverable lines (thousands)
a_b b_b c_b d_b all values given in a table

+ Good for quick estimate of Software cost.
- Does not account for difference in hardware constraints, personnel quality and experience, use of modern tools and techniques and more

INTERMEDIATE COCOMO is
Attributes used in estimation:
- Product - required Software reliability, size of application database, complexity of product
- Hardware - run time performance and memory constraints
- Personnel - analyst capability, application experience, programming exp
- Project Attributes - use of Software tools, required development schedule

EAF (effort adjustment factor) given in table

DETAILED COCOMO
5 phases:
- Plan and requirement
- System Design
- Detailed design
- Module code and test
- Integration and test

SOFTWARE ENGINEERING

WEEK 6 - REQUIREMENTS AND SPECIFICATION

Description of what system should do

Requirement Engineering process of finding out, analysing, documenting and checking these services and constraints

Functional requirements
Non Functional Requirements - performance, security, availability

Completeness - need that all services required by user should be defined
Consistency - requirements should not have contradictory definitions.

Main classes in non functional requirements:
Product - Specify or constrain behaviour of software.
Organisational - derived from policies or procedures in cultural org
External - regulatory requirements

Speed, Size, ease of use, reliability, robustness

Validation
Consistency, Completeness, realism, verifiability

SOFTWARE ENGINEERING

11/04/15. DEBUGGING

Testing - find flaw
Debugging - Locating cause of flaw and eliminating it.

What to look for?
- Validation - are we building the right product?
- Verification - are we building product right?
- Should involve a hypothesis
- Needs to be systematic
- Correctness, reliability, robust, performance, usability, utility

Strategy:
- Large scale - can it handle 200 users?
- Small - does each function for all parameters
- Walk through code

Black box testing:
Test a module using only the knowledge in its documentation

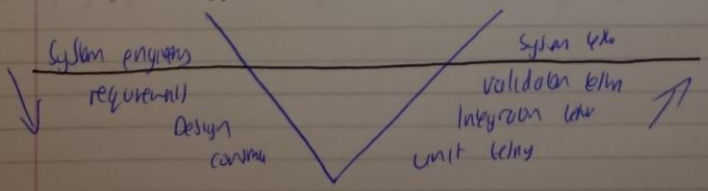
"This function determines average price of goods given these queries"

Reasonable 12.45, 12.46, 1356
Unlikely : 12.45, 12.46 123987.12
Almost incorrect : 12.45, 12.46, -13.42

White box testing
- Test a module using a knowledge of its internals
- try to trip up algorithm

- Good when testing features that may be deliberately altered, if you can't break white box, breaking them black box is still likely

Overall Strategy



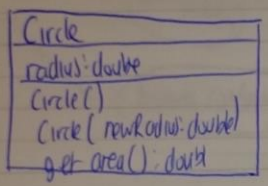
System engineering - middle part of larger system.
Read: entire system, higher than software engineer.

SOFTWARE ENGINEERING

11/9/15 INHERETENCE, UML

Benefits of Use of Software objects
Modularity - can be written and maintained independly of source code of other objects
once created, can easily be passed around System
Information hiding - by interacting only with objects methods, details of internal implementation remain hidden
Code re-use -
Plug-ability and debugging ease

UML Class Diagram



class name
data field
constructor and method

Constructor denoted as ClassName(parameterName: parameterType)
Method denoted as: methodName(parameterName: parameterType): returnType
+ Sign indicates public
- Sign indicates private
underline indicates static use ↑ to indicate superclass

Class Design
Cohesion - should describe a single entity
Consistency
Encapsulation
Instance v. Static

Overloading means to define multiple methods with same name but different signature
Overriding - provide a new implementation for a method in subclass

polymorphism - when object can have many shapes