**Midterm 1 Review**

1. Software Engineering: A Gentle Introduction
2. Software Process
3. Requirement Engineering
4. Agile Software Development
5. System Modeling
6. Software Design & Architecture

**Lecture 1: A Gentle Introduction**

**Software** - a set of instructions or programs instructing a computer to do specific tasks.

Many categorizations → real-time software, mission-critical, system, application, generic, custom

**System Software vs Application**

System - sometimes called low-level software, used to control the hardware components of the system (OS, device driver, utilities)

Application Software - used by the user to accomplish a specific task (web browser, media player)

**Generic Software vs Customized**

Generic - suitable for vast population of users (photoshop, ms office)

Customized - specific customers to meet their own needs

60% of software costs are development costs, 40% are testing costs

**Software Engineering vs Computer Science**

* comp sci focuses on theory and fundamentals, software engineering is concerned with the practicalities of developing and delivering useful software

**Software Engineering vs System Engineering**

* system engineering is concerned with all aspects of computer-based systems development including hardware, software, and process engineering, more general

**Software Process Activities**

Software Specification - customers and engineers define constraints

Software Design and development

Software Validation - checked to ensure it is what the customer requires

Software Evolution - modified to reflect changing customer and market requirements

**Stakeholder** - a person, group, or organization actively involved in a project and is affected by its outcome or can influence its outcome, directly or indirectly

Types: End users, system managers, system owners, system developers

**Issues of professional responsibility**

Confidentiality, competence, intellectual property rights, computer misuse

**Lecture 2: Software Process**

**Software Process:**

Specification - what the system should do

Design and Implementation

Validation - does it do what the customer wants

Evolution - changing the system in response to changing needs

**Software Development Life Cycle (SDLC)**

**Requirements Analysis**

* Process of establishing what services are required and the constraints on the systems operation and development, define in detail, check validity, requirements elicitation

**Design**

* Design a software structure that realizes the specification

**Implementation**

* Translate structure into an executable program

Design Activities-

Architectural : identify the overall structure of the system

Interface : interfaces between system components

Component selection (Search for reusable components)

Database model design (how it is represented in a database)

**Testing**

* Show that a system conforms to its specifications and meets requirements, design of test plan, test cases, and test cases executions

**Evolution**

* The software is flexible and can change through changing business circumstances

**Software Process Model**

An abstract representation of a process. It presents a description of a process from a particular perspective. Describes the order of activities and relations between them. Doesn’t specify details of the different phases of activities in the process

* Waterfall Model
* V-Shaped Model
* Spiral Model
* Evolutionary Prototyping Model
* Iterative and Incremental Model
* Integration and Configuration Model
* Agile Model

**Plan Driven Process vs Incremental Process**

Plan driven processes are where all of the process activities are planned in advance and progress is measured against this plan. Incremental is easier to change the process to reflect changing customer requirements. Practical processes include both

**Selecting a Software Process Model**

Understand the pros and cons of every model, understand capabilities and objectives

Common factors affecting include - size and skill of team, technology and tools used, geographical location of the team

**Waterfall Model** - difficulty of accommodating change, needs well understood requirements, large system usage, developed at several sites, plan driven

**V-Shaped Model** - same issues as waterfall, focuses on testing and validation. Use for mission critical systems.

**Spiral Model -** each loop represents a phase in the process, no fixed phases, risks are explicitly assessed and resolved throughout the process, rarely used

**Prototyping Model -** requirements engineering process to help with elicitation and validation, design process and testing process, may leave out functionality, focus on functional requirements

**Iterative and Incremental Model** - cost of accommodating customer needs is reduced, less analysis and documentation than waterfall, easy to get customer feedback, rapid delivery of useful software, regular change tends to corrupt its structure

**Integration and Configuration** - software reuse where systems are integrated from existing components, stand-alone application systems, objects developed as a package, web services

**Types of Reusable Software**

Stand-alone application systems

Packages to be integrated with a component framework

Web services

**Agile Model** - SCRUM, planning is incremental, customer value can be delivered with each increment, lower risk of project failure



**Process Improvement Cycle** - Measure, Change, Analyze

**Process Improvement Activities**

Process measurements

Process analysis

Process change

**Lecture 3.1: Requirements Engineering**

**What is a software requirement?**

A property which must be exhibited in order to solve some problem in the real world. Defines a users expectations from the software. A service the software must provide. A system property that can be implicit or explicit

**Types of Requirements:** Functional, Non-functional, UI, System, User, Business, Market

**What is requirements engineering?**

Establishing the services that the user requires from the system and the constraints under which it is to be developed and operated

Software requirements have a significant effect on the software design (architecture)

**Architecturally Significant Requirement (ASR)** - profound effect on the architecture, implicit requirement

* Specifies a software systems **quality attribute**, **core features**, impose **constraints**, defines **environment** the software will run

**Primary User:** interacts with the system directly and most affected by it.

**Secondary User**: does not directly interact with the system (UI) but is still affect by it

Functional Requirement - service the system should provide, how it should react and behave, explicit statement, complete and consistent, explicit statement

Non-Functional Requirements - constraints on the services, system as a whole, may be more critical than functional

Quality Attributes: ASRS:

● Design Qualities ○ Conceptual Integrity ○ Reusability ○ Modifiability

● Run-time Qualities ○ Availability ○ Performance ○ Security ○ Scalability

● System Qualities ○ Maintainability ○ Testability

● User Qualities ○ Supportability ○ Usability

**Requirements Elicitation** - involves technical staff working with customers to find out about the application domain, services it should provides, and constraints

**Requirements Analysis Stages** -

Requirements Discovery, Requirements Classification/Organization, Requirements Prioritisation, Requirements Specification

**Requirements Discovery Techniques:**

Interviewing: Formal or informal interviews with stakeholders

Ethnography: A social scientist spends time observing how people work

Stories and scenarios: Domain experts develop of how a system may be used for a task

**User Stories and Scenarios**

Real-life examples of how a system can be used for a particular task. Based on a practical situation, stakeholders can relate to them and can comment on their situation concerning the story

**Interviewing Stakeholders**

Closed Interview: based on predetermined list of questions

Open Interview: various issues are explored with stakeholders

* Normally use a mix of closed and open ended

**Lecture 3.2: Requirements Engineering Part 2**

**Software Requirements Prioritization (SRP):**

identify important requirements, implemented as early as possible

**Stakeholders** should prioritize the requirements. (end users, developers..) primary users!!

Success of a software is determined by the ability to satisfy requirements of stakeholders

**Requirements Prioritization Techniques**

1. **Nominal Scale Prioritization Techniques**

numerical assignment, categorized into groups (Top-Ten(K)), all requirements in one priority group have equal priority, used in early stages of project, medium or large # of reqs, most common

Top-Ten(K) Requirements - stakeholders pick their top ten, do not rank them, identify all that will be implemented, then categorized into groups

Ratio Scale Prioritization Techniques - ranked lists of requirements, show relative difference between requirements, more sophisticated and complex

1. **Ratio Scale Prioritization Techniques**

ranked lists of requirements, show relative difference between requirements, more sophisticated and complex

**Binary Priority List**

● In this technique the requirements are prioritized using a binary search tree. From a set of all requirements take any one requirement and put it as the root requirement (root of the tree)

● Take another requirement and compare it to the root requirement.

● If the requirement has a lower priority than the root requirement, compare it to the requirement to the left the root requirement. If the requirement has higher priority than the root requirement, compare it to the requirement to the right the root requirement.

● Repeated the previous 2 steps for all requirements.

● Finally, traverse the tree using in-order traversal method.

1. **Dependency Prioritization Techniques**

**Requirements Validation :**

**Validity**

Does the system provide the functions which best support the customers needs?

**Consistency**

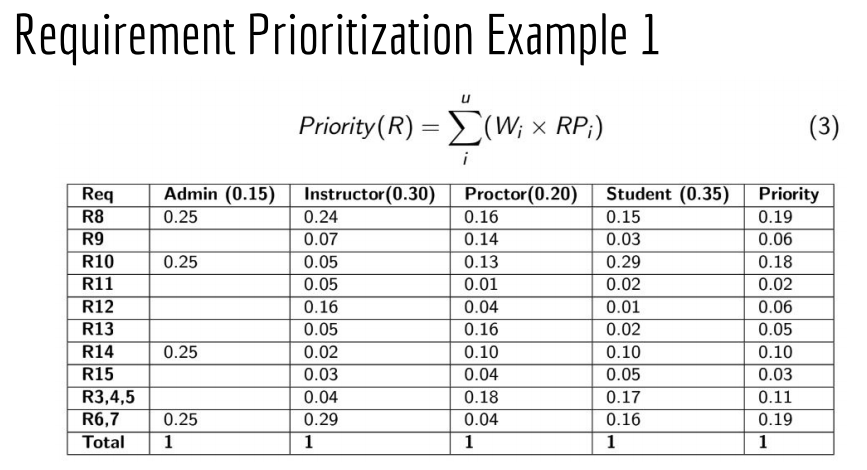
Are there any requirements conflicts?

**Realism**

Can the requirements be implemented given available budget and technology?

**Verifiability**

Can the requirements be checked?

****

**Requirement Specification:** The process of writing down the user and system requirements

User requirements have to be understandable by end users and customers without a technical background

**Lecture 4: Agile**

**Rapid software development-**

-software that can evolve quickly to reflect changing business needs

-Plan-driven development is important but does not meet the fast-changing needs of many businesses

**Agile development**

-Interleafing Design and implementation

-series of increments with frequent version evaluation and stakeholder involvement and every version

-minimal documentation

**Methods:**

Focus on the code NOT the design

Based on iterative approach

Deliver working software quickly -- respond to changes quickly and efficiently

1. Customer involvement -- closely involved, evaluate iterations, prioritize requirements
2. Incremental delivery -- customer specifying requirements
3. People not process -- team skills exploited - members develop their own way of working
4. Embrace change -- design system to accept changes (expect this)
5. Maintain simplicity -- in both software and the process

**Agile Programming Techniques:**

Extreme Programming (XP)

-iterative to the ‘extreme’

-new versions several times a day

-increments to customers every 2 weeks

-all tests run for every build -- only accepted if all successes

XP/Agile principles

-small frequent updates

-full-time customer engagement

-pair programming -- avoids long hours of work

-change supported - regular system releases

-simplicity - constant refactoring of code

**Refactoring**

-constant code improvement

-look for possible software improvements and act on it (even if there is no immediate need)

-code well-structured and clear (easy changes)

Examples:

-re-organizing a class hierarchy to remove duplicate code

-tidying up and renaming attributes an methods to make them easier to understand

-The replacement of inline code with calls to methods that have been included in a program library

**Test-driven development:**

-Writing tests before code clarifies requirements

-written as programs instead of data- execute automatically

-automatic test after each added functionality to ensure to errors are introduced

**Problems with test-first development:**

-programmers prefer coding to testing - skip steps in writing tests

-tests can be difficult to write incrementally

-hard to judge completeness of a set of tests

**Pair programming:**

-coownership - spreads knowledge

-informal review at each line

-encourages refactoring

-sit together at the same computer

-pairs created dynamically

-Knowledge sharing reduces risk when a team member leaves

-Not necessarily inefficient

**Scrum -**  agile method focused on managing iterative development (3 phases)

-planning (establish general objectives and design software architecture)

-a series of sprint cycles (each develops an increment)

-project closure (completes required documentation)

TERMINOLOGY (Slide 36-37)

Scrum Sprint Cycle:

-2-4 weeks

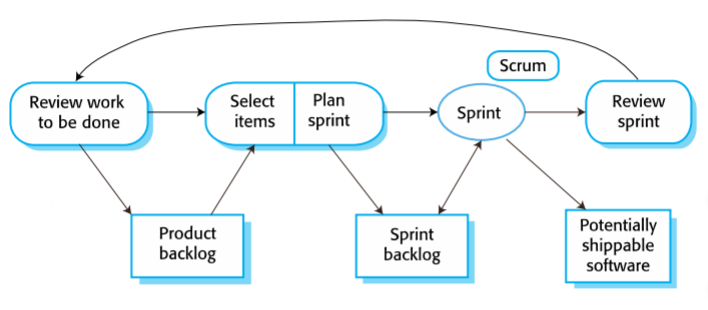
-starting point for planning is the product backlog -- list of work to be done

-project team with customers chooses next feature to implement during sprint

-during the sprint the team is isolated with all communication channelled through the Scrum master

-prevents external distractions

-at the end of the sprint, work is reviewed and presented to stakeholders… next sprint cycle begins



**Teamwork in Scrum:**

Scrum Master:

-daily meetings. Tracks backlog of work to be done, records decisions, measures progress against backlog, communicates with customers and management

The whole Team(Scums):

-attends short daily meetings to share information, describe progress, problems arisen, planned

-keeps everyone up to date

**Scrum benefits:**

-broken down to manageable and understandable chunks

-unstable requirements

-scrums have visibility of everything --- communication is improved

-customers see on-time delivery and gain feedback

-trust is developed and established between customers and developers

**Lecture 5: System Modelling**

**System Modeling** - process of developing abstract models of a system, each model presents a different view or perspective

**System Perspectives/Views**

External Perspective - model the environment of the system

Interaction - interactions between components, or system and environment

Structural - model organization or the structure of data

Behavioural - model dynamic behavior of the system

**How are software models/architectures described?**

**Informal Description** - using natural language (sketches)

**Semi-formal description** - standardized notations and modelling languages (UML, OMT)

**Formal Description** - formal notations with a precise semantic based on mathematical foundation (OCL, ADL)

System Views - External perspective, Interaction, Structural, Behavioural

**UML models...** system requirements, logical structure, workflows, interactions

**UML - Unified Modelling Language**

**UML Structure Diagrams**

1. Class Diagram - system classes, interfaces, types and relations
2. Object Diagram - objects would look in a particular scenario
3. Composite Structure Diagram - model key components and interface interaction
4. Component Diagram - key components in the system
5. Package Diagram - model hierarchical structure classes and components
6. Deployment Diagram - model physical view of system
7. Profile Diagram - describe an extension to uml by defining meta-model, constraints

**UML Behavioural Diagrams - describes all dynamic aspects of the system architecture**

1. Use Case Diagram

Model interactions between system and users or other external systems

1. Activity Diagram

Model sequential and parallel activities within system

1. Sequence Diagram

Model interactions between objects, useful when the order of interactions is important

1. State Machine Diagram

Model the states of an object during run time and events that affect them

1. Communication

Model communications between objects needed to support interactions

1. Timing

Model interaction between objects and how it affects the object state, fuses sequence and state diagrams

1. Interaction

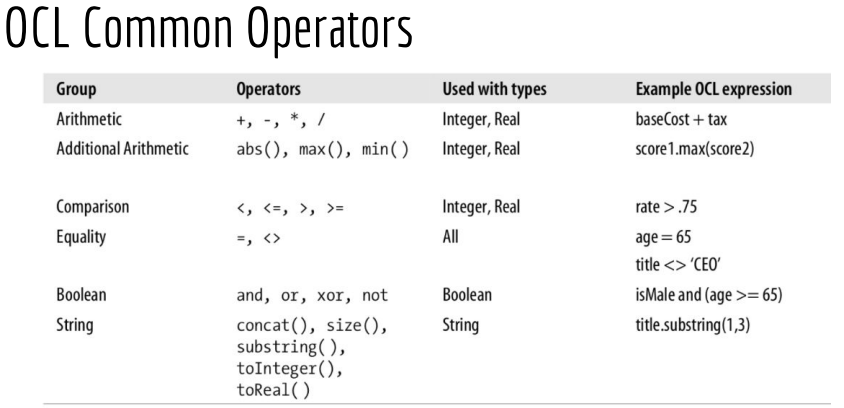
Provide a high level video of how several interactions work together to implement and system function/behaviour, combines both activity and sequence diagram

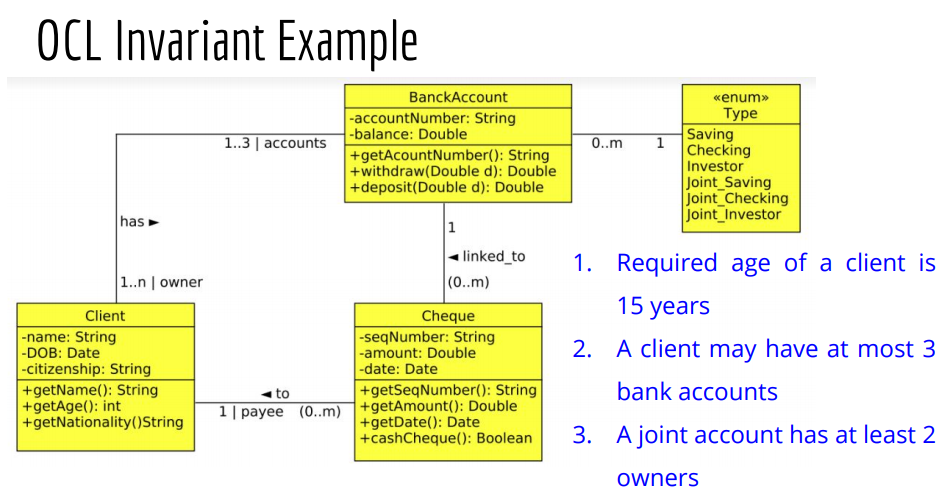
**Object Constraint Language (OCL) types:**

Invariant: constraint that must always be true. class attributes

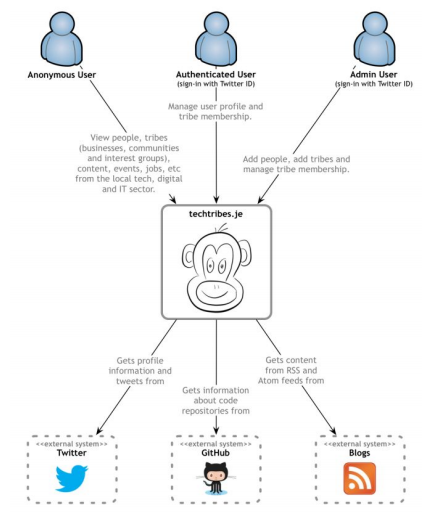
Precondition: constraint defined on a class method, used to validate input parameters

Postcondition: class method, checked after execution, how values were changed by a method





1. context Client inv self.getAge() >= 15
2. context Client inv self.accounts → size <=3
3. context BankAccount inv if (self.type = Joint\_Saving or self.type = Joint Checking or self.type = Joint Investor then self.owner → size >= 2)

**Context Models** - illustrate operational context of a system, what lies outside the system boundaries

Show the other systems in the environment, focuses on defining system boundaries

**Context Diagram** - high level diagram that sets the scene of the system

One box in center representing system with no details of interior, users, actors, roles, personas. External entities, and relationships between entities and system

**Structure Models**

Class Diagram, Component Diagram, Container Diagram

**Interaction Models**

Use Case Diagram - actors may be people or other systems

Sequence Diagram - interactions between actors and objects within a system

Behavioral Models - models of dynamic behaviour of a system as it is executing

Activity Diagram - logic of an algorithm, describe steps performed in a UML use case

State Diagram - how an object moves through various states, overall behaviour of state machine

**Lecture 6: Software Design and Architecture**

**Software Architecture:** The fundamental organization of a system embodied in its components, their relationships to each other, and to the environments and principles guiding its design and evolution. Also defined as creating a high level structure that satisfies both function and non functional requirements

Why is it important?

Ignoring software architecture will most likely lead to a software product that has poor quality attributes, does not meet business goals, misses some functional requirements

**Software Architect:** Responsible for,

Understanding requirements, business goals, constraints

Select the technologies and the resources

Make all architectural designs

Documenting

Present technical leadership

**4+1 Views Model**

* Describe architecture of a software system
* Describe the system from the viewpoints of different stakeholders

a ) Logical View

b ) Process View

c ) Development View

d ) Physical View

e ) Scenarios View

Scenarios View - describes functional requirements, core of the model. Typically use user stories, epics, and use case diagrams to present this view

Logical View - provide an abstract description of the system functions and how they help achieving the end user goals, usually visualized using class diagrams, state diagrams

Process View - provide a description of the processes in the system, describe system activities when performing a specific task or service, visualized using Activity diagrams

Development View - describes how the system is structured and organized into components and layers (Like ogres). Illustrates a system from a programmer's perspective and is concerned with software management, use package and component diagrams

Physical View - known as deployment view focus on mapping the system component on the physical layer, also describes physical connections between components, use deployment diagrams

**Software Design Principles**

**Coupling and Cohesion**

Coupling measures how closely connected two classes or modules are.

Cohesion is a measure of the strength of relationship between the methods of data of a class and some unifying purpose or concept served by that class.

\*\* HIGHLY COHESIVE AND LOOSELY COUPLED \*\* key requirement!

Attributes of Modern Software Architecture

1. cloud-ready
2. highly cohesive and loosely coupled
3. distinct components
4. reactive
5. enable asynchronous and synchronous processing
6. technologies agnostic
7. apply data partitioning
8. well defined APIs
9. adaptable
10. enable agility

**Key Software Design Principles**

1. Just Enough Upfront Design - only design necessary components, adaptability
2. Separation of Concerns - avoid functional overlapping between components
3. Do Not Repeat Yourself - duplication in logic, avoid adding unnecessary code
4. Single Responsibility Principle - an entity should only focus on one function
5. Principle of Least Knowledge - each component should only talk to its friends