Lesson 1

This lesson will serve to provide and overview of the entire course then cover questions such as:

1. What is software engineering?
2. Why do we need it?

Importance Of Software Engineering

There are many takes on the importance of software engineering. Software engineering today involves many stakeholders and engineers working together to deliver technology that meets the requirements of all stakeholders and the customer. Today, software is ubiquitous with everyday life.

Discipline Of Software Engineering

Why is it so difficult to built good software? To build good software requires:

- Methodologies
- Techniques
- Tools

The deliverable must be a *high quality* product that *works* and *fits the budget*.

The Software Crisis

There were many reasons for the software crisis in the 1960s:

1. The demand for software from 1950 to 2000 grew exponentially
2. The development effort for bigger deliverables (e.g., operating system, distributed system, etc.) became a *software engineering* effort
3. The software developers' productivity could not keep up with the software size and complexity

Evidence of the Software Crisis

In 1990, Davis studied nine software development contracts where majority of the budget spent on software ($5M out of $7M) resulted in software that was:

- Delivered but never successfully used
- Not delivered

This study resulted in the NATO software engineering conferences starting in January 1969.

Software Development

Software development is the process in which an abstract idea becomes a concrete system that implements the abstract idea. For this to happen, an abstract idea must go through several software processes which are:

1. Systematic
2. Formal

Such that many different parts can come together to implement the abstract idea.

Software Process

There are many different software processes. In this course we will cover:

1. Waterfall
2. Evolutionary prototype
3. RUP/USP (Rational Unified Process/Unified Software Process)
4. Agile

Software Phases

There are various software phases in the software process:

1. Requirements engineering
2. Design
3. Implementation
4. Verification and validation
5. Maintenance

Tools Of The Trade

Software tools can help close the gap between developers' productivity and software size and complexity. The main tools we will examine in this class are:

1. IDE (integrated development environments)
2. VCS (version control systems)
3. Coverage and verification tools

Section Quizzes

Software Failure Quiz

What is this (image provided)?

Explosion of Ariane 5 rocket due to software errors.

Software Crisis Quiz

What are the major causes of the software crisis?

- Increasing product complexity
- Slow programmers' productivity growth
- Rising demand for software

Preliminary Questions 1 Quiz

1. *What is the largest software system on which you have worked*? Banking
2. *How many LOC/day were you producing*? Depends.

Preliminary Questions 2 Quiz

How many LOC/day do you think professional software engineers produce?

Around 50-100 LOC.

Lesson 2

In this lesson we will go through several software engineering lifecycle models as well as the pros and cons of each. A **software lifecycle** is a sequence of decisions that determine the history of your software.

Traditional Software Phases

Recall that there are several phases in software:

1. **Requirements engineering**: the process of establishing the needs of stakeholders that are to be solved by software
2. **Design**: consists of various design activities beginning from a high-level of software design to a low-level
3. Implementation

   : the process of taking designs and building out the software; there are four principles of implementation:

   1. Reduction of complexity
   2. Anticipation of diversity
   3. Structuring for validation
   4. Use of external standards
4. **Verification and validation**: e.g., did we build the right system and did we build the system right?
5. Maintenance

   : the process of maintaining the software built; this process also includes:

   - Corrective maintenance
   - Perfective maintenance
   - Adaptive maintenance

Software Process Model Introduction

The software process model should describe what we should do next and how long we should continue to do it for each activity.

Software Processes

There are various software lifecycle processes:

- Waterfall

  : a software process that occurs in a linear fashion

  - Pros: finding errors early
  - Cons: this process is not flexible
- Spiral

  : a process that involves several of steps (determine objectives, identify and resolve risks, development and tests, and plan the next iteration)

  - Pros: risk reduction, added functionality, early software delivery
  - Cons: specific expertise, dependent on risk analysis, complex
- Evolutionary prototyping

: mainly based around prototyping

    - Pros: immediate feedback
    - Cons: difficult to plan
- **RUP (Rational Unified Process)**: is a UML (Unified Modeling Language) based process which includes inception, elaboration, construction, and transition for each activity
- **Agile**: a group of software development methods which are based on highly iterative and incremental development

Choosing A Model

Choosing the right model for software development depends on many factors below are several to consider:

- Requirements understanding
- Expected lifetime
- Risk
- Schedule constraints
- Interaction with management and customers
- Expertise

Classic Mistakes

There are several classic mistakes during the software development process:

1. People: heroics, work environment, and people management
2. Process: scheduling issues, planning issues, failures
3. Product: gold plating, feature creep, development that is actually research
4. Technology: silver bullet syndrome, switching tools, no version control

Section Quizzes

Software Phases Quiz

What are the traditional software phases?

1. Requirements engineering
2. Design
3. Implementation
4. Verification and validation
5. Maintenance

Choosing A Model 1 Quiz

Which of the following models is most suitable to develop a software control system?

Pure waterfall.

Choosing A Model 2 Quiz

Which model is the most suitable for a software project if you expect midcourse corrections?

Spiral or evolutionary prototyping.

Lesson 3

In this lesson we will go over IDEs (Integrated Development Environment) - in particular Eclipse - and how developer tools can enhance the developer experience.

Eclipse Introduction

From Wiki:

> Eclipse is an integrated development environment used in computer programming. It contains a base workspace and an extensible plug-in system for customizing the environment.

IDE Overview

What is an IDE? An IDE provides the following:

1. Views
2. Source code editor
3. SCM (software configuration management)
4. Builders
5. Runtime
6. Testing
7. Debugger

Plug-ins

What is a plug-in? From wiki:

> In computing, a plug-in (or plugin, add-in, addin, add-on, or addon) is a software component that adds a specific feature to an existing computer program. When a program supports plug-ins, it enables customization.

Lesson 4

This week, we continue the discussion on developer tools. This lesson covers version control systems - particularly Git.

Interview With John Britton

A **VCS (version control system)** helps developer manage various revisions or versions of source code. Git is one such VCS which can enhance source code management.

Version Control System Introduction

Why is VCS useful?

1. Enforces discipline
2. Archives versions
3. Maintains historical information

4. Enables collaboration
5. Recovers from accidental deletions or edits
6. Conserves disk space

Essential Actions

There are several essential actions when it comes to version control:

1. Add
2. Commit
3. Update

Do Nots In VSC

There are a couple of things we should not do in VCS:

- Do not add derived files (such as executable files)
- Do not add bulky binary files
- Do not use local copies

Two Main Types Of VCS

There are two main types of VCS:

- **Centralized**: each person works directly with a centralized repository
- **Decentralized**: each person works directly with a forked-copy of the upstream repository and can push final changes to the upstream repository

Git Workflow

There are several of steps in a Git workflow:

1. Workspace (working directory)
2. Index (stage)
3. Local repository (HEAD)
4. Remote repository

Section Quizzes

VCS Quiz

Have you used any version control systems (VCS)? Which ones?

Git and TortoiseSVN.

Lesson 5

In this lesson we will focus on requirements and prototyping including requirements engineering activities and object-oriented design.

Interview With Jane Cleland-Huang

Q: What are **software requirements**?

A: Software requirements provide us a description of what a system has to do. They typically describe functionality of the features that system has to deliver in order to satisfy its stakeholders.

Q: What is **RE (requirements engineering)**?

A: RE covers a number of activities such as:

- Working with stakeholders to discover requirements
- Analyzing discovered requirements to weigh tradeoffs
- Specifications that describe software functionality
- Validation of deliverable
- Requirements management process (change management)

RE includes *engineering* in the term because it requires a systematic process throughout the whole lifecycle of the deliverable.

General RE Definition

RE establishes the services that the customer requires from the software system. RE also establishes the constraints under which the software system operates.

SRS (software requirements specification) which is established under RE should focus on the the *what* (e.g., what should our software system do) rather than the *how* (e.g., how should our software system accomplish tasks).

Software Intensive Systems

What do we mean when we talk about software systems? A software system is really a software *intensive* system that includes software, hardware, and context in which the system is used.

Software Quality

Based on what we discussed about software intensive systems, software quality is not just a function of our software but also the user experience provided by our deliverable.

Identifying Purpose

Identifying a purpose is to define requirements. This is a very difficult task due to factors such as:

- Sheer complexity of purpose/requirements
- People not knowing what they want
- Changing requirements
- Conflicting requirements

Completeness And Pertinence

It is very difficult to have a complete picture of the software. Another issue occurs when software developers collect too many irrelevant requirements. Requirements must be accurate, complete and pertinent.

RE Definition Breakdown

The RE definition can be broken down as follows:

1. RE is a set of activities
2. Communication is as important as analysis
3. Quality means fitness-for-purpose (must understand software purpose)
4. Designers need to know how and where the system will be used
5. RE is split between needs and what is possible
6. Successful RE depends on identifying all stakeholders

Defining Requirements

Requirements can be application domain or machine domain:

Application domain is the world in which software will operate and are characterized by:

- Domain properties
- Requirements

Machine domain is the domain on which the software will run and are characterized by:

- Computers
- Programs

Lastly, **specifications** are formal descriptions of what the software system should do in order to fulfill requirements in both domains. Specifications must take into account:

1. *Events* in the real world that the machine can *sense*
2. *Actions* in the real world that the machine can *cause*

Functional And Non-functional Requirements

There are two general types of requirements for system at a high-level:

1. **Functional**: what the system does or computation to solve the problem
2. **Non-functional**: refers to the system's qualities, e.g., security, usability, etc.

User And System Requirements

We also must take into account user and system requirements:

User requirements:

- Written for customers
- Often in natural language, no technical details

System requirements:

- Written for developers
- Detailed functional and non-functional requirements
- Clearly and more rigorously specified

Requirements Origins

So where do these requirements come from? There are many possible sources, below are some of the main sources:

- Stakeholders
- Application domain
- Documentation

Elicitation Problems

During software development, there exists several elicitation problems which make gathering requirements difficult:

1. Thin spread of domain knowledge
2. Knowledge is tacit
3. Limited observability
4. Bias

Traditional Techniques

Thankfully, there are some techniques we can use to combat elicitation issues discussed earlier:

- **Background reading**: reading through existing documentation on the problem space to gather requirements but can be cumbersome and open-ended
- **Hard data and samples**: includes using already collected data and samples to establish requirements but can be difficult to determine what data is useful
- **Interviews**: can probe users in-depth to determine requirements but typically requires an expert who knows how to interview to optimize results
- **Surveys**: can quickly collect information from a large number of people but can constrain the information the users can provide
- **Meetings**: can collectively decide on useful or not useful information for requirements but needs to be properly organized to be optimal

Other Techniques

There also exists other techniques for requirements gathering:

- **Collaborative techniques**: supports incremental development of complex systems with large diverse user populations
- **Social approaches**: techniques which exploit social sciences to better collect information from stakeholders and environment (e.g., ethnographic techniques which can help with observing users in their original environment)
- **Cognitive techniques**: leverages cognitive science to discover expert knowledge

Modeling Requirements

Once requirements are established, it is important to decide what to model and how to model:

1. *What to model* depends on which aspects of the requirements are most important
2. *How to model* will offer different perspectives on the models selected for modeling

Analyzing Requirements

There are three steps when analyzing requirements:

1. **Verification**: *developers* check if the requirements accurately address customer needs as well as completeness, relevance, etc.
2. **Validation**: *stakeholders* check if the collected requirements define the system that the stakeholders desire
3. **Risk analysis**: relates to analyzing any possible risks involved in development and mitigating those risks

Requirements Prioritization

Requirements should be prioritized as there are limited resources and an inability to satisfy all requirements in the real world. Prioritization may be categorized as follows:

- Mandatory: i.e., a feature that is critical to address customer needs
- Nice to have: i.e., a feature that may compliment customer needs
- Superfluous: i.e., a feature that is not actually needed

Requirements Engineering Process

In general, there are four main steps in the requirements engineering process:

1. **Elicitation**: abstracting requirements from various sources
2. **Modeling**: representing the requirements using formal notation
3. **Analysis**: identifying possible issues with the requirements
4. **Negotiation**: discussions between developers and stakeholders regarding requirements until a consensus is reached

Note that this process is a highly iterative process and make take many iterations before requirements are finalized.

SRS

Why is the SRS important? The SRS offers a way to communicate requirements to others. Different projects require different SRS. The IEEE has defined a standard in which requirements may be specified:

1. Introduction
2. User requirements
3. System requirements (functional and non-functional)

It is important to note that requirements should have the following properties:

- Simple (specific not compound)
- Testable
- Organized
- Numbered (traceability purposes)

Section Quizzes

Pertinence Quiz

Consider an information system for a gym. In the list below, mark all the requirements that you believe are pertinent.

- Members of the gym shall be able to access their training programs
- The system shall be able to read member cards
- Personal trainers shall be able to add clients

Completeness Quiz

> Consider an information system for a gym. In the list below (provided), mark all the requirements that you believe are pertinent.

1. *Is the above list complete*? No

Irrelevant Requirements Quiz

Why can irrelevant requirements be harmful?

- They can introduce inconsistency
- They can waste project resources

Defining Requirements Quiz

> Referring to the figure that we just discussed (provided), indicate, for each of the following items, whether they belong to the machine domain (1), application domain (2), or their intersection (3).

1. *An algorithm sorts a list of books in alphabetical order by the first author's name*. Machine domain
2. *A notification of the arrival of a message appears on a smart watch*. Their intersection
3. *An employee wants to organize a meeting with a set of colleagues*. Application domain
4. *A user clicks a link on a web page*. Their intersection

Requirements Quiz

Which of the following requirements are non-functional requirements?

- The WordCount program should be able to process large files
- The Login program for a website should be secure

Requirements Prioritization Quiz

> Imagine that you have collected the following set of five requirements for an ATM system but only have resources to satisfy two, possibly three of those
>
> Suitably prioritize the requirements by marking them as Mandatory, Nice to have, or Superfluous.

1. *The system shall check the PIN of the ATM card before allowing the customer to perform an operation*. Mandatory
2. *The system shall perform an additional biometric verification of the customer's identity before it allows the customer to perform an operation*. Superfluous
3. *The system shall allow customer to withdraw cash using an ATM card*. Mandatory
4. *The system shall allow customer to deposit money using an ATM card*. Nice to have
5. *The system shall allow customer to change the PIN of an ATM card*. Superfluous

Lesson 6

Previously we covered requirements engineering. In this lesson we will look into OOD (object oriented design) and UML (Unified Modeling Language) which we will use in the rest of the course.

Object Orientation Introduction

What is **OO (object orientation)**? Object orientation focuses several key concepts:

1. Data over function
2. Information hiding
3. Encapsulation
4. Inheritance

Objects And Classes

An **object** is a computing unit organized around a collection of state or instance variables that define the state of the object. Each object has operators which are methods that allow reading and writing instance variables.

On the other hand, a **class** is a template or blueprint from which new objects (instances) can be created.

Benefits Of OO

Why should we use OO?

1. Reduce maintenance costs
2. Improve development process
3. Enforce good design

OO Analysis History

At the time OMT (Object Modeling Techniques) were being developed (led by Rumbaugh), scientists (Jacobson and Booch) also began to think about another technique (UML). OMT comprises of data, functions, and control.

OO Analysis Overview

In OO analysis, we are concerned about defining the data objects first then the functions between each object second. We also look to transfer real world objects into requirements.

UML Structural Diagrams: Class Diagrams

We will discuss several UML diagrams in this lesson. The class diagram will be one of them. The class diagram is a static, structured view of the system which describes: classes and their relationships.

Classes include:

- Class name
- Attributes and their types
- Operations and their types

Attributes represent the structure of a class and may be found by:

- Examining class definitions
- Studying requirements
- Applying domain knowledge

Operations represent the behavior of a class and may be found by examining interactions among entities.

Relationships describe interactions between objects. There are several types of relationships:

- Dependencies (e.g., x uses y)
- Associations/aggregations (e.g., x has a y)

- Generalization (e.g., x is a y)

Class Diagram: Creation Tips

Below are several recommendations when working with class diagrams:

1. Understand the problem
2. Choose good names
3. Concentrate of the *what*
4. Start with a simple diagram
5. Refine until you feel it is complete

UML Structural Diagrams: Component Diagram

The component diagram is a static view of components and their relationships. Unlike class diagrams, component diagrams have:

- *Nodes* to represent components (set of classes with a well-defined interface)
- *Edges* to represent a relationship

Component diagrams can be used to represent an architecture.

UML Structural Diagrams: Deployment

The deployment diagram is a static deployment view of system. Physical allocation of components to computational units.

- *Nodes* are computational units
- *Edges* represent communication

UML Behavioral Diagrams: Use Case

Use case diagrams describe the outside view of the system as well as:

- Sequence of interactions of outside entities (actors) with the system
- System actions that yield an observable result of value to the actors

Use cases have three parts:

1. Use case
2. Actor (human or a device)
3. Relationship

Building A Use Case Diagram

The behavior of a use case can be specified by describing its flow of events:

- How the use case starts and ends
- Normal flow of events
- Alternative flow of events
- Exceptional flow of events

Role Of Use Cases

There are several of reasons why use cases are important:

1. Requirements elicitation
2. Architectural analysis

3. User prioritization
4. Planning
5. Testing

Use Case Diagram: Creation Tips

Below are several recommendations when working with use case diagrams:

1. Use name that communicates purpose
2. Define one atomic behavior per use case
3. Define flow of events clearly
4. Provide only essential details
5. Factor common behaviors
6. Factor variants

UML Behavioral Diagrams: Sequence

Sequence diagrams emphasizes the time ordering of messages.

UML Behavioral Diagrams: State Transition Diagram

State transition diagrams specify the events that cause an object to move from one state to another and effects. These diagrams are composed of the following for each class:

- Possible states of the class
- Events that cause a transition from one state to another
- Actions that result from a state change

Section Quizzes

OO Benefits Quiz

Acme Corporation decided to use an OO approach in its software development process. What benefits can they expect to receive from this decision?

- Increased reuse because of the modular coding style
- Increased maintainability because the system design can accommodate changes more easily
- Increased understandability because the design models real-world entities

Modeling Classes Quiz

Consider the following requirement for an online shopping website: "Users can add more than one item on sale at a time to a shopping cart." Which of the following elements should be modeled as classes?

- Item
- Shopping cart
- User

Class Diagram Relationships Quiz

Which of the following relationships is an actual relationship for the system we are modeling?

- `RegistrationManager` uses `SchedulingAlgorithm` (dependency)
- `RegistrationManager` uses `Student` (dependency)
- `Student` registers for `CourseOffering` (association)
- `Course` consists of `CourseOffering` (aggregation)

- `Student` is a `RegistrationUser` (generalization)
- `Professor` is a `RegistrationUser` (generalization)

Recap 1 Quiz

An UML state transition diagram specifies:

- The events that cause an object to move from one state to another
- The effects of a state change

Recap 2 Quiz

Which of the following diagrams are UML structural diagrams?

- Class diagram
- Deployment diagram

Lesson 7

In this lesson we will go over the **Android** system - an operating system designed for mobile devices. Specifically we will cover the conceptual and practical aspects of Android.

Android Introduction

What is Android? Android is:

- Designed for mobile devices
- Based on the Linux Kernel
- Powered by the Dalvik VM

Basic Architecture Of Android

The architecture of android beginning with the highest level is:

1. Apps
2. Application Framework
3. Libraries and Android runtime
4. Linux kernel

Android App

The Android app is comprised of:

1. **Activities**: independent components such as a single screen with a user interface; can work together to form a cohesive whole
2. **Services**: a component that typically performs long running operations in the background such as a service for playing music or a downloading files
3. **Content provider**: provides a structure interface to a set of data such that data can be provided to various applications
4. **Broadcast receiver**: a receiver that could be registered to receive system or application events

Each one of the above components is connected by *intents* and all component properties are declared in the Android manifest `.xml` file

Intents

Intents are abstract descriptions of an operation to be performed and it consists of two main parts:

1. Action - the task to be performed
2. Data - the data on which the action will operate

E.g., intent to call a contact involves a *call* (action) which uses a *phone number* (data). For more information on intents, see the Android documentation.

Lesson 8

In this lesson we will cover the Unified Software Process starting with *software architecture* where the goal is to lay the foundation on which to build successful and long lasting software systems.

Interview With Nenad Medvidovic

Why is software architecture important?

Architectural design decisions are really the principal design decisions in a software system because it could have long-term implications if not made correctly. Architectural erosion happens when software grows over time while the architecture of a software system is ignored or poorly designed to begin with.

What Is Software Architecture

What is **SWA (software architecture)**? Depends on the source but generally software architecture could be thought of as:

- Perry and Wolf: Composed of elements (the what), form (the how), and rationale (the why)
- Shaw and Garland:
    - A level of design that involves description of elements from which systems are built
    - Interactions among those elements
    - Patterns that guide their composition
    - Constraints on these patterns

General Definition Of SWA

A general definition of SWA: *a set of principal design decisions about the system.* The blueprint of a software system includes:

- Structural properties
- Behavioral properties
- Non-functional properties
- Interactions

There is also a temporal aspect of SWA since it should change over the lifetime of a software system.

Prescriptive Vs. Descriptive Architecture

A **prescriptive architecture** captures the design decisions made prior to the system's construction (as-conceived SWA). A **descriptive architecture** describes how the system has actually been built (as-implemented SWA).

Architectural Evolution

When a system evolves, ideally its prescriptive architecture should be modified first. However, in practice this almost never happens.

Architectural Degradation

Over time, two types of architectural degradation occurs for software systems with poor SWA:

1. **Architectural drift**: introduction of architectural design decisions orthogonal to a system's prescriptive architecture
2. **Architectural erosion**: introduction of architecture design decisions that violate a system's prescriptive architecture.

Architectural Recovery

What happens when an architecture drifts or erodes?

We have to take a path leading to **architectural recovery** which typically means to determine SWA from implementation and fix it. Developers should not simply *tweak* code to recover their SWA.

Architectural Elements

A SWA typically is not a monolith composition and interplay of different elements. SWA includes:

1. Processing elements
2. Data elements
3. Interaction elements
4. Components
5. Connectors
6. Configuration composed of components and connectors

Components - Connectors - Configurations

Regarding architectural components, connectors, and configurations:

A **component** is an architectural entity that:

1. Encapsulates a subset of the system's functionality and/or data
2. Restricts access to that subset via an explicitly defined interface

A **connector** is an architectural entity affecting and regulating interaction. Finally, a **configuration** is an association between components and connectors of a SWA.

Architectural Styles

An **architectural style** is a named collection of architectural design decisions applicable in a given context. Alternatively from Shaw and Garland, an architectural style:

> ...defines a family of systems in terms of a pattern of structural organization, a vocabulary of components and connectors, with constraints on how the can be combined.

Types Of Architectural Styles

There are many types of architectural styles in SWA:

1. Pipes and filters
2. Event-driven
3. Publish-subscribe
4. Client-server
5. P2P (Peer-to-peer)
6. REST (Representational State Transfer)

P2P Architectures

P2P architectures advocate decentralized resource sharing and discovery. E.g., Skype utilizing super nodes to support a decentralized architecture but still allowing users to communicate P2P.

Takeaway Message

Several takeaways from this lesson:

1. A great SWA is a ticket to success
2. A great SWA reflects deep understanding of the problem domain
3. A great SWA normally combines aspects of several simpler architectures

Section Quizzes

Architectural Recovery Quiz

Which of the following sentences is true?

- Architectural drift results in unnecessarily complex architectures

Architectural Design Quiz

What are ideal characteristics of an architectural design?

- Scalability
- Low coupling

Architectural Styles Quiz

Mark which architectural style(s) (given) characterizes the following systems:

1. *Android OS*: event-driven, publish-subscribe
2. *Skype*: client-server, peer-to-peer
3. *World-wide Web*: client-server, REST
4. *DropBox*: client-server

Lesson 9

In this lesson we will go through an example where a librarian gives a list of requirements to a software engineer. This is the same set of lectures as Lesson 08 (P2L5) - Library Exercise (UML)

A Tale Of Analysis And Design

To recap, the software analysis and design process is as follows:

1. Analyze requirements
2. Refine classes and attributes
3. Add attributes
4. Identify operations
5. Add relationships
6. Refine relationships
7. Refine the class diagram

Lesson 10

It can be difficult to decide on a design that will best fit our software system. Therefore, in this lesson we will look into *design patterns* which can help to provide reusability to our software system.

History Of Design Patterns

The history of design patterns could be summarized as:

- 1977: Christopher Alexander introduces the idea of patterns as successful solutions to problems
- 1987: Ward Cunningham and Ken Beck leverage Alexander's idea in the context of an OO language
- 1987: Eric Gamm's dissertation on importance of patterns and how to capture them
- 1992: Jim Coplien's book, *Advanced C++ Programming Styles and Idioms*
- 1994: Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) and their book, *Design Patterns: Elements of Reusable Object-Oriented Software*

Patterns Catalogue

There are five main categories of design patterns we may be concerned with:

1. **Fundamental**
2. **Creational**
3. **Structural**
4. **Behavioral**
5. **Concurrency**

We will cover some patterns from each category. For more information on patterns in each category see the Wiki page on software design patterns.

Pattern Format

The format of a design pattern is as follows:

- Name
- Intent
- Applicability
- Structure
- Sample code

There are more formats which include fields such as motivation, consequences, implementation, and related patterns. However, for the purpose of this class we will mainly focus on the five attributes above for each design pattern.

Factory Method Pattern

Below are properties of the factory method pattern:

- **Name**: factory method pattern
- **Intent**: allows for creating objects without specifying their class by invoking a factory method (i.e., a method whose main goal is to create class instances)
- Applicability

  :

  - Class can't anticipate the type of objects it must create
  - Class wants its subclasses to specify the type of objects it creates
  - Class needs control over the creation of its objects
- Structure

  : an example of factory method may include:

  - **Creator**: provides interface for factory method
  - **ConcreteCreator**: provides method for creating actual object
  - **Product**: object created by the factory method

See lecture for sample code example.

Strategy Pattern

Below are properties of the strategy pattern:

- **Name**: strategy patter
- **Intent**: allows for switching between different algorithms for accomplishing a task
- Applicability

  :

  - Different variants of an algorithm
  - Many related classes differ only in their behavior
- Structure

  : an example of strategy pattern may include:

  - **Context**: interface to outside world
  - **Algorithm (strategy)**: common interface for the different algorithms
  - **Concrete strategy**: actual implementation of the algorithm

See lecture for sample code example.

Other Common Patterns

Other patterns include:

- **Visitor**: a way of separating an algorithm from an object structure on which it operates
- **Decorator**: a wrapper that adds functionality to a class (stackable)
- **Iterator**: access elements of a collection without knowing underlying representation
- **Observer**: notify dependents when object changes
- **Proxy**: surrogate controls access to an object

Choosing A Pattern

There are several guidelines to choosing a design pattern:

1. Understand your design context
2. Examine the patterns catalog
3. Identify and study related patterns
4. Apply suitable pattern

However, there are pitfalls when selecting design patterns which may include (but are not limited to):

- Selecting wrong patterns
- Abusing patterns

We should always be careful to not rely too heavily on design patterns and ensure that whichever design pattern we select fits our problem.

Negative Design Patterns

In addition to design patterns, there are also negative design patterns, i.e., how not to design manage, etc. These are also called *anti-patterns* and *bad smells* as discussed in Christoper Alexander's book: A Pattern Language

Section Quizzes

Choosing A Pattern Quiz

Answer following questions:

1. *Imagine that you have to write a class that can have one instance only. Which of the design pattern that we've discussed is most appropriate*? Factory
2. *Imagine that you have to write a class that can have one instance only. Using one of the design patterns that we discussed in this lesson, write the code of a class with only one method (except for possible constructors) that satisfy this requirement*. The code should be as follows:

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}

    public static Singleton factory() {
        if (instance == null) {
           instance = new Singleton();
        }
        return instance;
    }
}
```

Lesson 11

In this lesson we will try to combine concepts from software architecture, low-level design, and design patterns with concepts from software engineering activities to discuss a unified software process.

History Of RUP (Rational Unified Process)

In 1997, Rational defined six best practices for modern software engineering:

1. Develop iteratively, focusing on risk
2. Manage requirements
3. Employ a component-based architecture
4. Model software usually
5. Continuously verify quality
6. Control chances

Key Features Of RUP

There are several features of RUP:

1. **Software process model**: includes order of phases and transition criteria
2. **Component-based**: includes software components and well-defined interfaces
3. **Tightly related to UML**: related notation and basic principles
4. **Distinguishing aspects include**: use-case driven, architecture-centric, and iterative and incremental

Use Case Driven

RUP is use case driven. A system performs a sequence of actions in response to user input. Use cases capture this interaction and answer the question: *what is the system suppose to do for each user*?

Architecture Centric

RUP is architecture centric. Use cases define the function and the architecture defines the form. The process for this may include:

1. Creating a rough outline of the system
2. Translating key use cases into sub-systems
3. Refining architecture using additional use cases

Iterative And Incremental

RUP is iterative and incremental. Each software project has numerous cycles and each cycle results in a product release. Each cycle is divided into four phases:

1. **Inception**: includes mainly business modeling activities
2. **Elaboration**: includes mainly requirements plus analysis and design activities
3. **Construction**: includes mainly analysis and design, implementation, test, and deployment activities
4. **Transition**: includes some testing and deployment activities

For each phase, there may be multiple iterations.

Iterations

Within each iteration the following activities may be observed:

1. Identify relevant use cases
2. Create design
3. Implement the design
4. Verify code against use cases
5. Release a product at the end

Section Quizzes

UML 1 Quiz

What is the difference between a use case and a use case model?

A use case model is a set of use cases.

UML 2 Quiz

What are use case diagrams used for?

- They can be used to prioritize requirements
- They can be used during requirements elicitation
- They can be used for test case design

Iterative Approach Quiz

What are the benefits of iterative approaches?

- Give developers early feedback
- Minimize risk of developing wrong system
- Accommodate evolving requirements

Lesson 12

In this lesson we will cover software testing which includes software verification and validation. We will also cover some techniques for software testing, TDD (test-driven development), and agile methods which will allow more flexibility in the previously discussed RUP activities.

General Concepts Introduction

Software is buggy. In fact, experts argue that software bugs cost the US economy around $60B a year. On average, there are about 1-5 bugs per 1000 lines of code. The idea of 100% correct mass-market software is impossible. Therefore, we need to verify the software as much as possible.

Failure, Fault, and Error

There are three main concepts we should know when testing software:

1. **Failure**: an observable incorrect behavior
2. **Fault**: incorrect code
3. **Error**: cause of a fault

Verification Approaches

How can we verify software? There are several approaches:

1. Testing

   : testing if a piece of software returns an expected output given an input

   - Pros: does not generate false positives
   - Cons: is incomplete
2. Static verification

   : considers all possible inputs

   - Pros: considers all program behaviors
   - Cons: generates false positives
3. Inspections

   : a manual and typically group activity that involves reviewing a piece of software for correctness

   - Pros: systematic and thorough
   - Cons: informal and subjective
4. Formal proofs of correctness

   : a proof (e.g., simulation that verifies physics or mathematics) that piece of software correctly implements what is specified in the specification document

   - Pros: strong guarantees
   - Cons: complex and expensive

Testing Introduction

Testing means to execute a program on a tiny sample of the input domain. More generally, testing is a:

- Dynamic technique
- Optimistic approximation

Testing Granularity Levels

There are different levels of testing:

1. **Unit**: testing one unit of software (e.g., if a variable is doubled)
2. **Integration**: testing a particular software feature (e.g., does the home page load the correct user data in order)
3. **System**: testing the whole software (e.g., do the various application features work?)
4. **Acceptance**: testing if the software system meets the customer's specifications
5. **Regression**: testing if the new features and unchanged features work as expected

Alpha And Beta Testing

We previously discussed developer testing but there are also different types of testing by organization:

- **Alpha**: testing done by users internal to the developing organization
- **Beta**: testing done by users external to te developing organization

Black And White Box Testing Introduction

There are also testing techniques such as:

- **Black-box testing**: testing a piece of software based on a description of the software (specification). The goal is to cover as much specified behavior as possible. However, it cannot reveal errors due to implementation details
- **White-box testing**: testing that looks at software implementation and is design to covered as much coded behavior as possible. However, it cannot reveal errors due to missing specifications

Section Quizzes

Failure, Fault, and Error 1 Quiz

Given the following code:

```
int doubleValue(int i) {
   int result;
   result = i*i;
   return result;
```

A call to doubleValue(3) returns 9. This is: a failure.

Failure, Fault, and Error 2 Quiz

Where is the fault that causes the failure in the program (provided previously)? Write the line number of the code that contains the fault.

In line 3 where `result = i*i;`.

Failure, Fault, and Error 3 Quiz

What is the error the caused the fault (given previously)?

The `result` should be the input multiplied by 2 not by itself. However, as a user we would not know what the error is, only the developer would know.

Verification Approaches Quiz

> 50% of my company employees are testers, and the rest spends 50% of their time testing.

Who said that?

Bill Gates when he was at Microsoft.

Lesson 13

In this lesson we will cover one of two testing methods: *black-box testing* which refers to functional testing of software.

Black-box Testing Overview

Black-box testing has to do with testing a system without knowing its internal details. Some advantages of this method of testing include:

- Focuses on the domain
- No need for the code -> early test design
- Catches logic defects
- Applicable at all granularity levels

Systematic Functional Testing Approach

How do we get from functional specifications to test cases? There are several steps:

1. **Identify independently testable features**
2. Identify relevant inputs

   :

   - We should **not** have to test all cases
   - We should **not** test random inputs
   - Instead we should identify partitions and select inputs from each
   - We could also look at boundary values (edge cases) for each partition
3. **Derive test cases specifications**
4. **Generate test cases**

Category Partition Method

The **Category-partition Method** (Ostrand and Balcer, CACM, June 1988) describes how to go from specifications to test cases in six steps:

1. **Identify independently testable features**
2. **Identify categories**: this involves identifying characteristics of each input element
3. **Partition categories into choices**: this involves identifying interesting cases (sub-domains)
4. Identify constraints among choices
   :
   - Eliminate meaningless combinations
   - Reduce the number of test cases
   - Three types of labels: `Property... If`, `Error`, and `Single`
5. Produce/evaluate test case specifications
   :
   - Can be automated
   - Produces test frames
6. Generate test cases from test case specifications
   :
   - Simple instantiation of frames
   - Produces set of concrete tests (final result)

Model Based Testing

Model-based testing fits into the previous functional testing approach like so:

1. Identify independently testable features
2. Identify relevant inputs *or* the model
3. Derive test cases specifications
4. Generate test cases

Finite State Machines

One of the model-based testing approaches utilizes **FSM (finite state machines)** which can be defined by the following components:

1. Nodes -> states
2. Edges -> transitions
3. Edge labels -> events and actions

To build a FSM from specifications, we can perform the following:

1. Identify system boundaries and input and output
2. Identify relevant states and transitions
3. Identify how a system can go from one state to another

Finite State Machines Considerations

There are some considerations for the FSM model-based testing approach:

1. Applicability
   :
   - Very general approach
   - State machines are readily available in UML
2. **Abstraction is key**
3. **Many other approaches**: e.g., decision tables, flow graphs, historical models, etc.

Section Quizzes

Overview 1 Quiz

```
printSum(int a, int b)
```

How many independently testable features do we have here?

We only have one feature, in this case `printSum()`.

Overview 2 Quiz

Identify three possible independently testable features for a spreadsheet.

- Sorting
- Plotting
- Saving

Test Data Selection Quiz

How long would it take to exhaustively test this function?

```
printSum(int a, int b)
```

This function takes two integers and prints the sum.

It would take hundreds of years.

Lesson 14

In *black-box* testing we tested a system without knowing the internal details of the software system. This lesson will examine *white-box* (structural) testing which involves the internal details of a software system as wells as benefits and drawbacks.

White-box Testing Overview

In **white-box testing**, the basic assumption is that executing the faulty statement is a necessary condition for revealing a fault.

Some advantages of white-box testing include:

- Can be measured objectively
- Can be measured automatically
- Can be used to compare test suites
- Allows for covering the coded behavior

There are also different kinds of white-box testing:

- Control-flow based
- Data-flow based

- Fault based

Statement Coverage

Statement coverage refers to the number of statements in the program (e.g., *expect a + b to be > 0*) and the coverage should measure the number of executed statements versus the total number of statements.

Statement coverage in practice is typically 80-90% with but not 100%.

Control Flow Graphs

Control flow graphs are ways to visualize the control flow of a particular block of code.

Branch Coverage

Similar to statement coverage, **branch coverage** refers to the number of branches in the program (a group of tests based on the control flow for a particular block of program) and the coverage should measure the number of executed branches versus the total number of branches.

Condition Coverage

Condition coverage refers to the individual conditions in the program and are measured based on the number of conditions that are both true *and* false versus the total number of conditions.

Modified Condition/Decision Coverage

MC/DC (modified condition/decision coverage) refers to testing important combinations of conditions and limiting test costs. This type of testing extends branch and decision coverage with the requirement that *each* condition should affect the decision outcome independently.

Section Quizzes

Coverage Criteria Intro 1 Quiz

Given the following code block:

```
printSum(int a, int b) {
   int result = a +b;
   if(result > 0)
      printcol("red", result);
   else if(result < 0)
      printcol("blue", result);
}
```

What are some possible test specifications that will satisfy some of the requirements we just saw? What constraint must be specified for line 4 and 6 to be executed?

1. Line 4: `a + b > 0`
2. Line 6: `a + b < 0`

Coverage Criteria Intro 2 Quiz

Implement the test cases matching the following specifications:

1. `a + b` needs to be positive and the result should be red
2. `a + b` needs to be negative and the result should be blue

Statement Coverage Quiz

Given statement coverage is mostly used in industry and "typical coverage" target is 80-90% why don't we aim at 100%?

That is not possible and would take too much time to reach that goal since almost always more lines of code are added to an active repository than can be covered.

Subsumption 1 Quiz

Does condition coverage imply branch coverage? No, branch coverage covers branches while condition coverage covers conditions.

Subsumption 2 Quiz

Does branch and condition coverage imply branch coverage? Yes, because it implies that branch and condition coverage needs to be met.

Branch And Condition Coverage Quiz

Add a test case (to provided code) to achieve 100% branch and condition coverage.

Set `x = 1` and `y = -1`.

Review 1 Quiz

Given the following code block:

```
int i;
read(i);
print(10 / (i - 3));
```

Test if test suite (1, -5), (-1, 2.5), (0, -3.333), does this test suite achieve path coverage? Yes, since there is only one path.

Review 2 Quiz

Referring to the previous path coverage test, does this test suite reveal the fault at line 3? No, because there is no input that causes it to `print(10/0)`.

Review 3 Quiz

Given the following code block:

```
int i = 0;
int j;
read(j);
if ((j > 5) && (i > 0))
   print(i);
```

Can you create a test suite to achieve statement coverage? No, because `i = 0` is always true

Lesson 15

The **agile development process** (also known as **test driven development** or **TDD**) is a software development process which is heavily based on testing. We will also look into two principles of agile software development commonly used in practice:

1. XP (extreme programming)
2. Scrum

Cost Of Change

In waterfall development processes the cost of change grows exponentially with time. How to combat this issue? Back in the day, one way was to discover errors early with upfront planning.

However, this was around 30 years ago, many things have changed since then and in general the software development process became much faster. Using modern tools, practice, and principles we might be able to *flatten* out the curve for the cost of change over time.

Agile Software Development

If cost is *flat* then upfront work becomes a *liability* and with ambiguity it may be good to delay, therefore there is value in waiting. There are six principles of agile software development that aim at flat cost:

1. Focus on the code
2. People over process
3. Iterative approach
4. Customer involvement
5. Expectation that requirements will change
6. Simplicity

XP Extreme Programming

XP is a lightweight methodology for small to medium size teams developing software in the face of vague or rapidly changing requirements. XP focuses on four main key points:

1. Lightweight
2. Humanistic
3. Discipline
4. Software Development

Additionally, XP embodies the following values and principles:

1. Communication
2. Simplicity
3. Feedback
4. Courage

In practice, XP advocates for:

1. Incremental planning

   : which could be broken down into the following:

   1. Select user stories for release
   2. Break stories into tasks
   3. Plan release
   4. Develop, integrate, and test
   5. Release software
   6. Evaluate system and iteration
2. Small releases

   : there many advantages of small releases:

   1. Deliver value quickly
   2. Rapid feedback
   3. Sense of accomplishment for the developments
   4. Reduces risk
   5. Quickly adapt to new requirements
3. Simple design

   : which advocates for design that:

   1. Is enough to meet the requirements
   2. Includes no duplicated functionality
   3. Has the fewest possible classes and methods
4. **Test first**: write tests early before developing a feature
5. **Refactoring**: restructuring code such that code is more clean and simple
6. **Pair programming**: a technique that leverages two developers where one may take on the role of programming and one may take on the role of strategizing
7. Continuous integration

   : process that advocates for continuously checking the following:

   1. Local tests
   2. Integration tests
   3. System tests
8. **On-site customer**: the customer is an actual member of the team (not very common in practice)

Testing Strategy

The testing strategy in agile is that testing is coded confidence. There are various types of tests we should implement for meaningful features and functionality:

- Unit tests
- System tests

Scrum Intro

Scrum is another agile development process similar to XP. There are various kinds of actors in a scrum setup:

1. **Product owner (customer)**: the person in charge of product intent
2. **Team**: group in charge of shipping features
3. **Scrum master**: the person in charge of overseeing the scrum processes and ensuring team fluidity

There are several of high-level scrum process:

1. Product backlog
2. Sprint planning
3. Sprint backlog
4. Daily scrum (iterative)
5. Sprint review and retrospective (iterative)
6. Potentially shippable product increment

Section Quizzes

Testing Strategy Quiz

Which of the following statements about Extreme Programming (XP) are true?

- XP follows the test driven development (TDD) paradigm
- XP is an iterative software development process

Lesson 15

In this lesson we will cover software refactoring in agile development which focuses on code understandability, maintainability, and design. We will also discuss concepts such as fully automated refactoring and *bad smells* which will allow us to determine when we should perform refactoring.

Software Refactoring Introduction

What is refactoring? **Refactoring** is an activity in software where the goal is to keep the program readable, understandable, and maintainable.

A key factor is that refactoring should be *behavior preserving*; any refactoring done to a particular piece of software will **not** change functionality. This could be checked but **not** guaranteed by testing.

Reasons To Refactor

Why should we refactor? We should refactor because of the following:

1. Requirements change
2. Design needs to be improved
3. Sloppiness/laziness creeping into software

History Of Refactoring

Refactoring is something programmers have always done. Refactoring is especially important for OO languages and is an increasingly popular and important part of agile software development.

Types Of Refactoring

There are many types of refactoring but for this class we will cover the following:

- **Collapse hierarchy**: useful when a superclass and a subclass are too similar
- **Consolidate conditionals**: useful when a set of conditionals has the same result
- **Decompose conditionals**: useful when a conditional statement is particularly complex
- **Extract method**: useful when a cohesive code fragment exists in a large method
- **Extract class**: useful when a class is doing the work of two classes
- **Inline class**: useful when a class is not doing much

Refactoring Risks

Refactoring is a powerful tool but there are several drawbacks:

- May introduce subtle faults
- Should not be abused
- Should be used carefully on systems in production

Cost Of Refactoring

The cost of refactoring depends on a several factors:

- Manual work
- Test development and maintenance
- Documentation maintenance

When Not To Refactor

We should not refactor when:

- Code is broken
- A deadline is close
- There is no reason to

Bad Smells

Bad smells are symptoms in the code which may indicate deeper problems. There are many types of bad smells. Below are some of the common bad smells and the solution that may be applied to resolve them:

- Duplicated code -> solved by extract method
- Long method -> extract method, decomposed conditional, etc.
- Large class -> extract class (or subclass)
- Shotgun surgery -> move method/field, inline class, etc.
- Feature envy -> extract method, move method

Section Quizzes

Introduction Quiz

Why can't testing guarantee that a refactoring is behavior preserving?

Because testing is inherently incomplete.

Extract Method Refactoring Quiz

When is is appropriate to apply refactoring "extract method"?

- When there is duplicated code in two or more methods
- When a method is highly coupled with a class other than the one where it is defined

Bad Smell Quiz

Which of the following can be considered to be "bad smells" in the context of refactoring?

- Method `m()` in class `C` is very long
- Every time we modify method `m1()`, we also need to modify method `m2()`