1531 Definition Revision (19T1)

Requirements Engineering

- **Importance of software engineering to software development**
    1. deliever value to customer (realising customer's goals).
    2. requires a software engineer to apply imagination and have a good understanding of both the problem domain and the software domain to be able to build a conceptual domain model of the product visioned.
    3. minimize risks of loss of time, money or even human-life.
- **Software development methology**
    - Waterfall
    - Iterative & Incremental Process
- **Difference between Waterfall and Agile**:
    1. Waterfall follows a **linear** sequential model: requirements analysis, design, implementation, testing. Agile builds software in iterations where **each iteration implements all the four phases** on a set of feature.
    2. Waterfall is rigid and **not open to changes** in requirements. Agile is **open and adaptable** to changing requirements.
    3. In Waterfall, customer **only involve at the start and end** of the software life-cycle. Agile characterised by **continuous involvement throughout** the life-cycle (prioritizing work-items and providing feedback on each iteration deliverable.)
- **What are user stories?**
  Are one of the primary development artifacts of **Scrum and XP project teams** created the **requirements engineering phase.** (not for waterfall)

  In an Agile project, US are discussed in meetings with **the Product Owner** (who wirtes the US) and **the development team.**

    - Attributes
        - **I**ndependent
        - **N**egotiable
        - **V**aluable
        - **E**stimable
        - **S**mall
        - **T**estable
- **Role-feature-reason (RGB template)**
  R: Role, G: Goal, B: Benefit.
  As a `Role`, I would like to `Goal`, so that I can `Benefit`.

  <u>Cannot have US for developer.</u>
- **Acceptance Criteria**
  Any member of team can assits produc owner to defining and review
- **3C model**
    - card: us
    - conversation: detailing us at anytime

- confirmation: defining AC and mark them down

Domain Modelling

- **Domain model**
  - Also referred to as a **conceptual model** or **domain object model**
  - Benefit:
    - Triggers discussions about what is the central to the problem and relationships to the sub-parts.
    - Ensures that the system-to-be reflects a deep, shared understanding of the problem domain as the objects in the domain model will represent domain concepts.
    - The common language foster unambiguous shared understanding of the problem domain and requirements.
- **Functional and non-functional requirements**
- **Use case diagram**
  Only functional requirements included.
  - System boundary boxes
  - Actor
  - Associations
    - <<initiate>>: the actor has initiated the use-case.
    - <<participate>>: the actor participates in a use-case but dows not actually trigger it.
  - Structuring (dot line: feature not directly connect to actor)
    - <<include>>: A->B = A is included by B.
    - <<extend>>: A->B = A extend to B.
    - Abstract and generalised use-case: A B->C
- **UML class diagram** (Unified Modelling Language)
  Purpose: describes the structure of the software system to be. Is the main building block of object-oriented modelling. Is used both for general conceptual modelling of the problem domain.
  - Structural Diagrams (**static**): <u>class diagram</u>
  - Behavioural Diagrams/Interaction diagram (**dynamic**): show interaction between components over time <u>(e.g. activity diagram/sequence diagram/use case diagram)</u>
- **OO Design** (Object Oriented)
  - Object: real-word entities, has attributes (properties) and behaviour (methods)
  - interface: is the set of the object's methods thata can be invoked on the object

  An object is instantiated from a class, and the object is the instance of the class. A class is sometimes referred to as an object's type.
  - An object has state but a class doesn't.
- **OO principles**
  - Abstraction
  - Relationship
    - Association
      - Composition ("has a") -◆: the contained item is an integral part of the container.
      - Aggregation ("contains") -◇: the contained item can exist on its own.
    - Inheritance ("is a kind of") -▷
- **differnce between requirements (use-case) analysis and domain-modelling**
  - requiremetns (use-case) analysis: black box ("what" does the system deliever)
  - domain-modelling: white (transparent) box ("how" does it work)

- **main purpose of domain modelling**
    1. support the clarification of requirements.
    2. foster an unambiguous shared vision of the problem domain.
- **Meaning of encapsulation in OO design**: "hiding" or "protecting"
    - For attributes
      Protects the objects state (internal instance variables) from direct modification by restricting direct access to them. Ensure that they can only be observed/modified through object's public interface (methods).
    - For methods
      isolates chagnes to the internal implementation without affecting the service requester.
- **Benefits of Encapsulation**
    1. ensures that an object's **state is kept consistent**
    2. Keeping the data **private**. Data can only access through the methods provided. **Increases usability**.
    3. Abstract the internal implementation of the class, reduces the dependencies so that a change to the class does **not cause a rippling effect** on the system.
    4. Increases **reusability** of the object's class.
- **CRC card**
  <u>Not part of UML specification.</u> Flexible, often need to be modified.
    - component
        - **C**lass (class name)
        - **R**esponsibility (knowledge obtained and action can do)
        - **C**ollaborator (relationship to other classes)

Software Testing

- UAT (User Acceptance Tests)
- Black box testing (input->output)
- White box testing (test different scenario)
- Regression testing (verifying developed software)

- **TDD (Test Driven Development)**
  An important agile design technique. Is a practice that enforces writing tests before you start implementation of the user-story. Combines two techniques:
    - **TFD**: wite test before finish production code.
- **refactoring**
  modify internal structure **without changing** the external behaviour.
  Helps achieve high quality in XP principle.

  Goal: delever a specification that delievers the customer's goal.
  Principle:
    1. Write tests
    2. Write code and make necessary chagnes until test succeeds
    3. Refactor and eliminate redundancy

  regression testing: both new and old test succeed.

- **Equivalent testing** is a software testing technique that divideds the space of all possible inputs into a "software unit" to ensure the program "behaves the same" for each group. Only need one input for each case.

Equavalence classes are subset of input data.

  - Field: Valid Equivalence class / Invalid Equivalence class
  - Discription: Discription to the class

Agile

- **RUP** (Rational Unified Process)
  - Inception: scopte the project, identify major players, required resources, risk...
  - Elaboration: understand problem domain, analysis
  - Construction: design, build and test
  - Transition: release software to production
- **Benefit**
  - improve productivity
  - improve quality
  - imporve stakeholder satisfaction
  - reduce costs
  - risk-adjusted return
- **Agile not suitable for**
  - **risk-free/change-free** project.
  - very **large** program which need more specified and clear requirement.
  - when programmer and product owner is geographically **far away**
- **XP** (Extreme Programming)
  higher adaptability (to chagne requirement) and predictability (difining all requirements at the beginning)
  - Principle
    - High Quality
      - Pair programming: codes and reviews
      - Continuous Integration: check code often
      - Sustainable pace: moderate, steady pace
      - Open Workspace: open invironment
      - **Refactoring**: improve structure
      - **Test-Driven Development**: Unit-testing and User Acceptance Testing
    - Simple Design
      - Steady Goal: focus on current iteration's story
      - Migrate the design from iteration to iteration
      - Spike solution, prototypes, CRC cards techniques during design
      - Mantra: Keep simple; Don't do what not needed; Don't dulicate code
    - Continuous Feedback
      - Constant feedback from working pair/testing/integration
      - Daily feedback from daily meetings
      - Customer get feedback with user acceptance scores and at the end of each iteration
      - Programmer receive customer feedback
  - **XP Planning**
    - Initial Exploration: epic story and user story
      - Conversation: developer and customer identify significant features

        - User stories: broken from each feature
        - User Story Points: Estimated by developer
    - Release Plan: story point
        - Negotiate release date: customer specify needed US; customer can't choose more than velocity.
        - Project velocity: by time (velocity × US); by scope (total US / numbers of week)
    - Iteration Planning
        - developers and customer choose iteration size
        - customer cannot change the story once it has begun (can chagne others)
        - iteration ends on specific day even US are not done
        - "Done" means all acceptance tests pass
    - Task Planning
        - customer and developer arrange iteration planning meating at the beginning of each iteration
        - customer choose US
        - US brokendown into programming task
        - developer sign up for any task and estimate how long it take
        - user project velocity
        - estimates in ideal programming dates of the task are summed up
        - the velocity in task days overrides the velocity in "Release Plan"
        - team holds meating half way through iteration
  - **Product Backlog between iteration**
    - Customer has flexibility to change priorities
    - Items pulled by developers cannot be prioritized by customer
    - Developers have steady goal

Design Quality

- **Software Rot/Smell** (bad code)
  - Rigidity: too difficult to change, single change cuases lots of other dependent modules
  - Fragility: tendency of the software break when a singelchagne is made
  - Immobility: design hard to reuse
  - Viscosity: changes are easier to implement through 'hacks'
  - Opacity: difficult to understand
  - Needless complexity
  - Needless repetition
- **Design quality** is characterised by its degree of:
  - <u>cohesion</u>: all element collaborate as a functional unit, which has a single, well-focused purpose.
    Benefit of high cohesion:
    - highly cohesive classes are much easier to maintain and less frequently chaged
    - high cohesion renders the classes more usable than others as they are designed with a weill-focused purpose
  - <u>coupling</u>: the degree of interdependence between components or classes.
    - High coupling: A depends on the iternal workings of B and isaffected by internal changesto component B
    - Low coupling: allows components to be used and modified independently
- **Low coupling and high cohesion** to achieve:
  - Extensible

- Reusable
- Mantainable
- Understandable
- Testable

- **SOLID Principle**
  - **S**RP - Single Responsibility Principle
    - One class should have only one responsibility
  - **O**CP - Open Closed Principle (reduces rigility)
    - Open for extension: the behaviour of the class can be extended.
    - Closed for modification: extending behaviour of module should not required changeing original source.
  - **L**SP - Liskov Substitution Principle
  - **I**SP - Interface Segregation Principle
  - **D**IP - Dependency Inversion Principle

Databases

- **RDBMS (Relational Database Management System)**
  - base on relational data model (i.e. stores data as tuples or records in tables)
  - allows the ser to create relationship between tables
  - Example:
    - Open Source: PostgreSQL, MySQL, SQLLite
    - Commercial: Oracle, DB2(IBM), MS SQL Server, Sybase
- **Data Modelling**
  - Logical models: abstract model (ER Model, OOModel)
  - Physical models: record-based models (relation model, classes which deal with the physical ayout of data in storage)

  Strategy:
  - conceptual-level modelling: with entity relationship (ER) models
  - implementation-level modelling: transfrom ER design to relational model
- **Aims of Data Modelling**
  - describe what data is
  - describe relationships
  - describe contraints on data

  ∴ Data Modelling is a design process: converts requirements into a data model

- **ER (entity-relationship) data modelling**
  - **entity** (or entity instance):
    a thing or object of interest in the real-world and is distringuishable from other objects.

    like an object instance in OO models

    - strong entity
    - weak entity
  - **attribute:**
    a data item or property describing the entity.

    - simple
    - composite
    - single-valued

        - multi-valued
    - **An entity-set** (or entity-type) can be viewed as either:
        - a set of entities with the same set of attributes
        - an abstract description of a calss of entities

      <u>like a class in OO models</u>
    - **relationship** (or relationship instance)
        - total
        - partial
    - **relationsip type**:
      consis of a collection of relationships of the same type
    - **degree**
    - **cardinality**: one to one / one to many / many to many
    - **level of participation constraint**: total / partial
- **ER with subclass**
    - overlapping
    - disjoint

ER Model to Relational Model

- **Relation model**
  describes the world as a collection of inter-connected relations or tables. Component:
    - attribute (column)
    - domain (allowed value for an attribute)
        - has name, data type and format
        - NULL belongs to all domains
    - relation schema
    - database schema (collection of relation schema with constraints)
    - tuple (row or record): a set of values
    - relation (table)
        - no ordering
        - each relation generally has a primar key
    - key
        - super-key: whose set of values are distinct
        - candidate key: any super-key such as <u>no subset </u> is also a superkey
        - primary key: a candidate key that can uniquely identify an entity
    - foreign key

  Difference to ER model:
    - relation model uses relations to model entitites and relationships
    - relation model has no composite or multi-valued attributes
    - relation model has no object-oriented notions (subclasses, inheritance)
- **Degree of relation**: number of attributes
- **Constraits**
    - Domain constraint: attributes values can only from domain
    - Referential integrity constraint: cannot be referenced as foriegn key unless the primary key is created

    - Key constraint: has be unique but allow NULL
    - Primary Key constraint: has to be unique but not allow NULL
- **Relational Schemas**
    - SQL (Structured Qurey Language) provides the formalism to express relational schemas
    - SQL provides a Data Definition Language (DDL) for creating relations

Software Architecture

- **Definition:**

  Is a pattern of structural organization, which defines how the system must be decomposed into its parts and how these part relate to one another. Basically is defined by:

    - <u>Components</u>: a **collection of computational units** (e.g. classes, databases, tools, processes...)
    - <u>Connectors</u>: **enable communication** between components (e.g. function call, remote procedure call, event broadcast...) and uses specific **protocol**
    - <u>Constraints</u>: defines **how the components can be combined** to form the system
- **Importance** (focus on the non-functional requirements)
    - **Partition complex system** into sub-system ("divide and conquer")
    - Helps to **focus on creative part** and avoid "reinventing the wheel"
    - Support **flexibility and future evolution** by decoupling unrelated parts ("separation of concerns")
    - Pre-determine key non-functional requirements (scalability, reliability, performance, usability etc...)
    - Promotes understanding and communication among stakeholder, end-user, architects and developers.
- **Some Architectural Style**
    - <u>Client/Server (2-tiered, n-tiered. World Wide Web style REST)</u>
        - Server: provides services, handle connections
        - Client: request services
        - Connector: is based on a **request-response** model
        - Example: File Server, Database Server, Email Server, Web Server
        - Benefits: effective, easy to add new server
        - Weakness: single point of failure; network congestion; complex and expensive
    - <u>Peer-to-Peer</u>
        - Each peer can be both server and client (Central server only store hash table)
        - Example: BitTorrent, Skype, Bitcoin
        - Benefit: Efficiency; Scalability; Robustness (not depend on single peer)
        - Weakness: complexity; resource not always available; more demanding of peer
    - <u>Pipe-and-Filter</u>
        - Component: Filter (one by one, independent, do not share state but works concurrently)
        - Connector: Pipe
        - suitable for processing and transforming data stream
        - Example: Unix shell script, compilers
        - Benefits: easy to understand; support reuse (only for agreed data format); flexible; support concurrent processing of data stream
        - Weakness: highly dependent on order of filter; input and output format has to be compatible to each other
    - <u>Central Repository</u>
        - Components:

        - Central data repository: central, reliable, permanent, representing state of the system
        - Data accessors: independent, do not interact directly and shared data (e.g. graphical editors, IDEs, database app, document repositories)
    - Connectors: Read/Write mechanism (e.g. procedure calls or direct memory accesses)
    - Benefit: efficient to share large amounts of data; Centralised management (concurrency access, security, back up)
    - Weakness: all components must agree on a repository data model; distribution of data; complex
  - <u>Publish-subscribe (Event based)</u>
    - Components:
      - Publisher: don't know the subscribers
      - Subscriber
    - Example: subscribe; stock; wireless sensor networks
    - SOA (Service Oriented Architecture)
    - Components: are created as autonomous, platform-independent, loosely coupled **services.**
    - Applications: B2B (Business to Business) services
- **MVC** (Application Architectural Pattern)
  - Decouple <u>data access</u>, <u>application logic</u> and <u>user interface</u> into three distinct components (and can be different server).
  - **Not a software architectural style**
- **Architecture view**

  is a projection of a model showing a subset of its detail.
  - Model view
    - decomposes functionality
    - sequence diagrams, UML class diagrams...
  - Component and Connector view
    - Describes a runtime structure of the system such as components, connectors (pipes, socket) ...
    - Box-and-line diagram (informal), UML component diagram (formal)
  - Allocation view
    - Describes how the software units map to the environment (hardware resources, file-system and people)
    - UML deployment diagram: static view, show the hardware for system (more physical, how the software allocated in sytsem)