CMPT 373: Software Development Methods

A Course Overview

Jeffrey Leung
Simon Fraser University

Fall 2019

Contents

1 Introduction

- Course objectives: Gain practical agile experience, limit complexity of large systems
- Software development activities: Requirements-gathering, design, coding, testing, delivery
- Team roles:
  - **Project owner:** Team member who understands and manages product vision, and manages the backlog with the goal of maximizing business value
  - **Scrum master:** Team member who guides the cohesion, organization, and performance of the team with the goal of creating a self-organizing agile team
    - Is an advisor and advocate, not the boss or decision-maker
  - **Team member:** Member who works to deliver the user stories which were chosen
  - **Repository manager (CMPT 373 only):** Team member who enforces code commit process

2 Teamwork

- **Group:** People working on similar tasks without collaborating
- **Team:** People collaborating on developing trust and making decisions together to achieve a shared objective
- Team stages:
  - **Forming:** Team stage when members get used to each other
  - **Storming:** Team stage where opinions conflict and members compete
  - **Norming:** Team stage where conflicts are resolved and members are comfortable with effective teamwork
  - **Performing:** Team stage where work is productive and the team works cohesively
  - **Transforming:** Team stage where work is very productive
  - **Adjourning:** Team stage where the project is closed and the team is disbanded
- Team rules:
  - Respect everyone
  - Criticize an idea or piece of work, not the person
  - Praise in public, criticize in private
  - Communicate
  - Agree on how decisions will be made
  - Avoid groupthink; embrace conflict
- Advice:
  - Be proactive
  - Everyone should succeed
    - Be co-operative not competitive
    - Express your ideas
  - Understand first, then be understood
  - Collaborate
    - Uniformity does not create unity
    - Value diversity

3 The Agile Paradigm

- **Plan Driven Process / Big Design Up Front (BDUF):** Software planning paradigm which lays out as much detail and timeline as possible at the beginning of the project
  - Requires perfect planning to succeed
- **Agile/iterative:** Software planning paradigm which constantly changes and improves on previous work
  - *Planning-driven*
  - Plan and test constantly
  - Constantly improved documentation
  - **(User) story:** Feature request which provides value to a stakeholder
  - **Task:** Program changes to advance a story
  - **Backlog:** Prioritized set of stories and tasks which are not currently planned for work
- **Sprint:** Period of time during which a set of stories are chosen to complete over the time
  - **Velocity:** Amount of work completed in previous iterations
- Meetings:
  - **Stand-up:** Meeting where each team member briefly updates the team on what they accomplished since the previous meeting, what they want to accomplish before the next meeting, and what obstacles are causing issues
  - **Sprint review:** Meeting where the team demonstrates working software to stakeholders
    - Product owner gathers feedback
    - No promises should be given
  - **Retrospective:** Meeting where the team reviews work done and identifies improvements
    - Improves the team, identifies things to improve, and creates an action plan to implement changes
  - **Sprint planning:** Meeting where the team chooses the next stories and tasks for the upcoming sprint
    - Choose new tasks by matching the velocity of the previous sprint ecompose stories into tasks
- **Sprint backlog:** List of tasks to deliver for a sprint
- **Story point:** Smallest possible unit of work
  - Based on effort, not work
  - Express each story in terms of points

4 Design and Architecture

- 3-tier architecture:
  - **Presentation layer:** System component which manages the user interface and interaction
  - **Business logic layer:** System component which handles data processing and manipulation
  - **Data layer:** System component which manages persistent data
  - Modular, logical, maintainable, testable
- **Class-Responsibility-Collaborator (CRC) Card:** Set of cards each showing the responsibilities and connections of a single class
  - Data classes are described by what information is processed
- **Unified Modeling Language (UML):** Method of diagram creation which illustrates relationships between entities
  - Connections:
    - Arrow: Has a
    - Arrow with label $0..*$: Has 0 or more
    - Arrow with triangle head: Subclass of a class
    - Dashed arrow: Dependency/usage
    - Dashed arrow with triangle head: Implementation of an interface
    - Entity label: Stereotype descriptor
- Software design:
  - Wicked, sloppy, heuristical, iterative
  - Required for construction
  - Requires implementation to accurately assess its effectiveness
  - Avoid inheritance until necessary; replace with dependency
  - Encapsulate whenever possible
- **Tracer bullet:** Skeleton implemetation of an end-to-end process in a system
  - Provides architectural framework to iterate on
- Methods of reducing complexity:
  - Limit the amount of details being considered at any given time
  - Having a code style standard for naming, brackets, indentation, spacing, comments, etc.
  - Encapulate and abstract
  - Design constructors to create fully formed objects
- **Loose coupling:** Concept of two classes who rely on little or no implementation detail of each other
- **Tight coupling:** Concept of two classes, one of which relies on a specific concrete type
  - E.g. Instantiating a new object

4.1 Design Patterns

- **Design principles:**
  - Separate changeable aspects of the program from aspects which will stay the same
  - **Open-closed principle:** Classes should be open for extension and closed for modification
  - **Composition over inheritance:** Prefer has-a relationships over is-a relationships
    - Allows easier modification and separation of responsibilities, as well as attachment and removal at runtime
  - Design with an interface, not a concrete class
- Types of design patterns:
  - **Creation patterns:** Architectural patterns which handle instantiation (e.g. factory, singleton)
  - **Structural patterns:** Architectural patterns which handle composition (e.g. decorator)
  - **Behavioural patterns:** Architectural patterns which handle communication between objects and distribute reponsibilites (e.g. iterator, observer, strategy, null object)
- **Inheritance:**
  - Base class functionality changes may unintentionally affect derived classes, which consequently means:
    - Local changes have non-local effects
    - Structure is inflexible for code maintenance
    - Object behaviour cannot be changed at runtime
  - Derived classes may need to remove base class functionality
- **Dependency injection (DI):** Architectural pattern where object creation is unlinked from object usage; the main program handles dependencies and passes references of dependencies to client code
  - **Client (DI)**: Class in DI which uses a service instantiated by the main class
  - **Injector (DI)**: Class in DI which instantiates a service object and provides it to the client for usage
    - **Constructor injection:** Injection method in DI where the framework constructs the object while providing the services
    - **Setter injection:** Injection method in DI where the framework constructs the object then providing the services
    - **Framework injection:** Injection method in DI where the framework sets the class members to references of the services
  - Provides ability to mock out all dependencies
  - Requires designing dependencies to use an interface
  - More difficult to trace code
- **Observer pattern:** Architectural pattern consisting of a UI and Model
  - Synchronization happens when either:
    - The UI repeatedly polls for updates (con: delayed, resource-intensive)
    - The Model has a reference to the UI and calls it when an event occurs (con: tight coupling)

        * The UI registers an Observer with the Model; the Observer implements a notify() function which the model calls
    - Benefits:
        * Loose coupling due to registering an observer at runtime
        * Multiple observers can register with th subject
        * Decouples model and UI
    - Drawbacks:
        * Difficult to trace code

- **Strategy pattern:** Architectural pattern where a family of interchangeable modules each contain an algorithm or unique component, which allows the algorithm to be easily modified

- **Decorator pattern:** Architectural pattern where a decorator subclass wraps a reference to the base class, and can be recursively wrapped with continually more decorators
    - Dynamically attaches behaviour/properties to an object
    - UML diagram will have both an is-a and has-a relationship between two classes
    - Decorator classes share the same supertype as their objects
    - Can be stacked multiple times
    - Avoids violating the open-closed principle
    - Cons
        * Adds many small classes
        * Difficult to learn an architecture
        * Method of instantiation is complex

- **Singleton pattern:** Architectural pattern where only a single instance of an instance can exist and has a global point of access
    - Allows lazy instantiation
    - Allows global dependency
    - Does not allow inheritance
    - Difficult to mock and test