UNIVERSIDAD FUERZAS ARMADAS

Name: Andy Josue Calderón Merchan

Date: 1/2/2022

READING SUMMARY

INTRODUCTION TO THE SOLID PRINCIPLES

OVERVIEW OF THE SOLID PRINCIPLES

SOLID stands for five principles and is a mnemonic acronym for them.

• Single Responsibility Principle

• Open/Closed Principle

• Liskov Substitution Principle

• Interface Segregation Principle

• Dependency Inversion Principle

Following these guidelines can help to ensure high-quality software, especially from the perspective of developers.

DESIGN PROBLEM

You've been hired to construct a Maze Generator.

The software must create mazes with N x M square rooms, where N and M are the maze's width and height, respectively.

The mazes must be able to be printed as ASCII characters or drawn in an image by the application.

SINGLE RESPONSIBILITY PRINCIPLE

Each class should be in charge of a specific aspect of the system's operation.

The responsibility of a class should be totally encapsulated within the class.

The idea of cohesion states that any component's responsibilities (methods, classes, sub-systems, etc.) should be firmly coordinated and focused on a single goal.

This concept is also linked to the concepts of:

- Encapsulation
- Localization of design
- Increase Reuse and Maintainability

OPEN/CLOSED PRINCIPLE

Software entities (for example, classes and generics) should be extensible but not modifiable.

- If you can still extend a class, it's considered open.
- If a class is open for usage by other classes, it is closed and should not be updated.

Interfaces or abstract classes in Java are used to declare public methods such as the abstractions.

The public method declarations from interfaces, abstract classes, and pure virtual classes are inherited by implementations or concrete classes.

INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES

The ability to especialize a derivado class in order to repurpose the generalization a base class is provided by herencia. An interface is similar to a base class, except that it only enables method definitions. It prohibits the declaration or definition of data members.

Abstract and pure virtual classes and methods are written in italics in UML.

Interfaces are supported by Java.

Data members and some method implementations may be included in an abstract and a pure virtual class (C++), at least one method declaration without an implementation is common.

OPEN/CLOSED PRINCIPLE

The current Open/Closed Principle encourages developers to specify public data elements using interfaces, abstract classes, and pure virtual classes.

- Inheritance

Public methods should be placed in their own abstractions, such as interfaces, abstract classes, or pure virtual classes.

A single class's public methods can be divided into several abstractions, every abstraction should have a single goal in mind.

- Aggregation

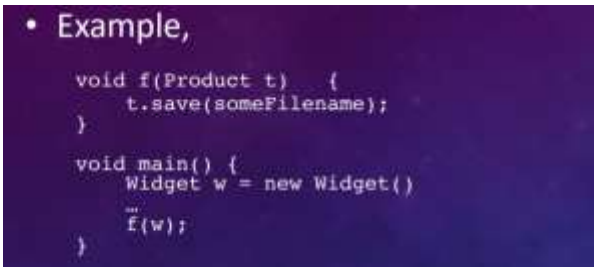
Sub-part objects can be used to encapsulate activities and allow them to alter dynamically.

- Parameterization
  Assist developers in decreasing complexity by eliminating coupling.

LISKOV SUBSTITUTION PRINCIPLE

Allow Product to be a base class with only one virtual method, save, that saves an object to a file.

- Widget.save() does not make the same assumptions as Product.save()
- Any special circumstances required by Widget.save() should be implemented programmatically, and errors should be handled accordingly.
  As a minimum, make sure Widget.save() does everything Product.save() is meant to do.



- Example,

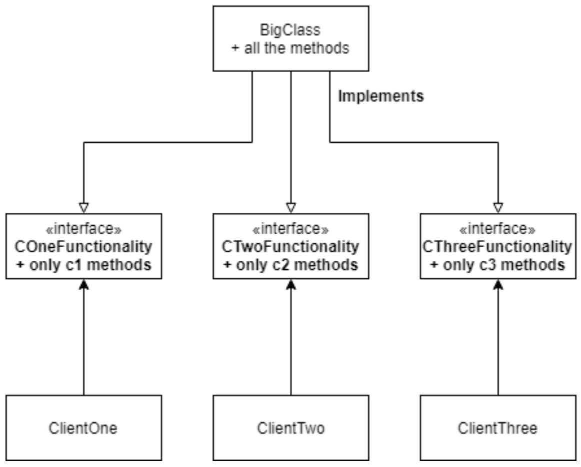
```
void f(Product t)   {
    t.save(someFilename);
}

void main() {
    Widget w = new Widget()
    …
    f(w);
}
```

INTERFACE SEGREGATION PRINCIPLE

An interface is a "window" or "portal" into a component's functionality, represents public methods of a component.

Interfaces are directly supported by Java.
No client (component user) should be made to rely on methods it doesn't use. Purp can be used to group a component's public methods.

Example:



DEPENDENCY INVERSION PRINCIPLE

Layer the system: some levels, such as reusable libraries or frameworks, will be more abstract or policy-setting, components from the abstract layers should not rely on components from the concrete layers.

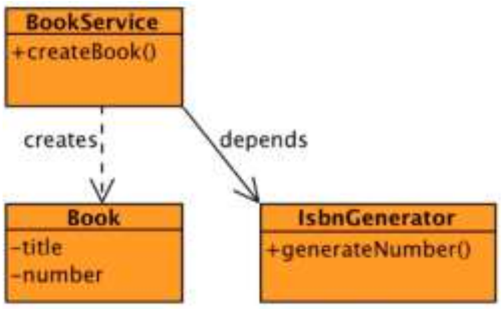
Details should not be relied upon in abstracts.

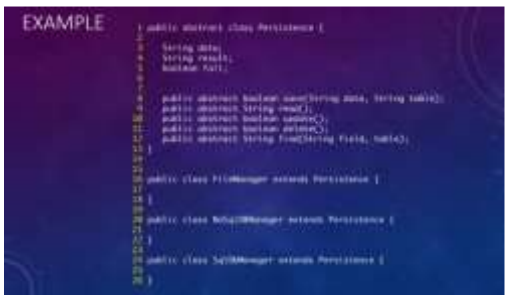
Details of implementation should be based on abstractions.

- Low-level modules should not be relied upon by high-level modules and should depend on abstractions.

The Dependency Inversion Principle is beneficial to developers.

- Increase Reusability
- Increase Maintainability





EXAMPLE

```
public abstract class Persistence {

    String data;
    String result;
    boolean full;

    public abstract boolean save(String data, String table);
    public abstract String read();
    public abstract boolean update();
    public abstract boolean delete();
    public abstract String find(String field, table);
}


public class FileManager extends Persistence {

}

public class NoSqlDBManager extends Persistence {

}

public class SqlDBManager extends Persistence {

}
```