Lazaron Shyta

Introduction

Software engineering is the discipline that encapsulates the development of high-quality software for computer-based systems. The collection of techniques, methods and tools that help in the production of high-quality software systems.

To remain useful, software needs to evolve with users needs. By engineered software, we mean a software that is analyzed, designed, tested and built.

<u>Some common problems that we have with software development are:</u>

1. Requirements are wrong. (Incomplete, inconsistent, ambiguous).
2. Continual change.
3. Developer and customer did not interpret the requirements the same way.

Almost half of the problems are caused in requirements analysis and design.

Successful software companies must have a development process that maximizes quality while reducing development time and cost.

Most Successful software development method -> *Agile Development*

Traditional Development did not solve the software crisis, because they were plan driven, which did not work properly in accommodating change.

Why a software crisis?

- Software delivered late
- Software cost expensive
- Software unreliable-breaks down
- Software difficult to maintain
- Software performs poor design

Agile development is a software development that is iterative, incremental and always involves the customer/client. (gives the best results)

Incremental: Instead of developing the whole required functionality of the software, Agile developers build increments of the required functionality in shorts periods of time.

Iterative: Agile developers build the increments of product's full functionality in iterations, over a sequence of shorts periods of time.

Communication and interaction between the Developers and the customer.

<u>Processes of Agile Development</u>

- Scrum Framework
- Extreme Programming
- Adaptive Systems Development Method
- Future-Driven Development

AGILE

- Communicate frequently with the customer
- Build software in small, but working increments, with each increment adding functionality
- Build Software increments iteratively, each iteration having short duration
- Release each new software increment to the customer for recap

Scrum Estimation

Before we start any software development, we must develop a plan that allow us to estimate the project duration.

As a start to planning the software development project, we need to estimate the development times of each user story and prioritize each user story.

At some point the customer will ask: **How long will it take to develop?** Developers need to create a project time estimate which will be the sum for each individual user story.

Assigning **estimates to the user stories** is the first step towards being able to identify a realistic deadline for the project.

Project Duration: User stories estimates + Team velocity (rate at which team develops)

The **entire team** should participate in estimating the time to develop each user story. A simple estimation activity is known as the planning poker is typically used to assign user stories estimates.

Estimation

Use **points**, not hours of days or weeks.
Secret of agile development is **not** to deal with time (hours/days etc.)
We assign points to user estimates to indicate relative sizes between different stories.
Story points are relative measures of size/not duration. (it's difficult to measure duration)

It's simple

We start by picking the easiest user story (starter user story), give it a point value, and then we size all our user stories relative to it.

Object-Oriented Design Principles

There are some principles that every expert recommend should be used by every developer in order to develop high quality software.

<u>Software design “quality” is an elusive concept</u>

- Quality depends on some specific design priorities
- Quality means different things to the client and developer

Quality: Robustness, Reliability, Reusability, Understandability and Maintainability of design.

Interface → The methods of a class which are public and accessible by client objects. Or, interface is a subset of all the class methods.

Interface → Is used to define an abstract data type that contains no implementation of methods, but exposes operations defined as “empty” methods.

Class Responsibilities

1. Refer to the actions that the class is responsible for (tasks that class implements).
2. The responsibility of a class should be minimal. (Single Responsibility).
3. Each class implements a minimum level of functionality.
4. The methods of a class should be functionally related, *e.g. Bank Account has the responsibility to offer methods to withdraw money from bank account.*

Separation of concerns: divide application into distinct software functionalities, with as little overlap as possible between functionalities (concerns).

Single Responsibility Principle → The class should be responsible for doing only one task and doing it well.

Cohesion and Coupling

Are considered the 2 major factors that influence high-quality software.

Design Principle → Implement your classes so they have **high cohesion** and **low coupling**.

Cohesion → A qualitative measure of how well a class’s methods work together to perform a given task. Class methods are functionally related.

Coupling → A qualitative measure of how much classes depend on each-other for successful execution.

Cohesion

Ideal situation: Every class should be responsible for performing only one task only and doing it well.

- ➔ An important goal in design is to ensure that each class that we design exhibits the **highest level of cohesion.**
- ➔ A good test for cohesion is: "Can I describe the purpose of the class in a short statement without using words like 'and'"?
- ➔ It's important to assign responsibility to classes in a way that keep cohesion high. Ideally, that means that each class **should perform a single task,** and the methods of that class should be strongly connected in what they do.
- ➔ Classes with cohesion are preferable because **high cohesion is associated with several desirable qualities** in software: Robustness, Reliability, Reusability and Understandable
- ➔ Basically, high cohesion means keeping similar and related things together.

Coupling

How classes use or depend on other classes.

A class depends in another class if:

- Classes have an association with each other.
- The second class generalizes(implement) other class.

Ways for a class to depend on another:

- It may use services provided by the other class. (may depend on interface) **good**
- It may use the internals of the other class (access variables directly etc.) **bad**

We want to minimize coupling because:

- If a class B is strongly coupled to a class A, and a class A is modified, class B depends may need to change as well. **bad**
- If B depends only on A's interface, then only a change to A's interface can potentially affect B. **good** (dependency inversion principle)
- However, if B depends on A's implementation, any change to A has a potential effect on B. **bad (dependency inversion principle)**

Bad Designs
What makes a design bad?

3 main contributors

- **Rigidity** – Code is hard to change because every change affects too any other parts of the software.
- **Fragility** – When you make a change, unexpected parts of the software systems "break".
- **Immobility** – Code is hard to reuse in another application, because it cannot be disentangled from the current application.

The design principles discussed below are all aimed at preventing ***bad*** *designs.*
Single Responsibility, Open-Closed, Liskov-Substitution, Interface Segregation, and Dependency Inversion)

SOLID

Single Responsibility Principle: A class should have one, and only one, reason to change.

We should use it because it makes our software easier to implement and prevents side-effects of future changes.

The more responsibilities a class has, the more often we need to change it. If the class implements multiple responsibilities, they are no longer independent of each other.

Single responsibility principle is one of the most commonly used design principles in object-oriented programming. To follow this principle, your class isn't allowed to have more than one responsibility. This avoids any unnecessary, technical coupling between responsibilities and reduces the probability that you need to change the class.

Violating the Single Responsibility Principle **reduces cohesion.**

Open Closed Principle: Considered the most important principle of object-oriented design.

Definition: Software entities should be open for extension but closed for modification. The general idea of this principle is that we must write our code so that we can add new functionality later, but without changing the existing code.

Conformance to this principle yields the greatest level of **reusability** and **maintainability**.

Classes that conform to the OCP meet two criteria:

Open for extension **Closed for modification**

To extend the behavior of a system, we may add new code, but we must not modify existing code.

Inheritance is a simple example of satisfying OCP, because instead of modifying the class that does not quite fit our needs, we **can declare a new class that inherits the methods of the original class and add its own new methods**. But only if inheritance is applicable.

OCP-Breaking Example

Consider a game in which a player is awarded a certain number of points for collecting a variety of fruit.
OCP is **broken** in the following Player class implementation…

```
public class Player {
    ...
    public void scoreFruit (String fruitType) {
        if (fruitType.equals("apple")) {
            this.score += 5;
        } else if (fruitType.equals("cantaloupe")) {
            this.score += 20;
        } else if (fruitType.equals("pomegranate")) {
            this.score += 50;
}}}
```

In order to introduce a new type of fruit, the programmer would have to change the scoreFruit method. This means that **new functionality must be implemented** primary by changing existing code rather than adding new code.

To fix this situation, we can change the method scoreFruit to accept as a parameter an object of class Fruit and adds the score by simply calling a method from the fruit class…

```
public class Player {
    ...
    public void scoreFruit(Fruit fruit) {
        this.score += fruit.getPointValue(); // method from fruit class
}}
```

Liskov Substitution Principle

An instance of a subclass should be able to replace an instance of its superclass **without** any untoward effect.

Use inheritance only such that a subclass instance may be used in any situation that the superclass instance may be used.

If a **subclass cannot replace its superclass**, then it violates the LSP.

Liskov Breaking Example

For example, consider a Bird superclass that is extended by subclasses for different types of bird. The bird superclass implements a fly() method, which is inherited by all subclasses. This is all well and good until we want to implement an Ostrich() subclass. They do not fly. The fly() method of the Ostrich() subclass ***needs to be overridden*** to account for the inability of ostriches to fly. This means that an instance of the Ostrich () subclass cannot replace an instance of the Bird superclass. ***This way the Liskov Principle is broken.***

Interface Segregation Principle

Clients of a class should not be forced to depend on interface methods that they do not use.

When we bundle methods for different client objects into ne interface, we create unnecessary coupling among the clients.

When one client causes the interface to change, all other clients are forced to recompile.

Example

Assume we have an interface IDoor which is used to create simple doors. This interface exposes methods Lock, Unlock, and a property to check if door is open- IsDoorOpen.

```
public interface IDoor
    {
            bool IsDoorOpen {get; set;}               OK!
            void Lock();
            void Unlock();
    }


public class SimpleDoor : IDoor
    {
            public bool IsDoorOpen
            {…}
            public void Lock()                  OK!
            {…}
            public void Unlock ()
            {…}
    }
```

Now, if we want a new requirement to build advanced doors, which need to alert the user to take some action after a specific time interval, we can add a new method TimeInterval(), to the existing IDoor interface, but then this **will impact all existing** SimpleDoor clients which have inherited from IDoor already. Further, SimpleDoor needs to implement a TimeInterval() method even though they do not want to use it, which clearly breaks ISP.

Solution: Define a new interface ITimerInterval and put the TimerInterval() method in it. Also, a new concrete class AdvancedDoor(), which will implement the ITimerInterval.

This design will ensure that changes made for either interface will not have any impact on another.

Dependency Inversion Principle

Depend on abstractions, not on implementations (concrete classes).

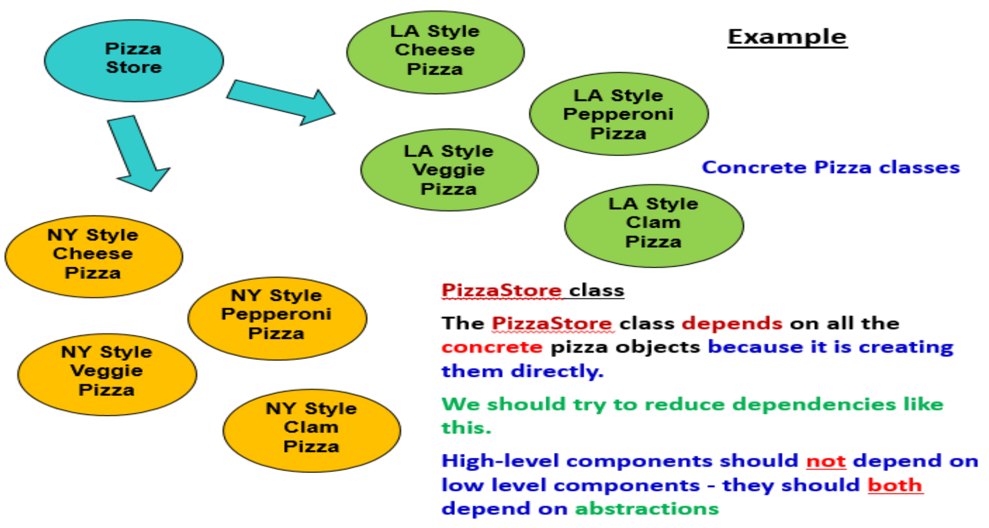
Specifically, high-level components should not depend upon low-level components.

A high-level component is a class with behavior defined in terms of low-level components.

Rather, **both** types of components (high and low) should depend upon abstractions, i.e. interfaces.

Example

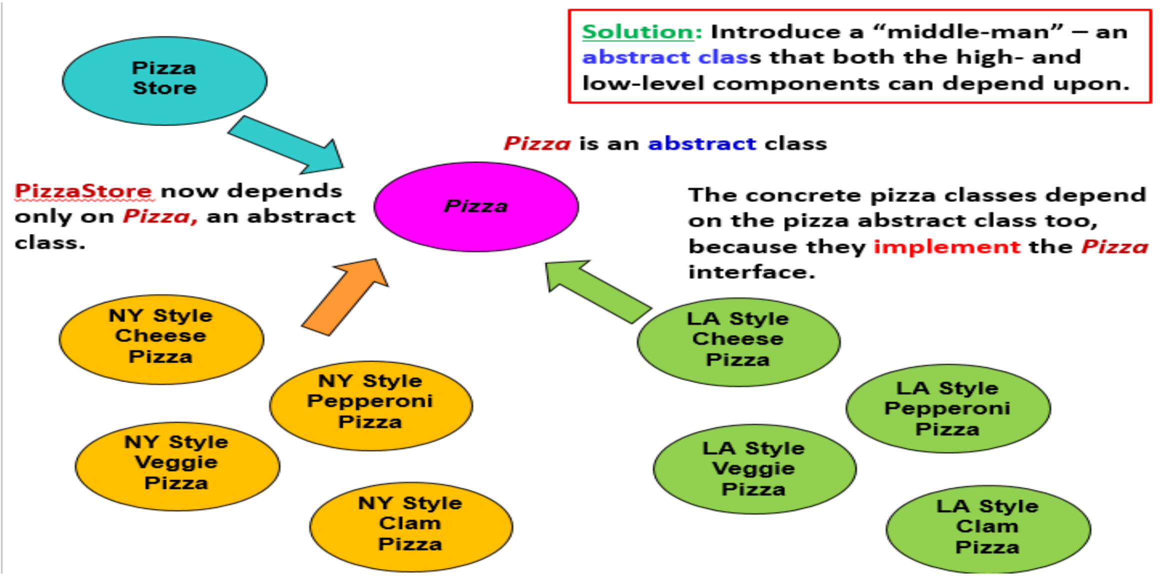
Consider a PizzaStore class. This class is a high-level component, but its behavior is defined in terms of low-level pizza classes – it creates all the different types of pizza directly, which are concrete classes.


Pizza Store
LA Style Cheese Pizza
LA Style Pepperoni Pizza
LA Style Veggie Pizza
LA Style Clam Pizza
NY Style Cheese Pizza
NY Style Pepperoni Pizza
NY Style Veggie Pizza
NY Style Clam Pizza
Example
Concrete Pizza classes
PizzaStore class
The PizzaStore class depends on all the concrete pizza objects because it is creating them directly.
We should try to reduce dependencies like this.
High-level components should not depend on low level components - they should both depend on abstractions

Because any changes to the concrete implementations of the Pizzas affect the PizzaStore class, we say that the PizzaStore depends on the Pizza implementations.

If the implementation of the Pizza classes changes, then we may have to modify PizzaStore.

Every new kind of Pizza we add creates another dependency for PizzaStore.


Solution: Introduce a "middle-man" – an abstract class that both the high- and low-level components can depend upon.
Pizza Store
Pizza is an abstract class
PizzaStore now depends only on Pizza, an abstract class.
Pizza
The concrete pizza classes depend on the pizza abstract class too, because they implement the Pizza interface.
NY Style Cheese Pizza
NY Style Pepperoni Pizza
NY Style Veggie Pizza
NY Style Clam Pizza
LA Style Cheese Pizza
LA Style Pepperoni Pizza
LA Style Veggie Pizza
LA Style Clam Pizza