CS2113T Cheatsheet AY21/22 Sem 1

by Richard Willie

Software Engineering

Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

Pros

1. The sheer joy of making things.
2. The pleasure of making things that are useful to other people.
3. The fascination of complex puzzle-like objects of interlocking moving parts and watching them work in subtle cycles, playing out the consequences of principles built in from the beginning.
4. The joy of always learning, which springs from the nonrepeating nature of the task.
5. The delight of working in such a tractable medium.

Cons

1. One must perform perfectly.
2. Other people set one's objectives, provide one's resources, and furnish one's information. One rarely controls the circumstances of his work, or even its goal.
3. Designing grand concepts is fun; finding nitty little bugs is just work. With any creative activity come dreary hours of tedious, painstaking labor, and programming is no exception.
4. Debugging has a linear convergence, or worse, where one somehow expects a quadratic sort of approach to the end. So testing drags on and on, the last difficult bugs taking more time to find than the first.
5. The product over which one has labored so long appears to be obsolete upon (or before) completion.

Object-Oriented Progamming

Object Oriented Programming (OOP) views the world as a network of interacting objects.

The four pillars of object-oriented programming are:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Objects

- Every object has both **state (data) and behavior (operations on data)**.
- Every object has **an interface and an implementation**.
  - Interface of an object consists of **how other objects interact with it** i.e., what other objects can do to that object.
  - Implementation consist of **internals of the object that facilitate the interactions** but not visible to other objects.

Objects as Abstractions

The concept of Objects in OOP is an **abstraction mechanism** because it allows us to abstract away the lower level details and work with bigger granularity entities i.e. ignore details of data formats and the method implementation details and work at the level of objects.

Encapsulation of Objects

An object is an encapsulation of some data and related behavior in terms of two aspects:

1. **The packaging aspect:** An object packages data and related behavior together into one self-contained unit.
2. **The information hiding aspect:** The data in an object is hidden from the outside world and are only accessible using the object's interface.

Classes

A class contains instructions for creating a specific kind of objects.

Class Level Members

- Variables whose value is shared by all instances of a class are called **class-level attributes**.
- Methods that are called using the class instead of a specific instance are called **class-level methods**.

Class-level attributes and methods are collectively called **class-level members** (also called static members sometimes because some programming languages use the keyword static to identify class-level members). **They are to be accessed using the class name rather than an instance of the class**.

Enumerations

An Enumeration is **a fixed set of values that can be considered as a data type**. An enumeration is often useful when using a regular data type such as int or String would allow invalid values to be assigned to a variable.

Associations

Connections between objects are called associations.

- Associations in an object structure can change over time.
- Associations among objects can be generalized as associations between the corresponding classes too.

Navigability

When two classes are linked by an association, it does not necessarily mean the two objects taking part in an instance of the association knows about (i.e., has a reference to) each other. **The concept of which object in the association knows about the other object is called navigability**.

Navigability can be **unidirectional or bidirectional**.

Multiplicity

Multiplicity is the aspect of an OOP solution that dictates how many objects take part in each association.

Implementing multiplicity

- A normal instance-level variable gives us a 0..1 multiplicity (also called optional associations) because a variable can hold a reference to a single object or null.

```
class Logic {
Minefield minefield;
    // ...
}

class Minefield {
    //...
}
```

- A variable can be used to implement a 1 multiplicity too (also called compulsory associations).

```
class Logic {
    ConfigGenerator cg = new ConfigGenerator();
    // ...
}
```

- Bidirectional associations require matching variables in both classes.

```
class Foo {
    Bar bar;
    // ...
}

class Bar {
    Foo foo;
    // ...
}
```

- To implement other multiplicities, choose a suitable data structure such as Arrays, ArrayLists, HashMaps, Sets, etc.

```
class Minefield {
    Cell[][] cell;
    // ...
}
```

Dependencies

A dependency is a need for one class to depend on another **without having a direct association** in the same direction.

In the code below, Foo has a dependency on Bar but it is not an association because it is only a transient interaction and there is no long term relationship between a Foo object and a Bar object. i.e. the Foo object does not keep the Bar object it receives as a parameter.

```
class Foo {
    int calculate(Bar bar) {
        return bar.getValue();
    }
}

class Bar {
    int value;

    int getValue() {
        return value;
    }
}
```

Composition

A composition is an association that represents **a strong whole-part relationship**. When the whole is destroyed, parts are destroyed too i.e., the part should not exist without being attached to a whole.

- Composition also implies that there cannot be cyclical links.
- Whether a relationship is a composition can depend on the context.
- A common use of composition is when parts of a big class are carved out as smaller classes for the ease of managing the internal design.
- Cascading deletion alone is not sufficient for composition.
- Identifying and keeping track of composition relationships in the design has benefits such as helping to maintain the data integrity of the system.

Composition is implemented using a normal variable. If correctly implemented, the 'part' object will be deleted when the 'whole' object is deleted. Ideally, the 'part' object may not even be visible to clients of the 'whole' object.

In this code, the Email has a composition type relationship with the Subject class, in the sense that the subject is part of the email.

```
class Email {
    private Subject subject;
    // ...
}
```

Aggregation

Aggregation represents **a container-contained relationship**. It is a weaker relationship than composition.

Implementation is similar to that of composition except the containee object can exist even after the container object is deleted.

In the code below, there is an aggregation association between the Team class and the Person class in that a Team contains a Person object who is the leader of the team.

```
class Team {
    Person leader;

    void setLeader(Person p) {
        leader = p;
    }
}
```

Association Classes

An association class represents **additional information about an association**. It is a normal class but plays a special role from a design point of view.

There is no special way to implement an association class. It can be implemented as a normal class that has variables to represent the endpoint of the association it represents.

In the code below, the Transaction class is an association class that represents a transaction between a Person who is the seller and another Person who is the buyer.

```
class Transaction {
    // all fields are compulsory
    Person seller;
    Person buyer;
    Date date;
    String receiptNumber;

    Transaction(Person seller, Person buyer, Date date, String
        receiptNumber) {
        // set fields
    }
}
```

Inheritance

We can think of inheritance as a model for the "is a" relationship between two entities.

- A superclass is said to be more general than the subclass.
- Applying inheritance on a group of similar classes can result in the common parts among classes being extracted into more general classes.
- Inheritance implies the derived class can be considered as a sub-type of the base class (and the base class is a super-type of the derived class), resulting in an is a relationship.
- Inheritance relationships through a chain of classes can result in inheritance hierarchies (aka inheritance trees).
- Multiple Inheritance is when a class inherits directly from multiple classes. Multiple inheritance among classes is allowed in some languages (e.g., Python, C++) but not in other languages (e.g., Java, C#).

Overriding

Method overriding is when **a sub-class changes the behavior inherited from the parent class by re-implementing the method**. Overridden methods have the same name, same type signature, and same return type.

Overloading

Method overloading is when there are **multiple methods with the same name but different type signatures**. Overloading is used to indicate that multiple operations do similar things but take different parameters.

Interfaces

An interface is **a behavior specification** i.e. a collection of method specifications. If a class implements the interface, it means the class is able to support the behaviors specified by the said interface.

A class implementing an interface results in an "is a" relationship, just like in class inheritance.

Abstract Classes

You can declare a class as abstract when a class is merely a representation of commonalities among its subclasses in which case it does not make sense to instantiate objects of that class.

- A class declared as an abstract class cannot be instantiated, but it can be subclassed.
- An abstract method is a method signature without a method implementation.
- A class that has an abstract method becomes an abstract class because the class definition is incomplete (due to the missing method body) and it is not possible to create objects using an incomplete class definition.

What is the difference between a Class, an Abstract Class, and an Interface?

- An interface is a behavior specification with no implementation.
- A class is a behavior specification + implementation.
- An abstract class is a behavior specification + a possibly incomplete implementation.

Substitutability

Every instance of a subclass is an instance of the superclass, but not vice-versa. As a result, inheritance allows substitutability: the ability to substitute a child class object where a parent class object is expected.

```
Staff staff = new AcademicStaff(); // OK
```

```
Staff staff;
AcademicStaff academicStaff = staff; // Not OK
```

Dynamic and Static Binding

Dynamic binding (aka late binding): a mechanism where method calls in code are resolved at runtime, rather than at compile time.

Overridden methods are resolved using dynamic binding, and therefore resolves to the implementation in the actual type of the object.

```
void adjustSalary(int byPercent) {
    for (Staff s: staff) {
        s.adjustSalary(byPercent);
    }
}
```

At runtime s can receive an object of any subclass of Staff. That means the adjustSalary(int) operation of the actual subclass object will be called. If the subclass does not override that operation, the operation defined in the superclass (in this case, Staff class) will be called.

Static binding (aka early binding): When a method call is resolved at compile time.

In contrast, **overloaded methods are resolved using static binding**.
Note how the constructor is overloaded in the class below. The method call new Account() is bound to the first constructor at compile time.

```
class Account {
    Account() {
        // Signature: ()
        // ...
    }

    Account(String name, String number, double balance) {
        // Signature: (String, String, double)
        // ...
    }
}
```

Polymorphism

Polymorphism allows you to write code targeting superclass objects, use that code on subclass objects, and achieve possibly different results based on the actual class of the object.

Assume classes Cat and Dog are both subclasses of the Animal class. You can write code targeting Animal objects and use that code on Cat and Dog objects, achieving possibly different results based on whether it is a Cat object or a Dog object. Some examples:

- Declare an array of type Animal and still be able to store Dog and Cat objects in it.
- Define a method that takes an Animal object as a parameter and yet be able to pass Dog and Cat objects to it.
- Call a method on a Dog or a Cat object as if it is an Animal object (i.e., without knowing whether it is a Dog object or a Cat object) and get a different response from it based on its actual class e.g., call the Animal class's method speak() on object a and get a "Meow" as the return value if a is a Cat object and "Woof" if it is a Dog object.

Three concepts combine to achieve polymorphism: substitutability, operation overriding, and dynamic binding.

- **Substitutability:** Because of substitutability, you can write code that expects objects of a parent class and yet use that code with objects of child classes. That is how polymorphism is able to treat objects of different types as one type.
- **Overriding:** To get polymorphic behavior from an operation, the operation in the superclass needs to be overridden in each of the subclasses. That is how overriding allows objects of different subclasses to display different behaviors in response to the same method call.
- **Dynamic binding:** Calls to overridden methods are bound to the implementation of the actual object's class dynamically during the runtime. That is how the polymorphic code can call the method of the parent class and yet execute the implementation of the child class.

Requirements

A software requirement specifies a need to be fulfilled by the software product.

Requirements come from **stakeholders**. Stakeholder is a party that is potentially affected by the software project. e.g. users, sponsors, developers, interest groups, government agencies, etc.

Identifying requirements is often not easy. For example, stakeholders may not be aware of their precise needs, may not know how to communicate their requirements correctly, may not be willing to spend effort in identifying requirements, etc.

Non-Functional Requirements

Requirements can be divided into two in the following way:

1. Functional requirements specify what the system should do.
2. Non-functional requirements specify the constraints under which the system is developed and operated

You may have to spend an extra effort in digging NFRs out as early as possible because,

1. NFRs are easier to miss e.g., stakeholders tend to think of functional requirements first
2. sometimes NFRs are critical to the success of the software. E.g. A web application that is too slow or that has low security is unlikely to succeed even if it has all the right functionality.

Prioritizing Requirements

Requirements can be prioritized based on the importance and urgency, while keeping in mind the constraints of schedule, budget, staff resources, quality goals, and other constraints. Some requirements can be discarded if they are considered 'out of scope'.

Quality of Requirements

Here are some characteristics of well-defined requirements:

- Unambiguous
- Testable (verifiable)
- Clear (concise, terse, simple, precise)
- Correct
- Understandable
- Feasible (realistic, possible)
- Independent
- Atomic
- Necessary
- Implementation-free (i.e. abstract)

Besides these criteria for individual requirements, the set of requirements as a whole should be

- Consistent
- Non-redundant
- Complete

Gathering Requirements

Brainstorming

Brainstorming is a group activity designed to generate a large number of diverse and creative ideas for the solution of a problem.

In a brainstorming session there are no "bad" ideas. **The aim is to generate ideas; not to validate them**. Brainstorming encourages you to "think outside the box" and put "crazy" ideas on the table without fear of rejection.

User Surveys
Surveys can be used to solicit responses and opinions from a large number of stakeholders regarding a current product or a new product.

Observation
Observing users in their natural work environment can uncover product requirements.

Interviews
Interviewing stakeholders and domain experts can produce useful information about project requirements.

Focus Groups
Focus groups are a kind of informal interview within an interactive group setting.

Prototyping
A prototype is a mock up, a scaled down version, or a partial system constructed

- to get users' feedback.
- to validate a technical concept (a "proof-of-concept" prototype).
- to give a preview of what is to come, or to compare multiple alternatives on a small scale before committing fully to one alternative.
- for early field-testing under controlled conditions.

Prototyping can uncover requirements, in particular, those related to how users interact with the system. UI prototypes or mock ups are often used in brainstorming sessions, or in meetings with the users to get quick feedback from them.

Product Surveys
Studying existing products can unearth shortcomings of existing solutions that can be addressed by a new product.

Specifying Requirements
Prose
A textual description (i.e. prose) can be used to describe requirements. Prose is especially useful when describing abstract ideas such as the vision of a product.

Avoid using lengthy prose to describe requirements; they can be hard to follow.

Feature List
Feature list is a list of features of a product grouped according to some criteria such as aspect, priority, order of delivery, etc.

User Stories
User stories are short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.
A common format for writing user stories is:

As a {user type/role} I can {function} so that {benefit}

The {benefit} can be omitted if it is obvious.

As a user, I can login to the system ~~so that I can access my data~~.

Usage

- User stories capture user requirements in a way that is convenient for scoping, estimation, and scheduling.
- User stories are short and written in natural language, not in a formal language.
- User stories are not detailed enough to tell us exact details of the product.
- User stories can capture non-functional requirements too because even NFRs must benefit some stakeholder.

Use Cases
- A use case describes an interaction between the user and the system for a specific functionality of the system.
- Use cases capture the functional requirements of a system.

Glossary
A glossary serves to ensure that all stakeholders have a common understanding of the noteworthy terms, abbreviations, acronyms etc.

Supplementary Requirements
A supplementary requirements section can be used to capture requirements that do not fit elsewhere. Typically, this is where most Non-Functional Requirements will be listed.

Software Design
Software design has two main aspects:

- **Product/external design: designing the external behavior of the product to meet the users' requirements.** This is usually done by product designers with input from business analysts, user experience experts, user representatives, etc.
- **Implementation/internal design: designing how the product will be implemented to meet the required external behavior.** This is usually done by software architects and software engineers.

Design Fundamentals
Abstraction
Abstraction is a technique for dealing with complexity. It works by establishing a level of complexity we are interested in, and suppressing the more complex details below that level. The guiding principle of abstraction is that only details that are relevant to the current perspective or the task at hand need to be considered.

- Data abstraction: abstracting away the lower level data items and thinking in terms of bigger entities.
- Control abstraction: abstracting away details of the actual control flow to focus on tasks at a higher level.
- Abstraction can be applied repeatedly to obtain progressively higher levels of abstraction.
- Abstraction is a general concept that is not limited to just data or control abstractions.

Coupling
Coupling is a measure of the degree of dependence between components, classes, methods, etc. Low coupling indicates that a component is less dependent on other components. **High coupling (aka tight coupling or strong coupling) is discouraged** due to the following disadvantages:

- **Maintenance is harder** because a change in one module could cause changes in other modules coupled to it (i.e. a ripple effect).
- **Integration is harder** because multiple components coupled with each other have to be integrated at the same time.
- **Testing and reuse of the module is harder** due to its dependence on other modules.

Some examples of coupling: A is coupled to B if,

- A has access to the internal structure of B (this results in a very high level of coupling)
- A and B depend on the same global variable
- A calls B
- A receives an object of B as a parameter or a return value
- A inherits from B
- A and B are required to follow the same data format or communication protocol

Cohesion

Cohesion is a measure of how strongly-related and focused the various responsibilities of a component are. A highly-cohesive component keeps related functionalities together while keeping out all other unrelated things.

Higher cohesion is better. Disadvantages of low cohesion (aka weak cohesion):

- Lowers the understandability of modules as it is difficult to express module functionalities at a higher level.
- Lowers maintainability because a module can be modified due to unrelated causes (reason: the module contains code unrelated to each other) or many modules may need to be modified to achieve a small change in behavior (reason: because the code related to that change is not localized to a single module).
- Lowers reusability of modules because they do not represent logical units of functionality.

Cohesion can be present in many forms. Some examples:

- Code related to a single concept is kept together, e.g. the Student component handles everything related to students.
- Code that is invoked close together in time is kept together, e.g. all code related to initializing the system is kept together.
- Code that manipulates the same data structure is kept together, e.g. the GameArchive component handles everything related to the storage and retrieval of game sessions.

Modeling

- A model is a representation of something else.
- A model provides a simpler view of a complex entity because a model captures only a selected aspect. This omission of some aspects implies models are abstractions.
- Multiple models of the same entity may be needed to capture it fully.

In software development, models are useful in several ways:

- To analyze a complex entity related to software development.
- To communicate information among stakeholders. Models can be used as a visual aid in discussions and documentation.
- As a blueprint for creating software. Models can be used as instructions for building software.

Modeling Structures

OO Structures

An OO solution is basically a network of objects interacting with each other. Therefore, it is useful to be able to model how the relevant objects are 'networked' together inside a software i.e. how the objects are connected together.

Software Architecture

The software architecture shows the **overall organization of the system** and can be viewed as a **very high-level design**.

The architecture is typically designed by the **software architect**, who provides the technical vision of the system and makes high-level (i.e. architecture-level) technical decisions about the project.

Architecture Diagrams

Architecture diagrams are **free-form diagrams**.

While architecture diagrams have no standard notation, try to follow these basic guidelines when drawing them.

- Minimize the variety of symbols. If the symbols you choose do not have widely-understood meanings e.g. A drum symbol is widely-understood as representing a database, explain their meaning.
- Avoid the indiscriminate use of double-headed arrows to show interactions between components.

Architecture Styles

Software architectures follow various high-level styles (aka architectural patterns), just like how building architectures follow various architecture styles.

N-tier Architectural Style

In the n-tier style, higher layers make use of services provided by lower layers. Lower layers are independent of higher layers. Other names: multi-layered, layered.

Client-server Architectural Style

The client-server style has at least one component playing the role of a server and at least one client component accessing the services of the server. This is an architectural style used often in distributed applications.

Software Design Patterns

Design pattern ia an elegant reusable solution to a commonly recurring problem within a given context in software design.

Singleton Pattern

Context

Certain classes should have no more than just one instance (e.g. the main controller class of the system). These single instances are commonly known as singletons.

Problem

A normal class can be instantiated multiple times by invoking the constructor.

Solution

Make the constructor of the singleton class private, because a public constructor will allow others to instantiate the class at will. Provide a public class-level method to access the single instance.

Implementation

```
class Logic {
    private static Logic theOne = null;

    private Logic() {
        // ...
    }

    public static Logic getInstance() {
        if (theOne == null) {
            theOne = new Logic();
        }
        return theOne;
```

```
    }
}
```

Notes:

- The constructor is private, which prevents instantiation from outside the class.
- The single instance of the singleton class is maintained by a private class-level variable.
- Access to this object is provided by a public class-level operation getInstance() which instantiates a single copy of the singleton class when it is executed for the first time. Subsequent calls to this operation return the single instance of the class.

If Logic was not a Singleton class, an object is created like this:

```
Logic m = new Logic();
```

But now, the Logic object needs to be accessed like this:

```
Logic m = Logic.getInstance();
```

Evaluation

Pros:

- easy to apply
- effective in achieving its goal with minimal extra work
- provides an easy way to access the singleton object from anywhere in the code base

Cons:

- The singleton object acts like a global variable that increases coupling across the code base.
- In testing, it is difficult to replace Singleton objects with stubs (static methods cannot be overridden).
- In testing, singleton objects carry data from one test to another even when you want each test to be independent of the others.

Given that there are some significant cons, it is recommended that you apply the Singleton pattern when, in addition to requiring only one instance of a class, there is a risk of creating multiple objects by mistake, and creating such multiple objects has real negative consequences.

Facade Pattern

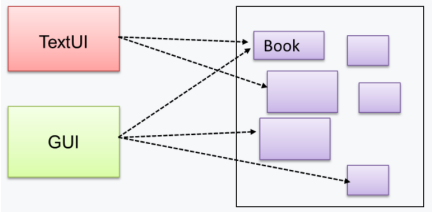
Context

Components need to access functionality deep inside other components.

The UI component of a Library system might want to access functionality of the Book class contained inside the Logic component.

Problem

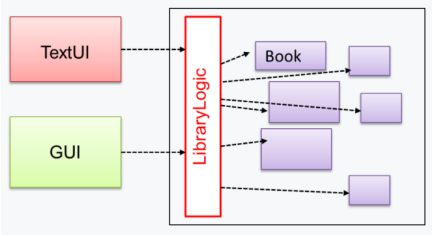
Access to the component should be allowed without exposing its internal details. e.g. the UI component should access the functionality of the Logic component without knowing that it contains a Book class within it.



Solution

Include a Facade class that sits between the component internals and users of the component such that all access to the component happens through the Facade class.

The following class diagram applies the Facade pattern to the Library System example. The LibraryLogic class is the Facade class.



Design Approaches

- In a smaller system, the design of the entire system can be shown in one place.
- The design of bigger systems needs to be done/shown at multiple levels.

Code Quality

Refer to *SE Textbook Code Quality Chapter* for more details.

Refactoring

The process of improving a program's internal structure in small steps **without modifying its external behavior** is called refactoring.

- **Refactoring is not rewriting:** Discarding poorly-written code entirely and re-writing it from scratch is not refactoring because refactoring needs to be done in small steps.
- **Refactoring is not bug fixing:** By definition, refactoring is different from bug fixing or any other modifications that alter the external behavior (e.g. adding a feature) of the component in concern.

Refactoring, even if done with the aid of an IDE, may still result in regressions. Therefore, each small refactoring should be followed by regression testing.

A comprehensive list of refactorings is available at the *Catalog of Refactoring*.

Documentation

Refer to *SE Textbook Documentation Chapter* for more details.

Error Handling

Exceptions

Exceptions are used to deal with 'unusual' but not entirely unexpected situations that the program might encounter at runtime.

Most languages allow code that encountered an "exceptional" situation to encapsulate details of the situation in an Exception object and throw/raise that object so that another piece of code can catch it and deal with it.

How exceptions are typically handled:

- When an error occurs at some point in the execution, the code being executed creates an exception object and hands it off to the runtime system.
- After a method throws an exception, the runtime system attempts to find something to handle it in the call stack.
- The exception handler chosen is said to catch the exception.

Advantages of exception handling in this way:

- The ability to propagate error information through the call stack.
- The separation of code that deals with 'unusual' situations from the code that does the 'usual' work.

Assertions

Assertions are used to define assumptions about the program state so that the runtime can verify them. An assertion failure indicates a possible bug in the code because the code has resulted in a program state that violates an assumption about how the code should behave.

If the runtime detects an assertion failure, it typically takes some drastic action such as terminating the execution with an error message. This is because an assertion failure indicates a possible bug and the sooner the execution stops, the safer it is.
When to use assertions?

- It is recommended that assertions be used liberally in the code. Their impact on performance is considered low and worth the additional safety they provide.
- Do not use assertions to do work because assertions can be disabled. If not, your program will stop working when assertions are not enabled.
- Assertions are suitable for verifying assumptions about Internal Invariants, Control-Flow Invariants, Preconditions, Postconditions, and Class Invariants.

Exceptions and assertions are two complementary ways of handling errors in software but they serve different purposes. Therefore, both assertions and exceptions should be used in code.

- **The raising of an exception indicates an unusual condition created by the user** (e.g. user inputs an unacceptable input) or the environment (e.g., a file needed for the program is missing).
- **An assertion failure indicates the programmer made a mistake in the code** (e.g., a null value is returned from a method that is not supposed to return null under any circumstances).

Logging

Logging is the deliberate recording of certain information during a program execution for future reference.

Integration

Combining parts of a software product to form a whole is called integration. It is also one of the most troublesome tasks and it rarely goes smoothly.

Build Automation

Build automation tools automate the steps of the build process, usually by means of build scripts. Some build tools also serve as dependency management tools (e.g., Maven and Gradle).

Continuous Integration and Continuous Deployment

An extreme application of build automation is called continuous integration (CI) in which integration, building, and testing happens automatically after each code change.

A natural extension of CI is Continuous Deployment (CD) where the changes are not only integrated continuously, but also deployed to end-users at the same time.

Some examples of CI/CD tools: Travis, Jenkins, Appveyor, CircleCI, GitHub Actions

Reuse

By reusing tried-and-tested components, the robustness of a new software system can be enhanced while reducing the manpower and time requirement. Reusable components come in many forms; it can be reusing a piece of code, a subsystem, or a whole software.

There are costs associated with reuse. Here are some:

- The reused code **may be an overkill** (think using a sledgehammer to crack a nut), increasing the size of, and/or degrading the performance of, your software.
- The reused software **may not be mature/stable enough** to be used in an important product. That means the software can change drastically and rapidly, possibly in ways that break your software.
- Non-mature software has the **risk of dying off** as fast as they emerged, leaving you with a dependency that is no longer maintained.
- The license of the reused software (or its dependencies) **restrict how you can use/develop your software**.
- The reused software **might have bugs, missing features, or security vulnerabilities** that are important to your product, but not so important to the maintainers of that software, which means those flaws will not get fixed as fast as you need them to.
- **Malicious code can sneak into your product** via compromised dependencies.

APIs

An Application Programming Interface (API) specifies the interface through which other programs can interact with a software component. It is a contract between the component and its clients.

When developing large systems, if you define the API of each component early, the development team can develop the components in parallel because the future behavior of the other components are now more predictable.

Quality Assurance

Software Quality Assurance (QA) is the process of ensuring that the software being built has the required levels of quality.

Validation vs Verification

Quality Assurance = Validation + Verification
QA involves checking two aspects:

1. **Validation:** are you building the **right system** i.e., are the requirements correct?
2. **Verification:** are you building the **system right** i.e., are the requirements implemented correctly?

Whether something belongs under validation or verification is not that important. What is more important is that both are done, instead of limiting to only verification (i.e., remember that the requirements can be wrong too).

Code Reviews

Code review is the systematic examination of code with the intention of finding where the code can be improved.
Reviews can be done in various forms. Some examples below:

- Pull Request reviews
- In pair programming
- Formal inspections

Advantages of code review over testing:

- It can detect functionality defects as well as other problems such as coding standard violations.
- It can verify non-code artifacts and incomplete code.
- It does not require test drivers or stubs.

Disadvantages:

- It is a manual process and therefore, error prone.

Static Analysis

- Static analysis of code can find useful information such as unused variables, unhandled exceptions, style errors, and statistics.
- The term static in static analysis refers to the fact that the code is analyzed without executing the code. In contrast, dynamic analysis requires the code to be executed to gather additional information about the code e.g., performance characteristics.
- Higher-end static analysis tools (static analyzers) can perform more complex analysis such as locating potential bugs, memory leaks, inefficient code structures, etc.

Linters are a subset of static analyzers that specifically aim to locate areas where the code can be made 'cleaner'.

Formal Verification

Formal verification uses mathematical techniques to prove the correctness of a program.
Advantages:

- Formal verification can be used to prove the absence of errors. In contrast, testing can only prove the presence of errors, not their absence.

Disadvantages:

- It only proves the compliance with the specification, but not the actual utility of the software.
- It requires highly specialized notations and knowledge which makes it an expensive technique to administer. Therefore, formal verifications are more commonly used in safety-critical software such as flight control systems.

Testing

Testing: Operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.

Testability is an indication of how easy it is to test an SUT.

Testing Types

Unit Testing

Unit testing: testing individual units (methods, classes, subsystems, ...) to ensure each piece works correctly.

A proper unit test requires the unit to be tested **in isolation** so that bugs in the dependencies cannot influence the test i.e. bugs outside of the unit should not affect the unit tests.

Stubs can isolate the SUT from its dependencies.

A stub has the same interface as the component it replaces, but its implementation is so simple that it is unlikely to have any bugs. It mimics the responses of the component, but only for a limited set of predetermined inputs. That is, it does not know how to respond to any other inputs. Typically, these mimicked responses are hard-coded in the stub rather than computed or retrieved from elsewhere, e.g. from a database.

Integration Testing

Integration testing: testing whether different parts of the software work together (i.e. integrates) as expected. Integration tests aim to discover bugs in the 'glue code' related to how components interact with each other. These bugs are often the result of misunderstanding what the parts are supposed to do vs what the parts are actually doing.

Integration testing is not simply a case of repeating the unit test cases using the actual dependencies (instead of the stubs used in unit testing). Instead, integration tests are additional test cases that focus on the interactions between the parts. In practice, developers often use a hybrid of unit+integration tests to minimize the need for stubs.

System Testing

System testing: take the whole system and test it against the system specification.
System testing is typically done by a testing team (also called a QA team). **System test cases are based on the specified external behavior of the system.** Sometimes, system tests go beyond the bounds defined in the specification. This is useful when testing that the system fails 'gracefully' when pushed beyond its limits.

System testing includes testing against non-functional requirements too. Here are some examples:

- Performance testing – to ensure the system responds quickly.
- Load testing (also called stress testing or scalability testing) – to ensure the system can work under heavy load.
- Security testing – to test how secure the system is.
- Compatibility testing, interoperability testing – to check whether the system can work with other systems.
- Usability testing – to test how easy it is to use the system.
- Portability testing – to test whether the system works on different platforms.

Alpha and Beta Testing

- Alpha testing is performed by the users, under controlled conditions set by the software development team.
- Beta testing is performed by a selected subset of target users of the system in their natural work setting.

Dogfooding

Eating your own dog food (aka dogfooding), is when creators use their own product so as to test the product.

Developer Testing

Developer testing is the testing done by the developers themselves as opposed to professional testers or end-users.
Delaying testing until the full product is complete has a number of disadvantages:

- Locating the cause of a test case failure is difficult due to a large search space; in a large system, the search space could be millions of lines of code, written by hundreds of developers! The failure may also be due to multiple inter-related bugs.
- Fixing a bug found during such testing could result in major rework, especially if the bug originated from the design or during requirements specification i.e. a faulty design or faulty requirements.
- One bug might 'hide' other bugs, which could emerge only after the first bug is fixed.
- The delivery may have to be delayed if too many bugs are found during testing.

Exploratory vs Scripted Testing

Here are two alternative approaches to testing a software:

- **Scripted testing:** First write a set of test cases based on the expected behavior of the SUT, and then perform testing based on that set of test cases.
- **Exploratory testing:** Devise test cases on-the-fly, creating new test cases based on the results of the past test cases.

Exploratory testing is also known as reactive testing, error guessing technique, attack-based testing, and bug hunting.

Acceptance Testing

Acceptance testing (aka User Acceptance Testing (UAT): test the system to ensure it meets the user requirements.

Acceptance testing comes after system testing. Similar to system testing, acceptance testing involves testing the whole system.

System Testing	Acceptance Testing
Done against the system specification	Done against the requirements specification
Done by testers of the project team	Done by a team that represents the customer
Done on the development environment or a test bed	More focus on positive test cases
Both negative and positive test cases	More focus on positive test cases

Requirements specification	System specification
limited to how the system behaves in normal working conditions	can also include details on how it will fail gracefully when pushed beyond limits, how to recover, etc. specification
written in terms of problems that need to be solved (e.g. provide a method to locate an email quickly)	written in terms of how the system solves those problems (e.g. explain the email search feature)
specifies the interface available for intended end-users	could contain additional APIs not available for end-users (for the use of developers/testers)

Passing system tests does not necessarily mean passing acceptance testing.

Regression Testing

When you modify a system, the modification may result in some unintended and undesirable effects on the system. Such an effect is called a **regression**.

Regression testing is the re-testing of the software to detect regressions. Note that to detect regressions, you need to retest all related components, even if they had been tested before.

Regression testing is more effective when it is done frequently, after each small change. However, doing so can be prohibitively expensive if testing is done manually. Hence, **regression testing is more practical when it is automated**.

Test Coverage

Test coverage is a metric used to measure the extent to which testing exercises the code i.e., how much of the code is 'covered' by the tests.

Here are some examples of different coverage criteria:

- **Function/method coverage:** based on functions executed e.g., testing executed 90 out of 100 functions.
- **Statement coverage:** based on the number of lines of code executed e.g., testing executed 23k out of 25k LOC.
- **Decision/branch coverage:** based on the decision points exercised e.g., an if statement evaluated to both true and false with separate test cases during testing is considered 'covered'.
- **Condition coverage:** based on the boolean sub-expressions, each evaluated to both true and false with different test cases. Condition coverage is not the same as the decision coverage.
- **Path coverage measures coverage** in terms of possible paths through a given part of the code executed. 100
- **Entry/exit coverage** measures coverage in terms of possible calls to and exits from the operations in the SUT.

Highest intensity of testing = 100% path coverage

Test Case Design

Except for trivial SUTs, exhaustive testing is not practical because such testing often requires a massive/infinite number of test cases.

Every test case adds to the cost of testing. Therefore, test cases need to be designed to make the best use of testing resources. In particular:

- **Testing should be effective** i.e., it finds a high percentage of existing bugs e.g., a set of test cases that finds 60 defects is more effective than a set that finds only 30 defects in the same system.
- **Testing should be efficient** i.e., it has a high rate of success (bugs found/test cases) a set of 20 test cases that finds 8 defects is more efficient than another set of 40 test cases that finds the same 8 defects

Positive vs Negative Test Cases

A positive test case is when the test is designed to produce an expected/valid behavior. On the other hand, a negative test case is designed to produce a behavior that indicates an invalid/unexpected situation, such as an error message.

Black Box vs Glass Box

Test case design can be of three types, based on how much of the SUT's internal details are considered when designing test cases:

- **Black-box (aka specification-based or responsibility-based) approach:** test cases are designed exclusively based on the SUT's specified external behavior.
- **White-box (aka glass-box or structured or implementation-based) approach:** test cases are designed based on what is known about the SUT's implementation, i.e. the code.
- **Gray-box approach:** test case design uses some important information about the implementation. For example, if the implementation of a sort operation uses different algorithms to sort lists shorter than 1000 items and lists longer than 1000 items, more meaningful test cases can then be added to verify the correctness of both algorithms.

Equivalence Partitions

Equivalence partition (aka equivalence class) is a group of test inputs that are likely to be processed by the SUT in the same way.

By dividing possible inputs into equivalence partitions you can,

- **avoid testing too many inputs from one partition.** Testing too many inputs from the same partition is unlikely to find new bugs. This increases the efficiency of testing by reducing redundant test cases.
- **ensure all partitions are tested.** Missing partitions can result in bugs going unnoticed. This increases the effectiveness of testing by increasing the chance of finding bugs.

Boundary Value Analysis

Boundary Value Analysis (BVA) is a test case design heuristic that is based on the observation that bugs often result from incorrect handling of boundaries of equivalence partitions.

BVA suggests that when picking test inputs from an equivalence partition, values near boundaries (i.e. boundary values) are more likely to find bugs.

Boundary values are sometimes called corner cases.

Typically, you should choose three values around the boundary to test: one value from the boundary, one value just below the boundary, and one value just above the boundary.

Combining Test Inputs

Testing all possible combinations is effective but not efficient.

Given below are some basic strategies for generating a set of test cases by combining multiple test inputs.

- The **all combinations** strategy generates test cases for each unique combination of test inputs.
- The **at least once** strategy includes each test input at least once.
- The **all pairs** strategy creates test cases so that for any given pair of inputs, all combinations between them are tested.
- The **random strategy** generates test cases using one of the other strategies and then picks a subset randomly (presumably because the original set of test cases is too big).

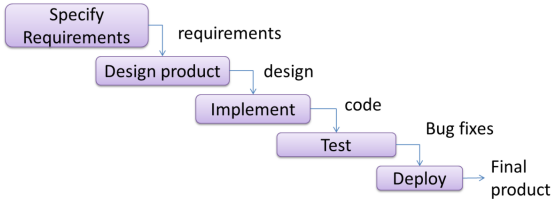
Refer to *SE Textbook Combining Test Inputs Chapter* for more details.

SDLC Process Models

Software development goes through different stages such as requirements, analysis, design, implementation and testing. These stages are collectively known as the software development life cycle (SDLC).

Sequential Models

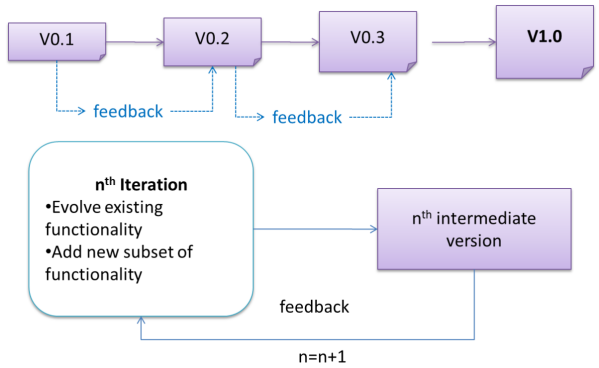
The sequential model, also called the waterfall model, models software development as a linear process, in which the project is seen as progressing steadily in one direction through the development stages.



- When one stage of the process is completed, it should produce some artifacts to be used in the next stage.
- This could be a useful model when the problem statement is well-understood and stable.
- The major problem with this model is that the requirements of a real-world project are rarely well-understood at the beginning and keep changing over time.

Iterative Models

The iterative model (sometimes called iterative and incremental) advocates having several iterations of SDLC.



In this model, each of the iterations produces a new version of the product. Feedback on the new version can then be fed to the next iteration.

The iterative model can take a breadth-first or a depth-first approach to iteration planning.

- **breadth-first:** an iteration evolves all major components in parallel e.g., add a new feature fully, or enhance an existing feature.
- **depth-first:** an iteration focuses on fleshing out only some components e.g., update the backend to support a new feature that will be added in a future iteration.

Agile Models

Extract from the *Agile Manifesto*:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

There are a number of agile processes in the development world today. eXtreme Programming (XP) and Scrum are two of the well-known ones.

XP

- Extreme Programming (XP) stresses customer satisfaction.
- XP aims to empower developers to confidently respond to changing customer requirements, even late in the life cycle.
- XP aims to improve a software project in five essential ways: communication, simplicity, feedback, respect, and courage.
- XP has a set of simple rules.

Pair programming, CRC cards, project velocity, and standup meetings are some interesting topics related to XP. Refer to *extremeprogramming.org* for more details.

Scrum

- Scrum is a process skeleton that contains sets of practices and predefined roles. The main roles in Scrum are:
  - **The Scrum Master**, who maintains the processes (typically in lieu of a project manager)
  - **The Product Owner**, who represents the stakeholders and the business
  - **The Team**, a cross-functional group who do the actual analysis, design, implementation, testing, etc.
- **A Scrum project is divided into iterations called Sprints.**
- **Each sprint is preceded by a planning meeting**, where the tasks for the sprint are identified and an estimated commitment for the sprint goal is made, and followed by a review or retrospective meeting, where the progress is reviewed and lessons for the next sprint are identified.
- **During each sprint, the team creates a potentially deliverable product increment** (for example, working and tested software).
- **Scrum enables the creation of self-organizing teams by encouraging co-location of all team members**, and verbal communication between all team members and disciplines in the project.
- **A key principle of Scrum is its recognition that during a project the customers can change their minds about what they want and need** (often called requirements churn), and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner.
- **In Scrum, on each day of a sprint, the team holds a daily scrum meeting called the "daily scrum".** During the daily scrum, each team member answers the following three questions:
  - What did you do yesterday?
  - What will you do today?
  - Are there any impediments in your way?
- **The daily scrum meeting is not used as a problem-solving or issue resolution meeting.** Issues that are raised are taken offline and usually dealt with by the relevant subgroup immediately after the meeting.

Principles

Single Responsibility Principle

Single responsibility principle (SRP): A class should have one, and only one, reason to change.

Liskov Substitution Principle

Liskov substitution principle (LSP): Derived classes must be substitutable for their base classes.

LSP sounds the same as substitutability but it goes beyond substitutability; **LSP implies that a subclass should not be more restrictive than the behavior specified by the superclass**.

Separation of Concerns Principle

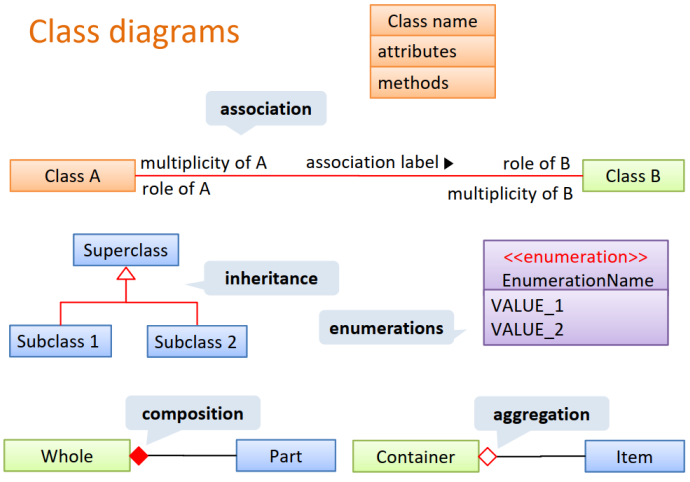
Separation of concerns principle (SoC): To achieve better modularity, separate the code into distinct sections, such that each section addresses a separate concern.

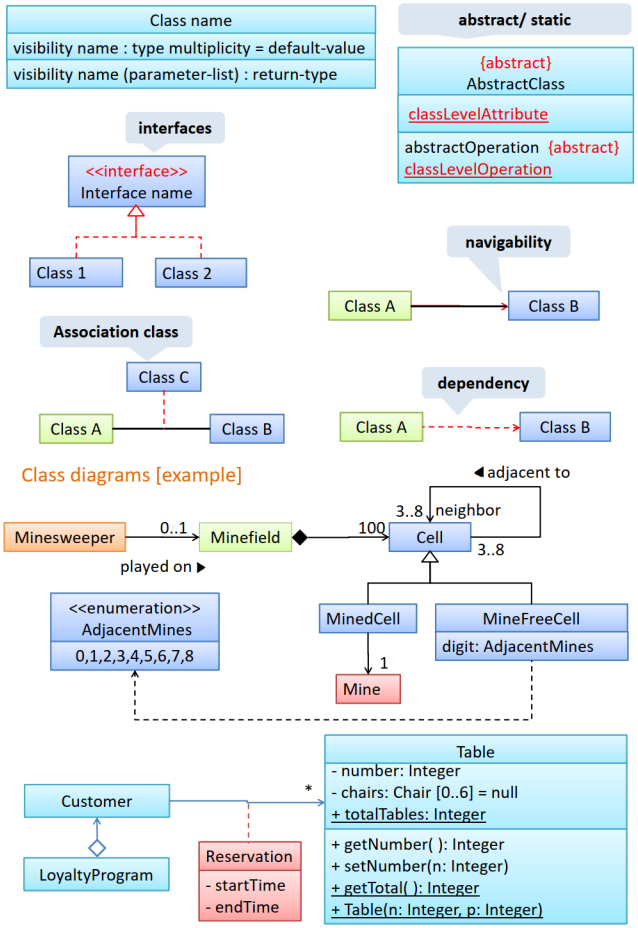
- Applying SoC reduces functional overlaps among code sections and also limits the ripple effect when changes are introduced to a specific part of the system.
- This principle can be applied at the class level, as well as at higher levels.
- This principle should lead to higher cohesion and lower coupling.

UML Diagrams

Class Diagrams

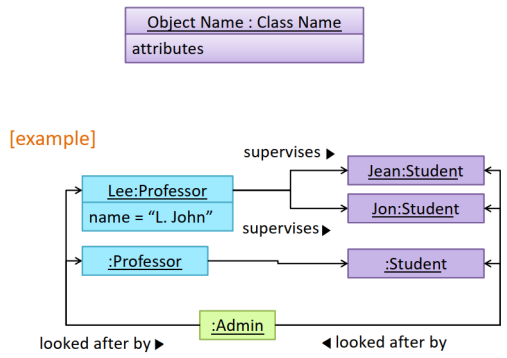
UML class diagrams describe the structure (but not the behavior) of an OOP solution.





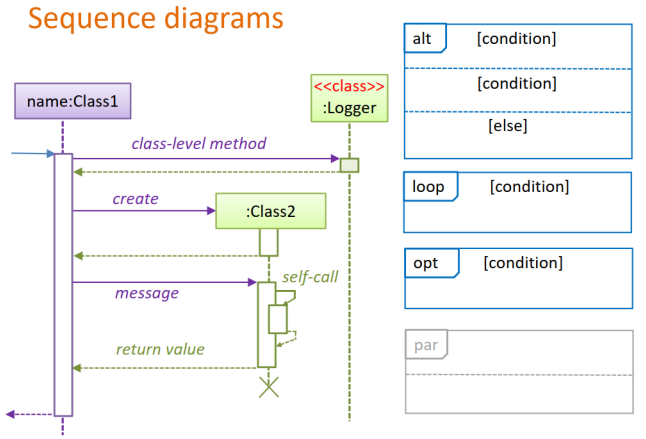
Object Diagrams

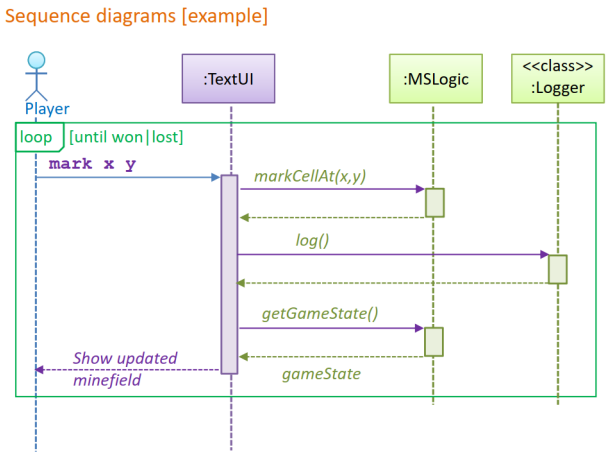
An object diagram shows an object structure at a given point of time.



Sequence Diagrams

A UML sequence diagram captures the interactions between multiple objects for a given scenario.





Notes

UML notes can augment UML diagrams with additional information.