Lecture 2 : In depth analysis of SOLID principles and their applications in codes

CSE 325 : Information System Design

A.H.M.Osama Haque : 1805002
Syed Jarullah Hisham: 1805004
Tanjeem Azwad Zaman : 1805006
Abdur Rafi : 1805008
Rownok Ratul : 1805019
Md Toki Tahmid : 1805030

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

January 18, 2024

Contents

Chapter 1

1805030 - SOLID Principles

Student Name: Md Toki Tahmid Student ID: 1805030

1.1 Key Points to Consider in Developing OOP Based Application

In general, while developing any OOP (Object Oriented Programming) based application, it is recommended to ensure four key ideas.

- **Manageability**

  Manageability refers to the ease with which a software product can be managed, maintained, and updated by its developers and administrators. A manageable software product is one that can be easily monitored, analyzed, and maintained throughout its lifecycle.

- **Reusability**

  Reusability refers to the ability of a software product to be used across multiple applications or projects without requiring significant modifications. A reusable software product is one that can be easily integrated with other software systems, and can be used to solve a variety of problems.

- **Readability**

  Readability refers to the ease with which a software product's code can be understood and interpreted by its developers. A readable software product is one that is well-structured, easy to navigate, and clearly documented.

- **Maintainability**

  Maintainability refers to the ability of a software product to be updated, modified, and enhanced over time. A maintainable software product is one that can be easily modified and extended to meet changing user needs and requirements.

The concept of object-oriented programming in built upon four principles:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

1.2 Motivation Behind SOLID

The above-mentioned principles are developed to help programmers write more maintainable, extensible, and reusable code. However, these principles do not provide any definite *Guideline* about how to achieve the goals that are recommended to be ensured in any software product as mentioned at the beginning. For example, it is often recommended by professionals that, each class should contain only one major function. It is also recommended not to write more than 2 layers of looping or 3 layers of nested conditionals. These ideas can not be solely implemented just by following the OOP principles.
To help developers with necessary and defined guidelines to attain manageability, reusability, read-

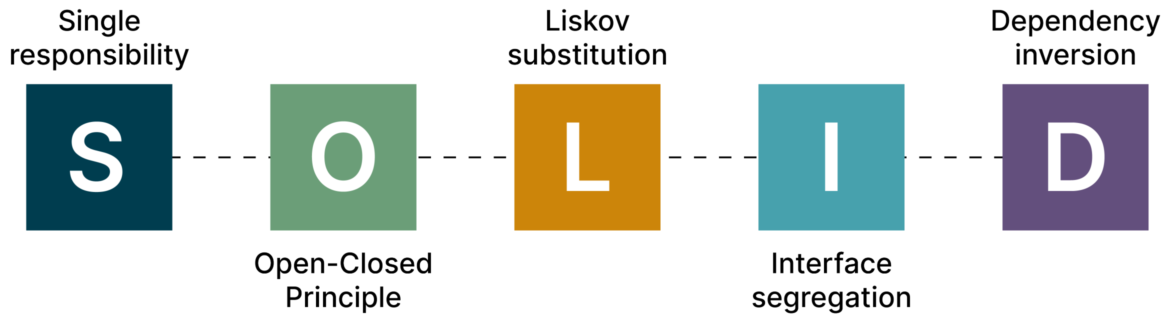


Figure 1.1: SOLID principles

ability, and maintainability, Robert C. Martin (also known as Uncle Bob) introduced five concepts that are generally known as **SOLID** principles, see Fig 1.1.

SOLID is not a library, not a framework, nor an end goal that must be achieved. Rather it is the collection of some ideas, that help the software to be flexible, supple, and adaptable.

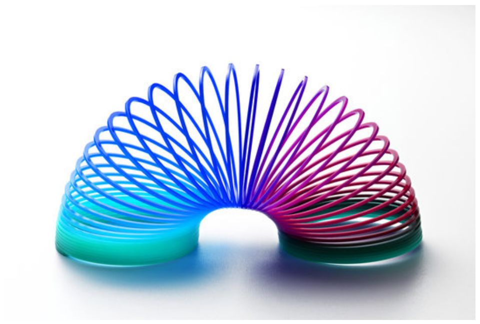


Figure 1.2: Major motivation behind SOLID principles is to make the software reshapeable, flexible, and adaptive.

1.3 Basic Understanding of the SOLID Principles

Here a brief description of SOLID principles along with some real-life examples from software engineering perspective are provided.

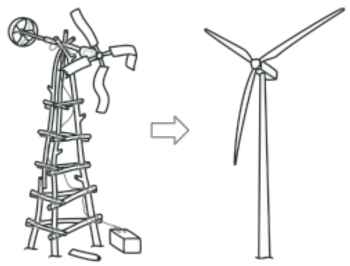
Table 1.1: SOLID Principles and Real-Life Examples

Principle Name	**Basic Idea**	**Real-Life Example**
Single Responsibility Principle (SRP)	A class should have only one reason to change.	A class that handles user authentication should only handle user authentication, and not also handle file I/O or network communication.
Open/Closed Principle (OCP)	Software entities should be open for extension but closed for modification.	A report generation module should be open for extension (e.g. to add new report types) but closed for modification (i.e. existing code shouldn't need to change).
Liskov Substitution Principle (LSP)	Subtypes must be substitutable for their base types.	If a program expects an object of type \`Animal\`, it should be able to work correctly with any subtype of \`Animal\`, such as \`Cat\` or \`Dog\`.
Interface Segregation Principle (ISP)	Clients should not be forced to depend on interfaces they do not use.	A client that only needs to read data from a database should not be forced to implement a method for writing data.
Dependency Inversion Principle (DIP)	Depend on abstractions, not concretions.	Instead of directly depending on a specific database implementation, a class should depend on an interface that abstracts away the details of the database.

We have provided just some basic understanding on the SOLID principles and a few real-life examples to get acquainted to the principles of SOLID. In the subsequent chapters, each of these principles will be explored in more details. At this point, we are going to discuss some essential terminologies that will be helping us while going through the codes of the next sections. In short, the key-points include:

- Refactoring
- Null Object Pattern
- Logging
- Caching

1.4 Refactoring



Code refactoring is the process of improving the structure, design, and organization of existing code, without changing its behavior. The goal of refactoring is to make the code easier to understand, maintain, and extend, while reducing its complexity, duplication, and technical debt. Here the key-points are:

- Refactoring does not change the behaviour of the software as seen from the end users
- It helps improving the code structure gradually
- Often refactoring helps in achieving a particular design pattern into the codebase.

1.5 Null Object Pattern

Sometimes in our code, we may have to deal with situations where an object that is expected to exist, doesn't exist, and is represented by a null reference. This can lead to errors, bugs, or unexpected behaviors in our program.

To avoid these problems, we can use the Null Object Pattern. Instead of checking whether the object exists or not, we create a **Null object** that behaves like the missing object, but doesn't actually do anything. This way, we can handle missing objects gracefully, without worrying about null reference errors.

For example, imagine we have a program that uses a database to store customer information. Sometimes, a customer may not have any address information recorded in the database. Instead of checking whether the address object exists or not, we can create a "null address" object that behaves like an address object, but doesn't actually contain any information (Fig 1.3). This way, we can avoid null reference errors when trying to access the address object for a customer that doesn't have an address. In the next page we provide a simple code snippet to understand the situation.

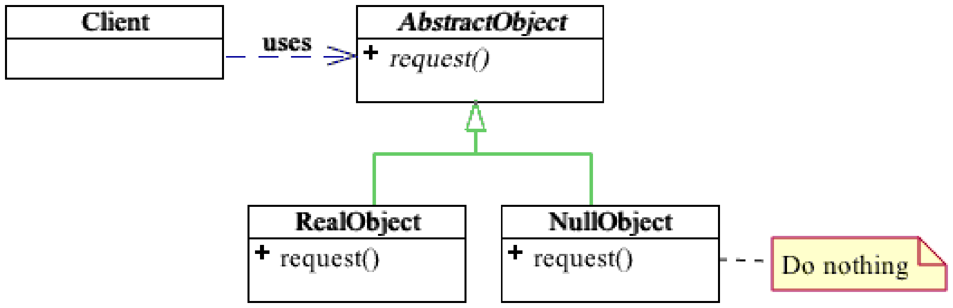


Figure 1.3: Class diagram of Null Object Pattern

```
interface Address {
String getStreet();
String getCity();
String getState();
String getZipCode();
}

class NullAddress implements Address {
@Override
public String getStreet() {
return "";
}
@Override
public String getCity() {
return "";
}
@Override
public String getState() {
return "";
}
@Override
public String getZipCode() {
return "";
}

class Customer {
    private String name;
    private Address address;

    public Customer(String name, Address address) {
        this.name = name;
        this.address = address;
    }
    public void printAddress() {
        System.out.printf("Address: %s, %s, %s %s%n",
        address.getStreet(), address.getCity(), address.getState(),
        address.getZipCode());
    }
}
// Usage example
public class Main {
    public static void main(String[] args) {


        // Create a customer without an address
        Customer customer2 = new Customer("Jane Smith", new NullAddress());
        customer2.printAddress(); // Output: Address: , ,
    }
}
```

1.6 Caching

In software engineering, caching refers to the process of storing frequently accessed data in a temporary storage location, usually in memory or on disk, to reduce the time and resources required to access the data again in the future.

Caching is commonly used to improve the performance of software systems by reducing the number of requests that need to be made to the original data source. For example, if a web application needs to fetch data from a database, it can cache the results in memory so that subsequent requests for the same data can be served from the cache instead of going all the way to the database. In the code example we are going to discuss throughout this note, caching is used to store and retrieve messages from a cached file server. The actual code snippet is shown below:

```
this.cache.AddOrUpdate(id,msg,(i,s)=>msg);
```

This is used to save messages into the cache against a particular message id. To retrieve/read messages from the cache, the following code is used:

```
this.cache.GetOrAdd(id,_ =>File.readAllText(file.FullName));
```

In the following chapters, we will see, how using cache server jointly within the business class's codebase might lead to the violation of SOLID principles, and how we resolve these issues.

1.7 Logging

Logging is an essential aspect of software engineering that enables developers to capture important information about the behavior of their software applications. Logging is the process of recording the events that occur within an application during its runtime. The primary goals of using LOG are:

- Debugging

  When an application encounters an error or bug, developers can use the log files to trace the sequence of events that led up to the problem.

- Performance monitoring

  Log files can be used to monitor the performance of an application over time. Developers can use this information to identify areas where the application is experiencing performance issues and optimize the code to improve performance.

- Auditing

  In some industries, such as finance or healthcare, it is necessary to maintain a record of all actions taken within an application. Log files can be used to provide an audit trail of all actions taken within the application, which can be useful for compliance and regulatory purposes

- Analytics

  Log files can be used to analyze user behavior and usage patterns within an application.

Logging is only one major functionality that if included in the business codebase, can cause the violation of SOLID principles. In the following sections, we will discuss each principle with detailed examples and resolve the issues discussed earlier.

Chapter 2

1805002 - SRP

Student Name: A.H.M. Osama Haque Student ID: 1805002

2.1 Definition

The **Single-Responsibility Principle** dictates that ”A module should be responsible to one, and only one, actor.” The term actor refers to a group that requires a change in the module.The classes should have only a single reason to change. Multiple reasons for change indicate more tightly-coupled designs that are more rigid and harder to maintain.

2.2 Explanation

An in-depth explanation of SRP is given below:

- In case of multi-purpose application, classes need to be changed accordingly. Therefore, it is necessary to maintain single responsibility so that internal structure is not affected on a whole.
- A system can be divided into multiple layers. However, layering does not necessarily ensure SRP.
- We need to make sure that the business classes all are single responsible. The concern of SRP is not related to the upper levels. Rather it is realted to the class levels.
- There is no particular mathematical measurement regarding SRP. It is more like judgemental.

2.3 Simple Example

2.3.1 Multiple Responsibilities

```
public class Student {
     public void registerStudent() {
         // some logic
     }
     public void calculate_Student_Results() {
         // some logic
     }
     public void sendEmail() {
         // some logic
     }
}
```

The class above violates the single responsibility principle. This Student class has three responsibilities.

- Registering students.
- Calculating their results.
- Sending out emails to students.

The code above will work perfectly but will lead to some challenges. The code is not reusable for other classes or objects. The class has a whole lot of logic interconnected that one would have a hard time fixing errors. And as the codebase grows, so does the logic, making it even harder to understand what is going on.

Imagine a new developer joining a team with this sort of logic with a codebase of about two thousand lines of code all congested into one class.

2.3.2 Achieving Single Responsibility

```
public class StudentRegister {
    public void registerStudent() {
        // some logic
    }
}
public class StudentResult {
    public void calculate_Student_Result() {
        // some logic
    }
}
public class StudentEmails {
    public void sendEmail() {
        // some logic
    }
}
```

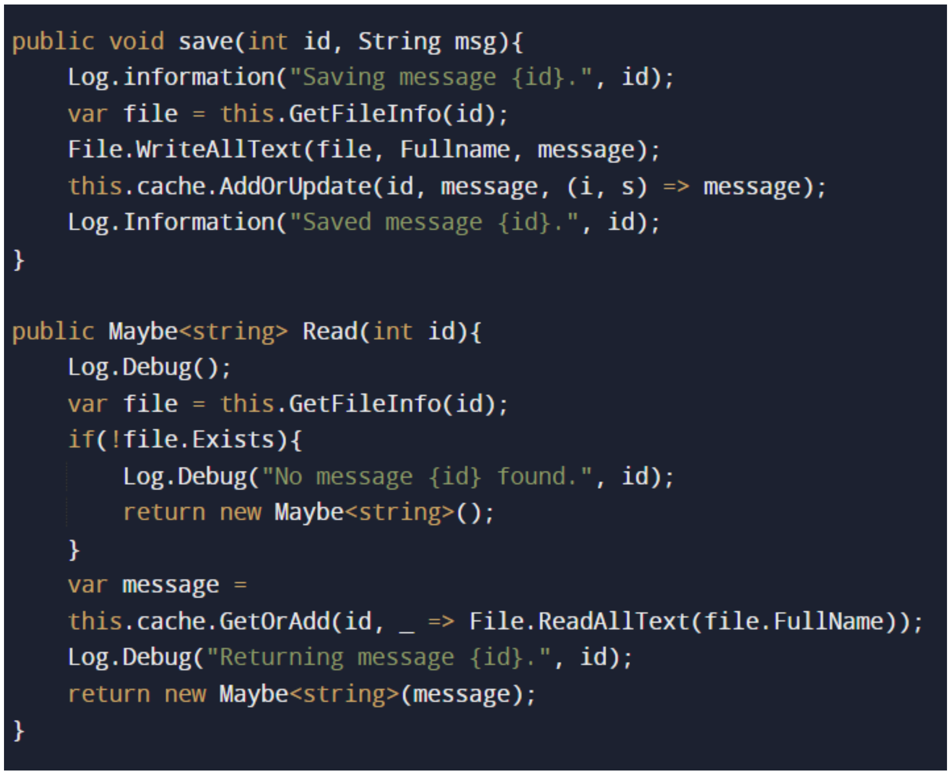
Now that we've separated each functionality in our program, our convenience can be explained as below:

- We can call the classes anywhere we want to use them in our code.
- The code is easier to understand as each core functionality has its own class.
- We can test for errors more efficiently.
- The code is now reusable. Before, we could only use these functionalities inside one class but now they can be used in any class.
- The code is also easily maintainable and scalable because instead of reading interconnected lines of code, we have separated concerns so we can focus on the features we want to work on.

For simplicity, examples used use had each class having one method. One can have as many methods as he wants but they should be linked to the responsibility of the class.

2.4 Detailed Example of SRP implementation

We have a file storing system that performs following functions:
Logging, Caching, Storage, Orchestration



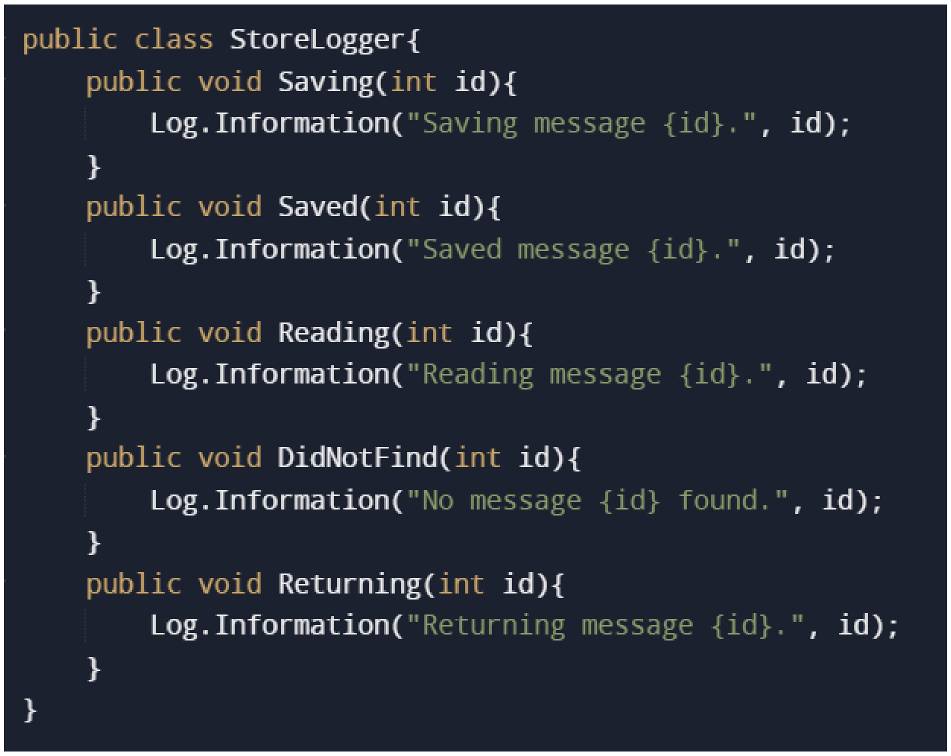
```
public void save(int id, String msg){
    Log.information("Saving message {id}.", id);
    var file = this.GetFileInfo(id);
    File.WriteAllText(file, Fullname, message);
    this.cache.AddOrUpdate(id, message, (i, s) => message);
    Log.Information("Saved message {id}.", id);
}

public Maybe<string> Read(int id){
    Log.Debug();
    var file = this.GetFileInfo(id);
    if(!file.Exists){
        Log.Debug("No message {id} found.", id);
        return new Maybe<string>();
    }
    var message =
    this.cache.GetOrAdd(id, _ => File.ReadAllText(file.FullName));
    Log.Debug("Returning message {id}.", id);
    return new Maybe<string>(message);
}
```

Figure 2.1: Original File Store class

The problem with this implementation is that the functions are not single responsible. For making changes to at least one functionality results into changing the whole class. Therefore it is better that we separate each functionality.

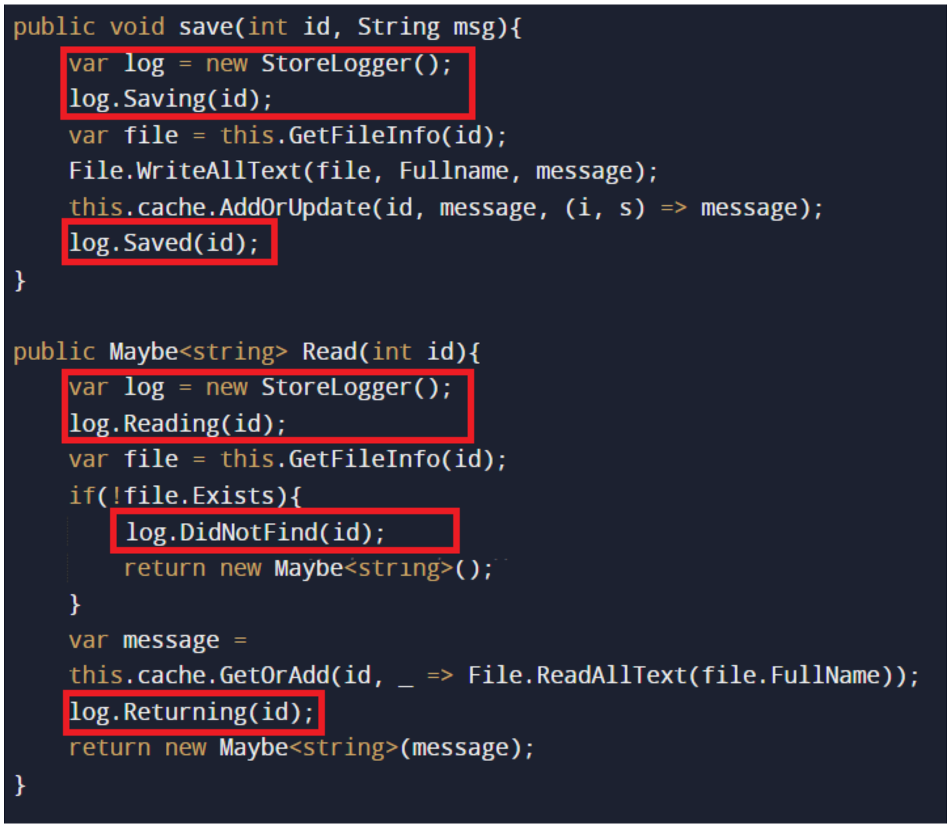
2.4.1 Separating Logging function

Our current implementation has five logging message in total. Suppose, we have the following ***StoreLogger*** class that handles all the five message logs. Now we can just make an instance of this class in the original fileStore class and use the methods separately.



```
public class StoreLogger{
    public void Saving(int id){
        Log.Information("Saving message {id}.", id);
    }
    public void Saved(int id){
        Log.Information("Saved message {id}.", id);
    }
    public void Reading(int id){
        Log.Information("Reading message {id}.", id);
    }
    public void DidNotFind(int id){
        Log.Information("No message {id} found.", id);
    }
    public void Returning(int id){
        Log.Information("Returning message {id}.", id);
    }
}
```

Figure 2.2: StoreLogger class



```
public void save(int id, String msg){
    var log = new StoreLogger();
    log.Saving(id);
    var file = this.GetFileInfo(id);
    File.WriteAllText(file, Fullname, message);
    this.cache.AddOrUpdate(id, message, (i, s) => message);
    log.Saved(id);
}

public Maybe<string> Read(int id){
    var log = new StoreLogger();
    log.Reading(id);
    var file = this.GetFileInfo(id);
    if(!file.Exists){
        log.DidNotFind(id);
        return new Maybe<string>();
    }
    var message =
    this.cache.GetOrAdd(id, _ => File.ReadAllText(file.FullName));
    log.Returning(id);
    return new Maybe<string>(message);
}
```

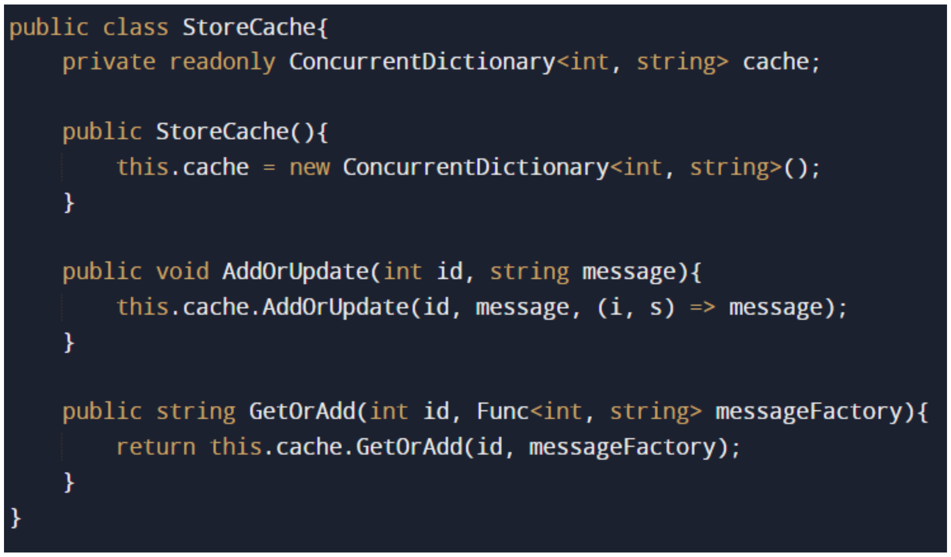
Figure 2.3: Modified FileStore Methods

Now that we have separate functionality for logging, anytime we need to make changes to caching or storing, we do not have to deal with the ***StoreLogger*** class. In case of changing the messages, we only need to change the messages in the ***StoreLogger*** class. The original classes (***Save(), Read()***)

remain untouched. Thus, we have removed dependency on logger function.

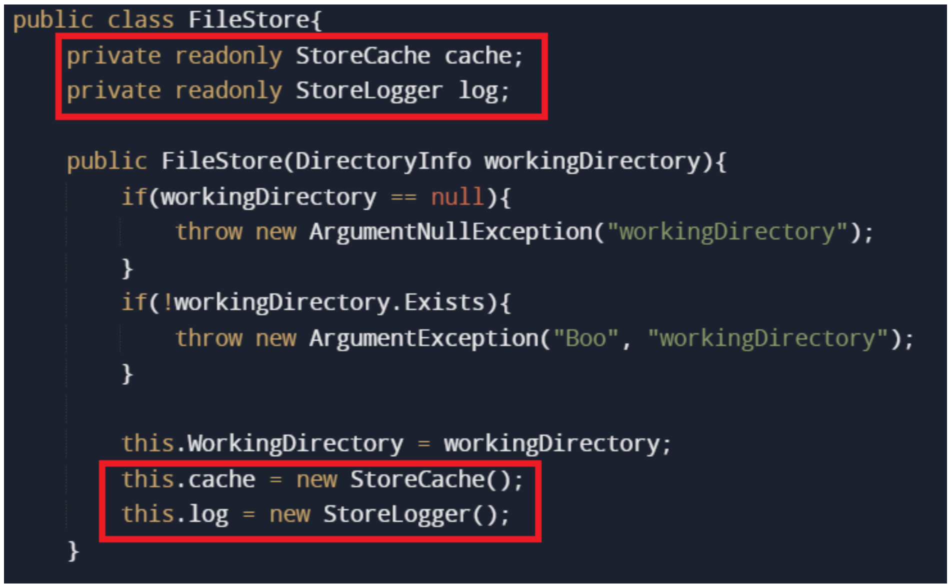
2.4.2 Separating Caching Function

Right now, we have a dictionary ***this.cache***. We want to remove this dependency so that we do not have to directly use this dictionary in our system. Suppose we use the following ***StoreCache*** class that handles all dictionary caching.

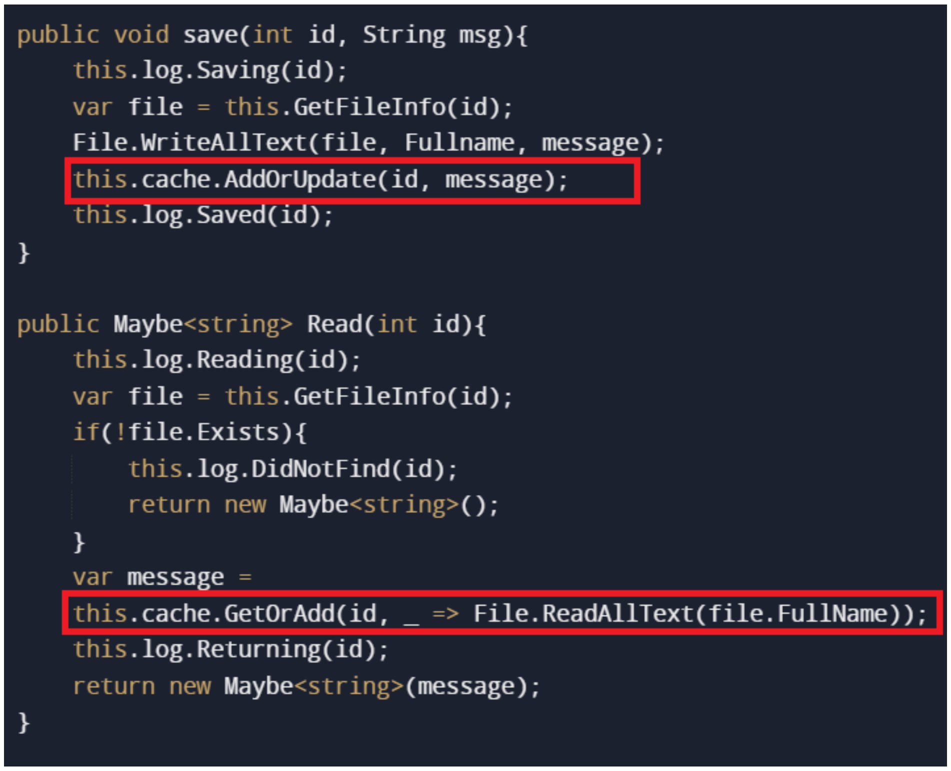


```
public class StoreCache{
    private readonly ConcurrentDictionary<int, string> cache;

    public StoreCache(){
        this.cache = new ConcurrentDictionary<int, string>();
    }

    public void AddOrUpdate(int id, string message){
        this.cache.AddOrUpdate(id, message, (i, s) => message);
    }

    public string GetOrAdd(int id, Func<int, string> messageFactory){
        return this.cache.GetOrAdd(id, messageFactory);
    }
}
```

Figure 2.4: StoreCache class



```
public class FileStore{
    private readonly StoreCache cache;
    private readonly StoreLogger log;

    public FileStore(DirectoryInfo workingDirectory){
        if(workingDirectory == null){
            throw new ArgumentNullException("workingDirectory");
        }
        if(!workingDirectory.Exists){
            throw new ArgumentException("Boo", "workingDirectory");
        }

        this.WorkingDirectory = workingDirectory;
        this.cache = new StoreCache();
        this.log = new StoreLogger();
    }
```



```
public void save(int id, String msg){
    this.log.Saving(id);
    var file = this.GetFileInfo(id);
    File.WriteAllText(file, Fullname, message);
    this.cache.AddOrUpdate(id, message);
    this.log.Saved(id);
}

public Maybe<string> Read(int id){
    this.log.Reading(id);
    var file = this.GetFileInfo(id);
    if(!file.Exists){
        this.log.DidNotFind(id);
        return new Maybe<string>();
    }
    var message =
    this.cache.GetOrAdd(id, _ => File.ReadAllText(file.FullName));
    this.log.Returning(id);
    return new Maybe<string>(message);
}
```

Figure 2.5: Modified FileStore Methods

So, now our business logic does not know how caches are stored. Its implementation is hidden (encapsulated) and made abstract.It is handled by ***StoreCache*** independently. If we need to change the caching mechanism, we just have to modify the ***StoreCache*** class without affecting other parts of the code. It is also worth noting that, earlier in the ***Save()*** and ***Read()*** functions, StoreLogger instances were created twice which was redundant. We could simply create one StoreLogger class instance in the ***FileStore*** constructor.

2.5 Conclusion

According to *Robert C. Martin*, SRP is one of the simplest principles of SOLID yet one of the most elusive to get right. Continually refactoring, checking for multiple responsibilities, and keeping an eye (nose) out for code smell can help ensure SRP is well-realized.

As with any design principle, there is no universal necessity of application. In some cases, adhering to the SRP principle may violate higher-order goals like avoiding unnecessary complexity. After all, if refactoring classes for a simple application results in obfuscated code, superfluous inheritance, and unclear functionality, one may be better served reverting to the practice of Keeping It Simple Stupid (KISS).

Chapter 3

1805004 - OCP

Student Name: Syed Jarullah Hisham Student ID: 1805004

3.1 Definition

This **Open Closed Principle** can be described as - ”Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification”. According to Robert C. Martin, this is the most important principle in object oriented design. The key idea here is, our code should be written in such a way that we will be able to add new functionality without changing the existing code.

3.2 Explanation

We can explain this principle in two parts -

- Open for extension
- Closed for modification

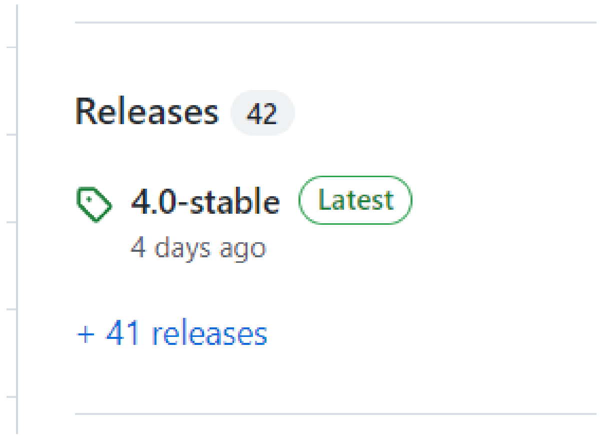
3.2.1 Closed for modification

Firstly, we will elaborate **”Closed for modification”**. Here the closed means that after one version release we should not alter the existing released versions' code. Because, if we modify that code, people who are using this in their code may get the version not working or broken. Also, if it is a library code, all the placed that library is used will get broken. So, there will be a ripple effect in all places that code or version used and thus break all the current running usages.

3.2.2 Open for extension

Secondly, we will elaborate **”Open for extension”**. Here comes the solution of previously explained problem of modification. Instead of modifying he existing version, we may create a new version with the bug fix or other extension or improvement of the code and make a new release. We can make the previous version obsolete or outdated and notify the users about the problem of previous version and why they should use the new version. We can also set a date or time indicating that for how much time the existing version will be supported. For extension of current version, we may use inheritance or composition.

We can describe the whole scenario as follows - A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients.


Releases 42

4.0-stable Latest

4 days ago

+ 41 releases

Figure 3.1: Release Versions in Github

3.3 Favour Composition Over Inheritance for extension

According to previous discussion, we will need any one of composition or inheritance while doing the extension for existing classes. On composition, a class, which desires to use the functionality of an existing class, doesn't inherit, instead it holds a reference of that class as a member variable. Inheritance and composition relationships are referred as IS-A and HAS-A relationships.

Because of IS-A relationship, an instance of sub class can be passed to a method, which accepts instance of super class. This is a kind of Polymorphism, which is achieved using Inheritance.A super class reference variable can refer to an instance of sub class. By using composition, we don't get this behavior, but still it offers a lot more to tilde the balance in its side.

Now, we will actually prefer composition over inheritance even though both will serve our purpose. Here are some reasons for doing this -

- Programming languages like Java and C# don't allow multiple inheritance. So, we can only extend one class at a time. So, if we need different variation in same functionality of a class, we will need to extend existing class into multiple extended classes which may not be an ideal scenario to handle. Rather, using composition, we can use multiple references of same object, each of which will establish different functionality. Thus, composition will give more flexibility in our code.

- Composition offers better test-ability of a class. We can easily create mock object representing composed class for the sake of testing only

- Different design patterns like Strategy pattern, decorator pattern use composition over inheritance

- Sometimes, inheritance breaks encapsulation. If sub class is depending on super class behavior for its operation, it suddenly becomes fragile. When behavior of super class changes, functionality in sub class may get broken, without any change on its part.

So, for these reasons, we will prefer composition while doing extension.

3.4 Composition Example

```
// interface
Interface IClass {
   f1();
   f2();
   f3();
}

// existing implementation
Class1 implements IClass {
   f1() {
      // implement f1
   }
   f2() {
      // implement f2
   }
   f3() {
      // implement f3
   }
}

// Suppose, we need to change f3 implementation. So new class Class2 by composition
Class2 implements IClass {

   // existing class reference
   Class1 obj1;

   // no change in f1
   f1() {
      obj1.f1();
   }
   // no change in f2
   f2() {
      obj1.f2();
   }
   f3() {
      // new implementation
   }
}
```

3.5 OCP in simple example

3.5.1 Problem without OCP

Consider a simple class PaymentManager which has different kinds of payment method.

```
public enum paymentType = {Cash, CreditCard};

public class PaymentManager {
   public PaymentType paymentType {get;set;}

   public void Pay(Money money) {
     if(paymentType == PaymentType.cash) {
            // pay with cash
     } else {
           // pay with credit card
     }
   }
}
```

and if we need to add a new payment type? we need to modify this class. So, definitely it is a violation of OCP principle.

3.5.2 Solution

Open for extension:

```
public class Payment {

   public virtual void Pay(Money money) {
      // base class
   }
}
```

Closed for modification:

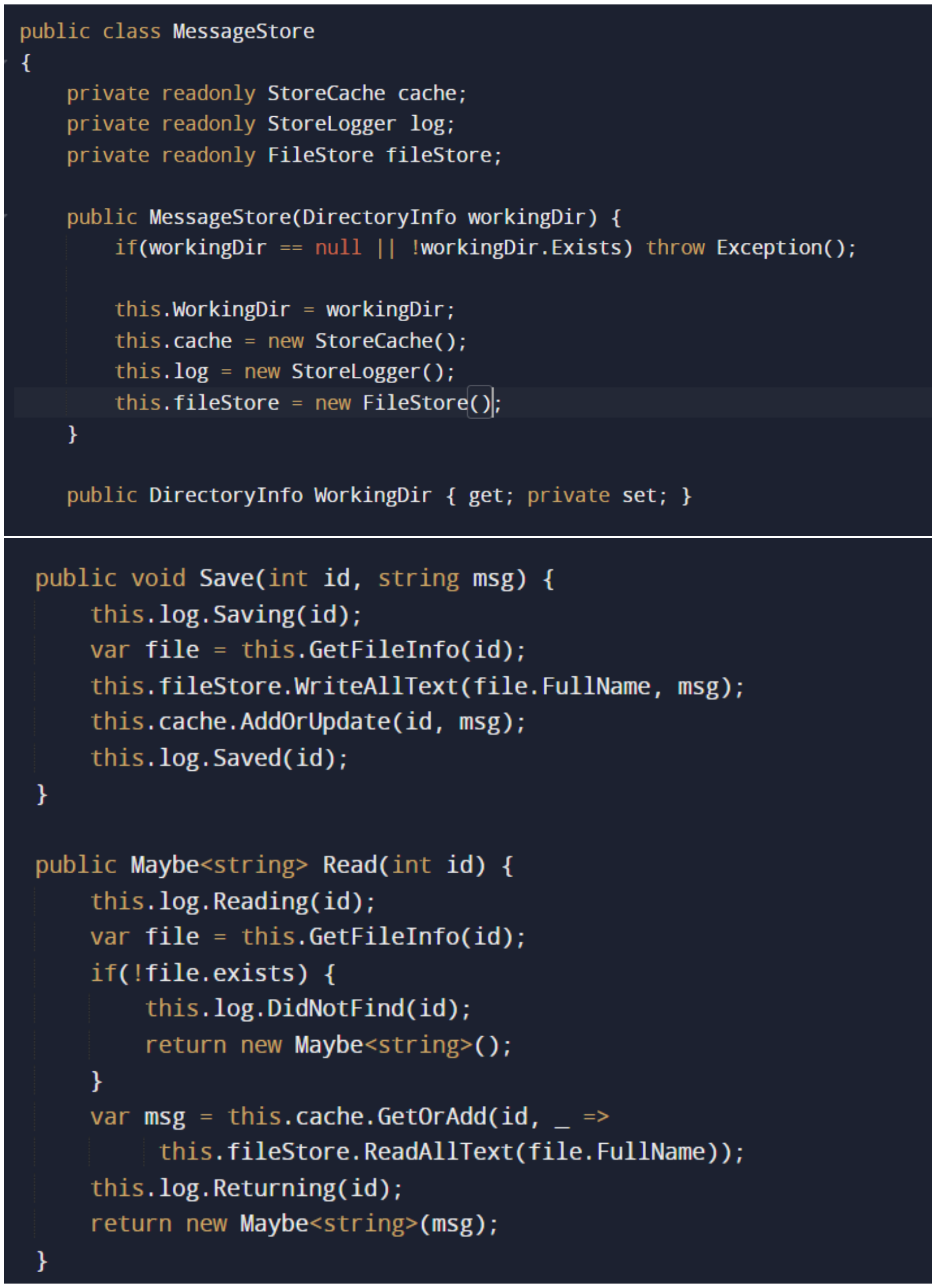
```
public class CashPayment:payment{
    public override void pay(Money money){
        // pay with cash
    }
}

public class CreditCard:payment{
    public override void pay(Money money){
        // pay with credit card
    }
}
```

Now if we need to add another payment logic we only need to create a new class and implement or extends the payment interface or base class. Thus, OCP can be maintained in this example.

3.6 OCP and Our Main Code Example

3.6.1 Existing implementation

After performing SRP in previous section, our code looks like this now -



```
public class MessageStore
{
    private readonly StoreCache cache;
    private readonly StoreLogger log;
    private readonly FileStore fileStore;

    public MessageStore(DirectoryInfo workingDir) {
        if(workingDir == null || !workingDir.Exists) throw Exception();

        this.WorkingDir = workingDir;
        this.cache = new StoreCache();
        this.log = new StoreLogger();
        this.fileStore = new FileStore();
    }

    public DirectoryInfo WorkingDir { get; private set; }
```

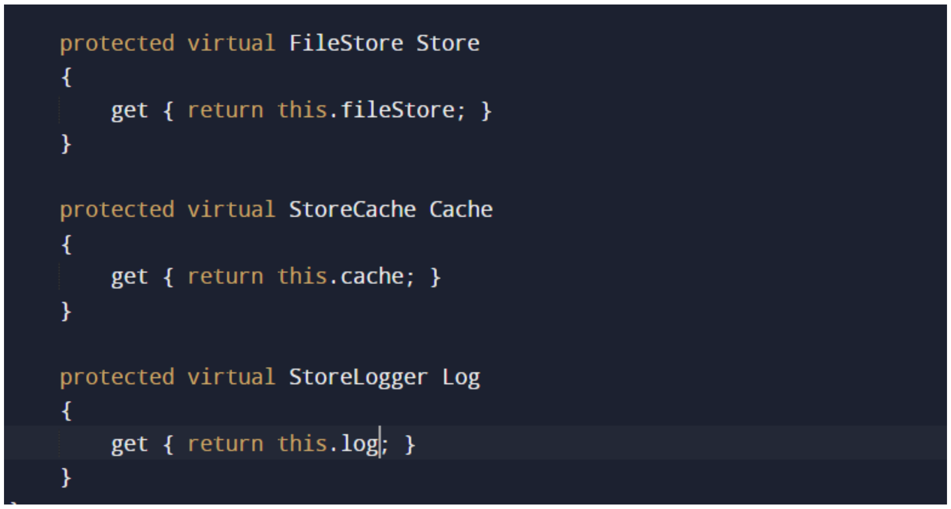
```
public void Save(int id, string msg) {
    this.log.Saving(id);
    var file = this.GetFileInfo(id);
    this.fileStore.WriteAllText(file.FullName, msg);
    this.cache.AddOrUpdate(id, msg);
    this.log.Saved(id);
}

public Maybe<string> Read(int id) {
    this.log.Reading(id);
    var file = this.GetFileInfo(id);
    if(!file.exists) {
        this.log.DidNotFind(id);
        return new Maybe<string>();
    }
    var msg = this.cache.GetOrAdd(id, _ =>
         this.fileStore.ReadAllText(file.FullName));
    this.log.Returning(id);
    return new Maybe<string>(msg);
}
```

Figure 3.2: MessageStore After SRP

3.6.2 Modification

Now we will make a change in our code. Instead of using FileStore which uses the FileSystem for storage, we will now use DBStore which will use a database system as storage. Now, for the OCP principle, we will not be changing the functionality inside the MessageStore class, rather we will extend it to a new Class MessageDBStore which will serve our purpose. But for that, we have to improve some implementation inside the MessageStore, StoreCache, StoreLogger and FileStore class; definitely not by changing any of its existing functionality. We will make these changes only to adopt with our new implementation.

- Firstly we will make all the methods virtual in the StoreCache, StoreLogger and FileStore class (for C# codes only; Java codes will remain same)
- We will implement a new DBStore class as like as FileStore but with a Database support.
- Now, before creating a new MessageDBStore class, we will add some virtual methods in the existing MessageStore class. Right now, as we have to replace the FileStore class with new DBStore class in the constructor. It is not a good practice to modify the constructor. Rather, using our new virtual methods, we will make our class independent of the store classes. So, when we will extend the class, there is no need to modify our constructor or any existing methods. Rather we will replace the virtual methods implementation with new implementation.
  - At the first step, we will add following methods in MessageStore class -

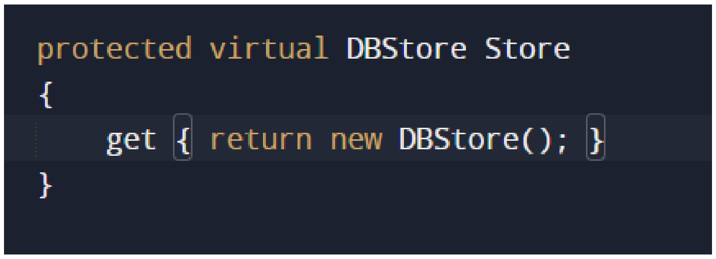


```
protected virtual FileStore Store
{
    get { return this.fileStore; }
}

protected virtual StoreCache Cache
{
    get { return this.cache; }
}

protected virtual StoreLogger Log
{
    get { return this.log; }
}
```

Figure 3.3: MessageStore After adding virtual methods

One more thing to note here that we may create an interface named Store which will be implemented by both FileStore and DBStore class. This will improve our implementation further.

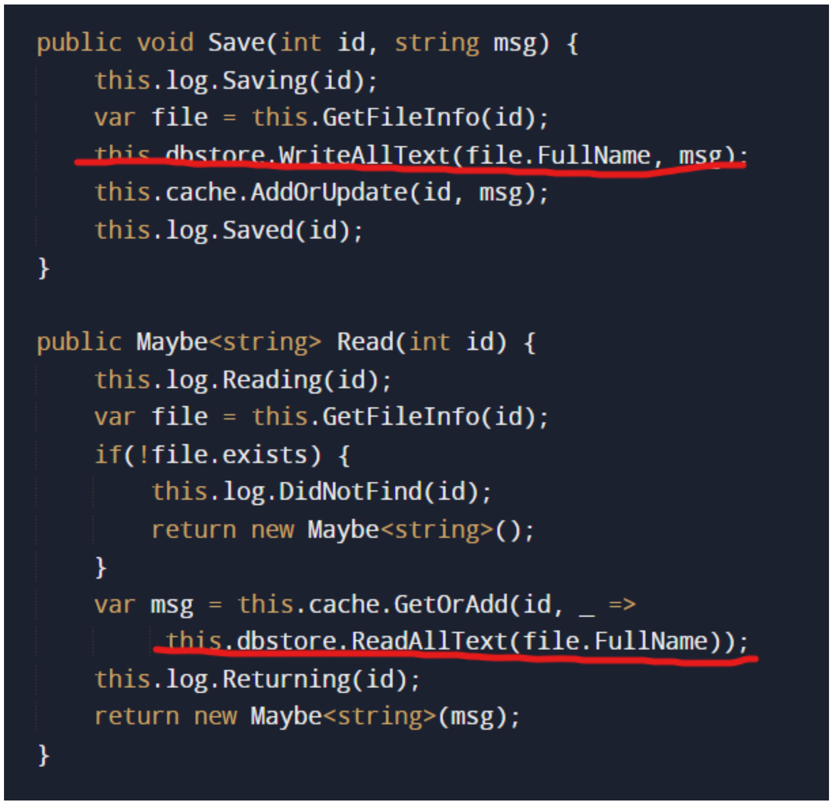
– As the second step, we will replace the fileStore with DBStore



```
protected virtual DBStore Store
{
    get { return new DBStore(); }
}
```

Figure 3.4: Replace FileStore with DBStore

– Finally, Save and Read method will be -



```
public void Save(int id, string msg) {
    this.log.Saving(id);
    var file = this.GetFileInfo(id);
    this.dbstore.WriteAllText(file.FullName, msg);
    this.cache.AddOrUpdate(id, msg);
    this.log.Saved(id);
}

public Maybe<string> Read(int id) {
    this.log.Reading(id);
    var file = this.GetFileInfo(id);
    if(!file.exists) {
        this.log.DidNotFind(id);
        return new Maybe<string>();
    }
    var msg = this.cache.GetOrAdd(id, _ =>
        this.dbstore.ReadAllText(file.FullName));
    this.log.Returning(id);
    return new Maybe<string>(msg);
}
```

Figure 3.5: Modification in Save and Read

N.B: As inheritance was used for OCP implementation in the class lecture, we have also used this here.

Chapter 4

1805006 - LSP and Extra Topics

Student Name: Tanjeem Azwad Zaman Student ID: 1805006

4.1 Definition(s)

The **Liskov Substitution Principle (LSP)** states that:

- *The object of a superclass should be replaceable with the object of a subclass without breaking the application.*

  OR

- *Functions that use pointers to base classes must be able to use objects of derived classes without knowing it*

4.2 Explanation

This principle enforces the correct usage of ***Inheritance*** (one of the OOP principles). In all discussions, superclasses also extend to include "interfaces". Objects of our subclasses must behave the same way as our superclasses. This can be achieved by following a few rules:

- A subclass must properly implement all methods of the parent interface or superclass. If a function of a superclass is not applicable for a subclass that extends/ inherits it, then chances are it will violate the LSP. This is evident when one keeps an inherited function blank, or feels it necessary to throw an exception.
- A method of the superclass overridden in a subclass needs to accept the same input parameter values as the superclass. This means, in the subclass, one can implement less restrictive validation rules in case of input, but is not allowed to enforce stricter ones. Otherwise, any application code that calls this method may get an exception if called with an object of the subclass.
- We can extend this to the return values of the method as well. The return values of the methods of a subclass must be a subset or a more stricter domain of the return values of the superclass. They can also return a subclass of the object returned originally. This will ensure that the code using the method works fine even if the superclass is replaced by the subclass.

4.3 Enforcing LSP

If one decides to apply LSP to their code, the behaviour of their classes gain more importance than the structure. As with most of the other SOLID principles, ther's no easy way to check this. The compilers and interpreters only check the structural rules of the code, and cannot enforce a particular "Behaviour".

One needs to apply their own checks to ensure proper implementation of LSP. This can best be done through *Code Reviews* and *Test Cases*. In the tests, one can run a specific part of the application code by replacing all the superclass objects with objects of all possible subclasses. Then, make sure that no case returns an error nor significant performance issues are incurred. Similar checks can also be done in Code Reviews. But what's more important is making sure that all the required test cases are created and executed.

LSP violations, once detected, can be solved by applying the ***Interface Segregation Principle (ISP)***, as discussed in the next section.

4.4 Concept in Code

```
// Interface
Interface IClass{
   f1();
   f2();
   f3();
}

// Suppose, new class implements ClassI, but cannot logically use f3
Class1 implements IClass {
   f1() {
      // implement f1
   }
   f2() {
      // implement f2
   }
   f3() {
      // not applicable for Class1. Returns null or throws exception
   }
}
```

4.5 Simple Examples

Some Simple examples of LSP violations (and their solutions) are provided below:

4.5.1 The Bird-Penguin Problem

Violating the LSP

The main theme here is that penguins are birds that cannot fly. So extending a generic bird class to form the penguin class will pose some issues.

```java
public class Bird {
    public void fly() {
        System.out.println("Flying");
    }
    public void walk() {
        System.out.println("Walking");
    }
}

public class Dove extends Bird{} //OK, because doves can both fly and walk
public class Penguin extends Bird{} //NOT OK, penguins can't fly
```

Adhering to LSP

In this implementation, penguins ARE birds, but they should not be able to fly. So LSP is violated. It can be corrected as follows:

```java
public class Bird {
    public void walk() {
        System.out.println("Walking");
    }
}

public class flyingBird extends Bird{
    public void fly() {
        System.out.println("Flying");
    }
}

public class Dove extends flyingBird{} //OK, because doves can both fly and walk
public class Penguin extends Bird{} //OK, penguins can't fly, but is a bird
```

This is basically implementing ISP (details in next chapter). This is how code violating LSP can be modified to correct itself.

4.5.2 The Rectangle-Square Problem

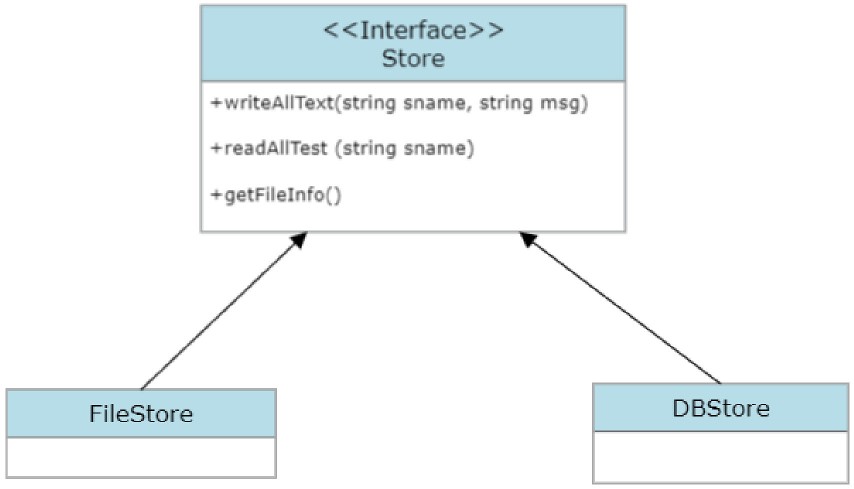
The classic case of extending a rectangle class to form the square class, because *"A square is just a rectangle with all its width equal to its height"* violates the LSP.

```
class Rectangle{
   public void setWidth(int w){
        this.width = w;
   }
   public void setHeight(int h){
        this.height = h;
   }
   public int getArea(){
        return this.width * this.height;
   }
}

class Square extends Rectangle {
    @Override
    public void setWidth(int w){
        super.setwidth(w);
        super.setheight(w);
    }
    @Override
    public void setHeight(int h){
        super.setwidth(h);
        super.setheight(h);
    }
}
//testcode
void testMethod (Rectangle r){
    r.setwidth(5);
    r.setheight(4);
    assert(r.getarea() == 20);
}
```

Here, passing a ***Rectangle*** object will pass the assertion, but passing a ***Square*** object fails, since *getarea()* returns 16. Thus this code fails to adhere to the LSP.

4.6 LSP and our Main Code Example

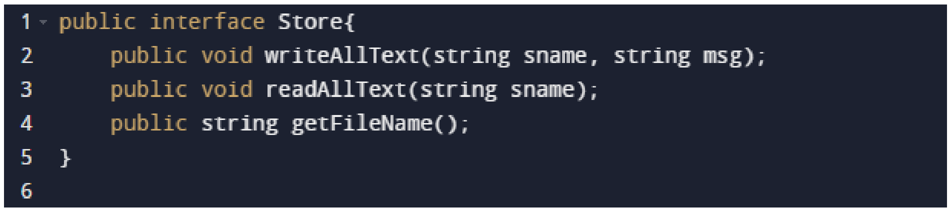
As things stand, after our code was modified to follow OCP, we had the following hierarchy, where both classes ***DBStore*** and ***FileStore*** implement the ***store*** interface:



<<Interface>>
Store
+writeAllText(string sname, string msg)
+readAllTest (string sname)
+getFileInfo()
FileStore
DBStore

Figure 4.1: Store Interface hierarchy after OCP

The code implementation of the class diagram above is as follows:

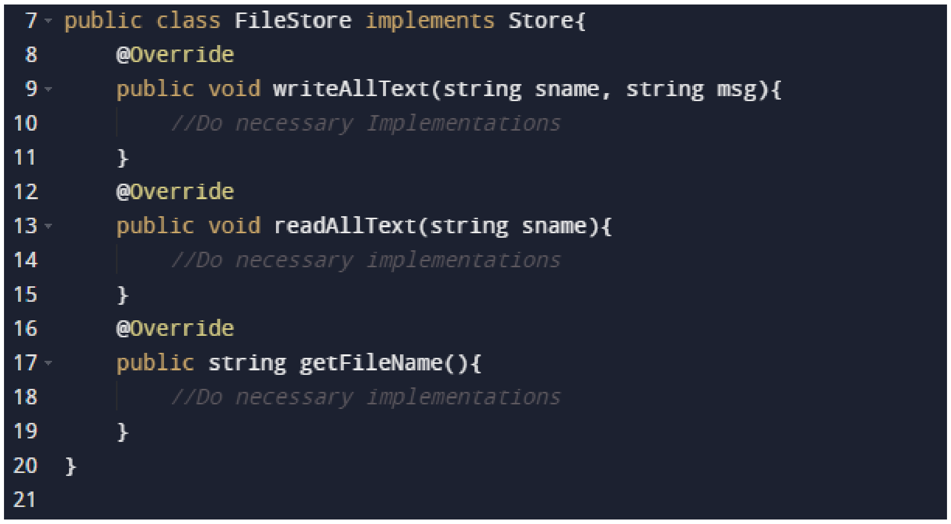


```
public interface Store{
    public void writeAllText(string sname, string msg);
    public void readAllText(string sname);
    public string getFileName();
}

```

Figure 4.2: Store Interface

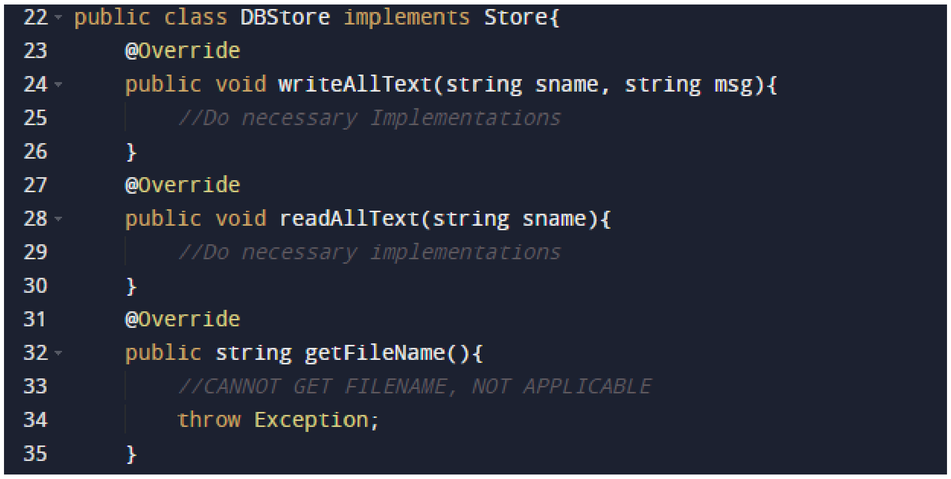
- This store interface will be implemented by both the FileStore and DBStore classes.
- But, for a Database represented by a DBStore object, *getFileName()* is not a meaningful function.
- But DBStore MUST override this function, as it has been defined in the implemented interface.
- This leaves us no choice but to throw an exception in this case.

Code implementation of the DBStore and FileStore classes:



```
public class FileStore implements Store{
    @Override
    public void writeAllText(string sname, string msg){
        //Do necessary Implementations
    }
    @Override
    public void readAllText(string sname){
        //Do necessary implementations
    }
    @Override
    public string getFileName(){
        //Do necessary implementations
    }
}

```

Figure 4.3: FileStore Class



```
public class DBStore implements Store{
    @Override
    public void writeAllText(string sname, string msg){
        //Do necessary Implementations
    }
    @Override
    public void readAllText(string sname){
        //Do necessary implementations
    }
    @Override
    public string getFileName(){
        //CANNOT GET FILENAME, NOT APPLICABLE
        throw Exception;
    }
```

Figure 4.4: DBStore Class

- Thus, any usage of ***dbStore.getFileName()*** will lead to exception being thrown.
- This is same as replacing usage of ***Store.getFileName*** with ***dbStore.getFileName()*** and getting error, which is violating LSP
- This can be fixed by Interface Seperation Method, which will be shown in the next chapter.

4.7 Extra

4.7.1 Bad Practices

Static Variables

Static variables are undesirable. Except for very few cases where static is unavoidable, it is best to avoid static variables in Software. Static is used in main functions, and most of the time developers don't even know where the main function resides in a web application.

Instantiating ("new")

- Don't instantiate objects in business classes.
- Using "new" in a business class automatically "tightly couples" the class with the instantiated object.
- Such instantiations make it difficult to implement changes later on.
- Making a new business object directly in another business class also carries the same risk of tight coupling
- New objects should be declared in Data classes or using Factories.

4.7.2 Business Class and Data Class

Business Class

- The classes where the main business logic is implemented and where devs work on
- Usually, outside of classes already implemented by frameworks, or has separate placeholders for implementation in frameworks
- Implements the main functionalities to be provided by the application
- Opposed to Data classes for representing Database rows as objects in ORM (Object Relational Model) structure
- Decouple 3rd party classes and functions from Business classes by using wrapper classes/functions.

Data Class

- Consists of only variables
- Needed to represent Database rows as objects.
- Fields are usually a one to one mapping with the columns of the DB table.

4.7.3 Class Readability

- Ideally 1-2 functions per class
- 5-10 lines per function
- Many (hundreds of) classes per application
- Easier and less stressful to follow flow, as opposed to 1 class with 500 lines

4.7.4 Lazy Class

- Anticipate new features and preemptively separate into a new class.
- Or, due to refactoring, a class becomes ridiculously small
- Class does not see any change or any significant usage in a long time (2-3 years)
- Termed as Lazy Class

```
public class Message {
    private string msg;
    public string getMsg(){
        return msg;
    }
    public void setMsg(string m){
        this.msg = m
    }
}
```

This class is basically a string. And it can be implemented seperately as a field, instead of a class. This will eventually turn into a lazy class.

Chapter 5

1805008 - ISP

Student Name: Abdur Rafi Student ID: 1805008

5.1 Definition

The ISP (Interface Segregation Principle) states that " Clients should not be forced to depend upon methods that they do not use." Here clients refer to the classes that implement an interface. So, if a class does not use a method of an interface, then it should not be affected in any way by any change to that method.

5.2 Context

The ISP addresses the issues that come form "polluted" interfaces. These interfaces usually do not follow the single responsibility principle. They either try to capture the behavior of multiple clients or contain methods that are specific to some clients, not all of them. As a result, some methods remain unused by some clients, or do not make sense for some clients.

5.2.1 Birth of "polluted" interfaces

Usually developers do not willingly write bad code. Clean interfaces become polluted over time, due to additions or changes. Let us consider an example to see how these interfaces are born and the issues related with them.

```
// Interface of a food Order
Interface FoodOrder{
    void addFoodItem(Food item);
    double getCost();
    void printMemo();
}
// Orders from customers inside the shop
Class InHouseOrder implements FoodOrder {
    void addFoodItem(Food item){}
    double getCost(){}
    void printMemo(){}
}
```

Suppose, this is part of a system for a new fast food shop. The shop starts with only in-house orders, no delivery or takeaways. The code is perfectly reasonable for now.

After some time, the shop becomes quite popular. So, the owner decides to take orders form phone calls as well. So, a new class PhoneCallOrder is added. Phone call orders are obviously orders, but they have an additional behavior. They should allow assignment of delivery man. To support this additional behavior, the developers may decide to add a new method to the FoodOrder interface.

```
// Interface of a food Order
Interface FoodOrder{
    ...
    // new method for orders through phone calls
    void assignDeliveryMan(Employee e);
}
// Orders from phone calls
Class PhoneCallOrder implements FoodOrder {
    void addFoodItem(Food item){}
    double getCost(){}
    void printMemo(){}
    void assignDeliveryMan(Employee e){}
}

Class InHouseOrder implements FoodOrder {
    ...
    // In house orders do not need delivery man
    void assignDeliveryMan(Employee e){
        // throws error
    }
}
```

After some more time, the owner decides to take online orders through Apps. A new class OnlineOrder is added. Online orders should send their status to the app to keep the customers updated.

```
// Interface of a food Order
Interface FoodOrder{
    ...
    // new method for online orders
    void sendStatus();
}
Class OnlineOrder implements FoodOrder{
    void addFoodItem(Food item){}
    double getCost(){}
    void printMemo(){}
    void assignDeliveryMan(Employee e){}
    void sendStatus(){}
}
// Orders from phone calls
Class PhoneCallOrder implements FoodOrder {
    ...
    // not applicable for phone call orders
    void sendStatus(){
        // throws error
    }
}
Class InHouseOrder implements FoodOrder {
    ...
    // not applicable for in house orders
    void sendStatus(){
        // throws error
    }
}
```

The FoodOrder interface has now become polluted due to new changes and improper handling of those changes. It contains 2 methods that are specific to some clients, not all. As a result, some clients have to place dummy implementations of those method and throw error or return null from the methods.

5.2.2 Problem of Polluted Interfaces

Suppose the sendStatus method's signature is changed; maybe the return type is changed or more paramters are added.Whatever it is, the other two classes, InHouseOrder and PhoneCallOrder are now need to be changed. This is the main issue of polluted interfaces. Even though no actual change of the classes InHouseOrder and PhoneCallOrder is made, their code has to be changed.

In a large system, this situation can become much worse where a lot of classes implement the polluted interface. In such scenarios, change of one method causes change in many classes which have nothing to do with that method. This is error prone, hard to maintain, and a waste of human resources.

5.2.3 The Solution-Interface Segregation

ISP provides solution to this scenario. ISP suggests breaking the interface into multiple (related or unrelated) interfaces. The smaller interfaces will capture only the general methods or the behavior of a single client. The clients will implement interfaces that capture their desired methods.

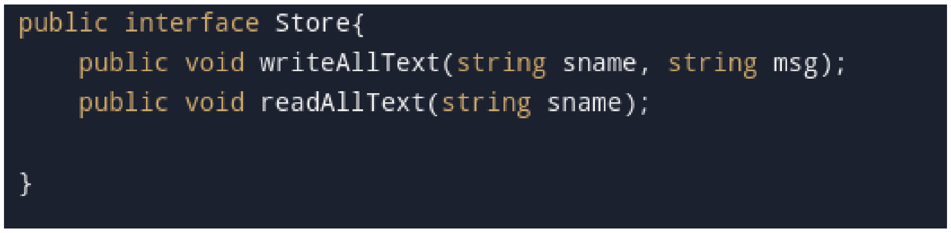
```
Interface FoodOrder{
    void addFoodItem(Food item);
    double getCost();
    void printMemo();
}
Interface OutsideOrder extends FoodOrder{
    void assignDeliveryMan(Employee e);
}
Interface IOnlineOrder extends OutsideOrder{
    void sendStatus();
}

Class InHouseOrder implements FoodOrder{
    ...
}
Class PhoneCallOrder implements OutsideOrder{
    ...
}
Class OnlineOrder implements IOnlineOrder{
    ...
}
```

Now changing the sendStatus method only cause change in the OnlineOrder class, nowhere else.

5.3 ISP and Our Main Code Example

As mentioned in 4.6, ISP can be applied to fix the violation of LSP in our example.

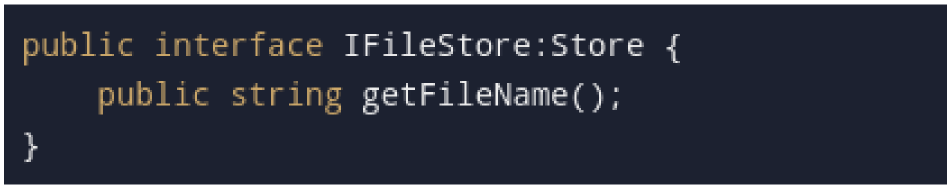
To do so, we will break down the Store interfaces into 2 interfaces. One will capture the general methods of all types of stores and another will capture the specific methods of FileStore, plus the general methods (by extending former interface).



```
public interface Store{
    public void writeAllText(string sname, string msg);
    public void readAllText(string sname);

}
```

Figure 5.1: Store Interface after ISP

Store interface captures the general methods for all types of stores.

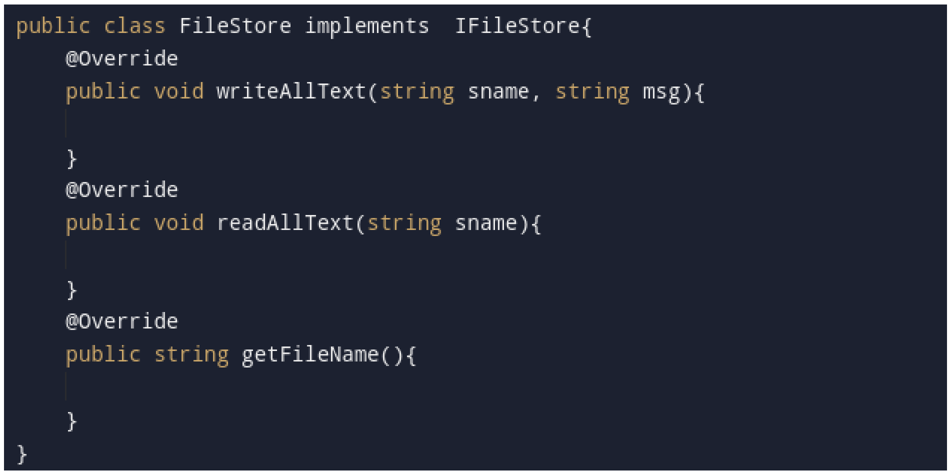


```
public interface IFileStore:Store {
    public string getFileName();
}
```

Figure 5.2: New IFileStore Interface

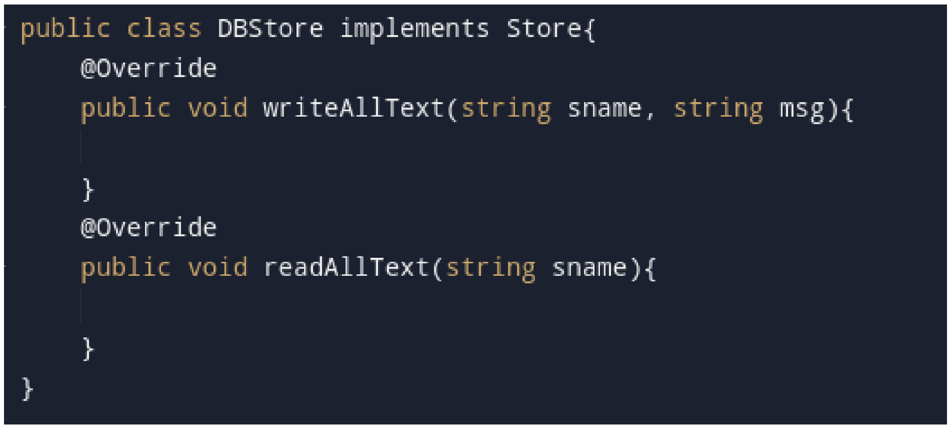
IFileStore captures methods that are specific to FileStore (getFileName, in this case). It also extends the Store interface to retain the common methods.

FileStore class now implements the IFileStore interface and DBStore class now implements the Store interface.



```
public class FileStore implements  IFileStore{
    @Override
    public void writeAllText(string sname, string msg){

    }
    @Override
    public void readAllText(string sname){

    }
    @Override
    public string getFileName(){

    }
}
```

Figure 5.3: FileStore Class after ISP



```
public class DBStore implements Store{
    @Override
    public void writeAllText(string sname, string msg){

    }
    @Override
    public void readAllText(string sname){

    }
}
```

Figure 5.4: DBStore Class after ISP

Now this portion of the code does not violate the Liskov Substitution Principle.

Chapter 6

1805019 - DIP

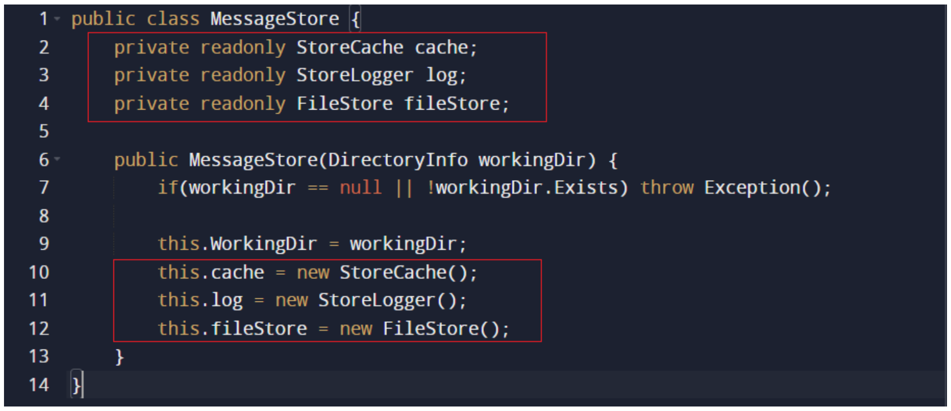
Student Name: Rownok Ratul Student ID: 1805019

6.1 Definition

DIP, **Dependency Inversion Principle**, is the concept of class extension without having a dependency towards a specific set of classes, that the composition uses. Usually, such dependency arises when SRP and OCP principles have been applied. In OCP, composition is preferred over inheritance. But composition stages multiple class instances as a part of the composed class. This creates a dependency from composed class to the older classes. Dependency inversion principle allows such dependency to be removed, producing more lucid code-flows.

6.2 Example Scenario

Lets have a look at the "MessageStore" class, that have been composed after applying SRP and OCP.



```
public class MessageStore {
    private readonly StoreCache cache;
    private readonly StoreLogger log;
    private readonly FileStore fileStore;

    public MessageStore(DirectoryInfo workingDir) {
        if(workingDir == null || !workingDir.Exists) throw Exception();

        this.WorkingDir = workingDir;
        this.cache = new StoreCache();
        this.log = new StoreLogger();
        this.fileStore = new FileStore();
    }
}
```

Figure 6.1: MessageStore Class

This design separates the caching, logging and storage from business logic codes with independent classes. A class diagram for such scenario is the following one. The problem in such a design is revealed when Storage scheme is likely to be changed. As such, if storage scheme is changed to database, then "MessageStore" existing class is required to be modified, a flaw in the design. DIP addresses such scenarios and uses dependency injection to revert such flaws.

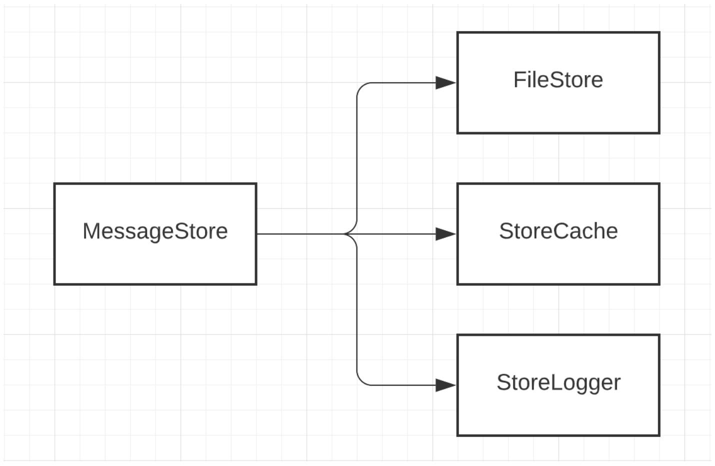
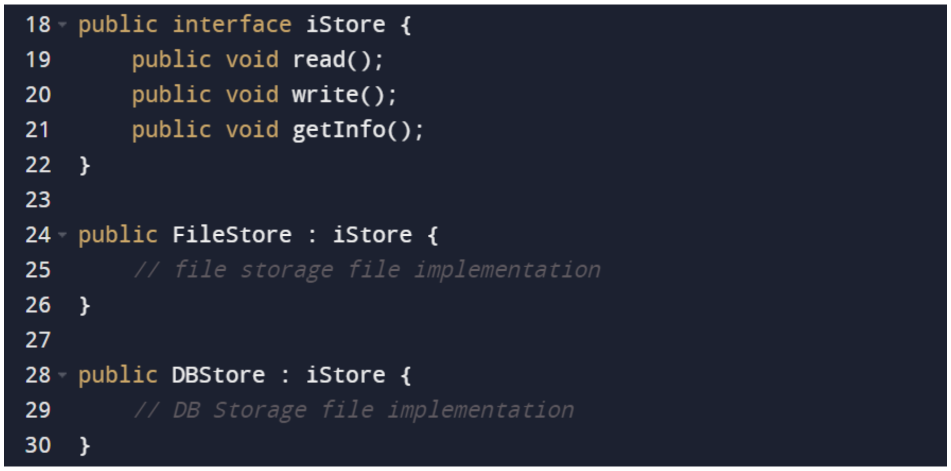


Figure 6.2: Dependency Graph

6.3 Applying DIP

Addressing the scenario, firstly for storage, a general interface is needed to be introduced. Here, we introduce "iStore" interface for a generalized case. Required classes, like "FileStore" implements this interface. This interface will also allow any other storage scheme to be implemented without further modification.

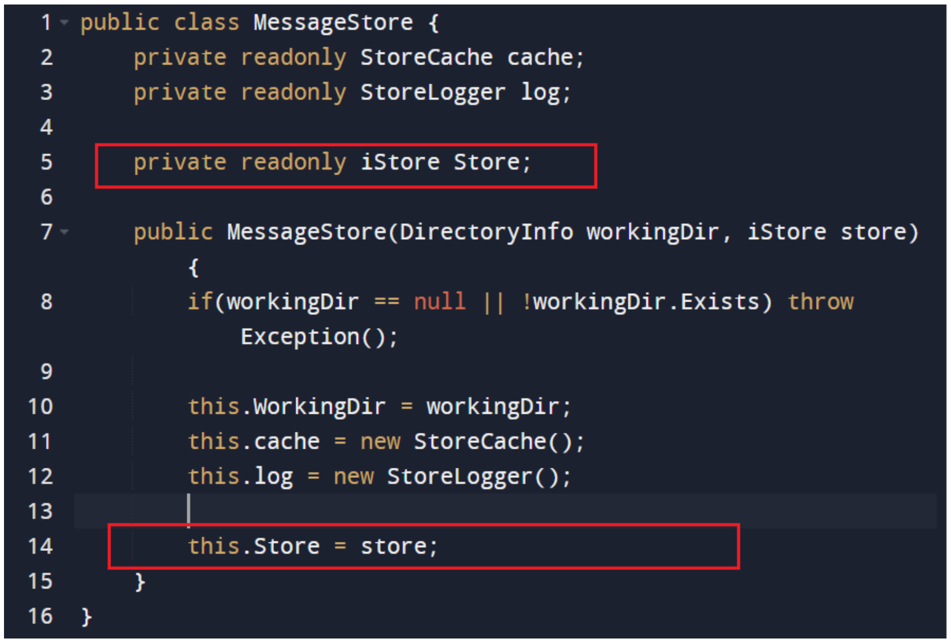


```
public interface iStore {
    public void read();
    public void write();
    public void getInfo();
}

public FileStore : iStore {
    // file storage file implementation
}

public DBStore : iStore {
    // DB Storage file implementation
}
```

Figure 6.3: iStore interface

6.3.1 Dependency Injection

Now, ”MessageStore” acknowledges the situation with a reference to the interface ”iStore”. The user passes a reference to the required class using the constructor. The reference is such runtime binded with appropriate class for usage. This scheme is known as **dependency injection**.



```
public class MessageStore {
    private readonly StoreCache cache;
    private readonly StoreLogger log;

    private readonly iStore Store;

    public MessageStore(DirectoryInfo workingDir, iStore store)
        {
        if(workingDir == null || !workingDir.Exists) throw
            Exception();

        this.WorkingDir = workingDir;
        this.cache = new StoreCache();
        this.log = new StoreLogger();

        this.Store = store;
    }
}
```

Figure 6.4: Dependency Injection

6.3.2 Dependency Inversion

Dependency injection allows a reverted dependency on the interface rather than on the class. Here, ”DBStore” class implements the ”iStore” interface and thus, any of the abstracted functionality can be used without having detailed implementation of ”Storage” scheme. ”MessageStore” poses no direct dependency on ”FileStore”, rather user of the class decides the ”Storage” scheme. This allows the dependency of ”FileStore” and ”DBStore” towards ”iStore”. Following figure depicts the inversion caused by the DIP.

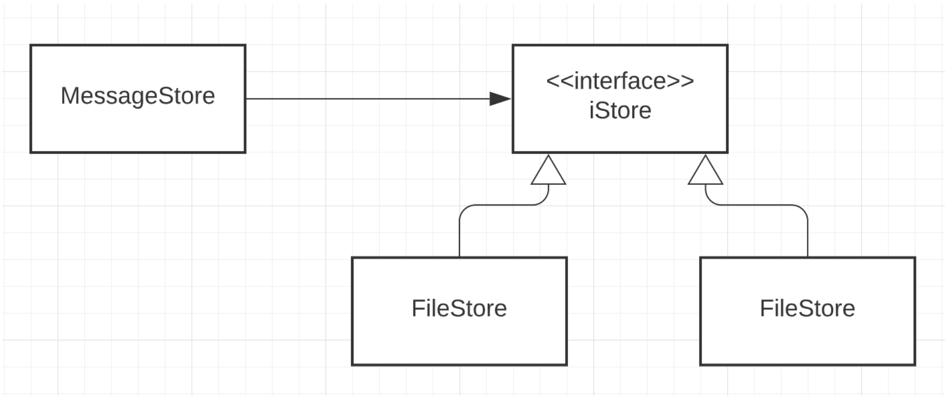


Figure 6.5: Dependency Inversion

6.4 Effectiveness of DIP

Dependency Inversion, in a indirect way, broadens the extensibility of a class for modification. As such, we can add any ”Storage” scheme that implements iStore, without having to modify the ”Message-Store” class. Also, ”MessageStore” does not need to be aware of any such extension, as it is dynamically reverted at runtime. This causes ”MessageStore” to work properly without any knowledge of storage functionality.
Notwithstanding, DIP introduces a new problem of interface overlap, other SOLID principles (LSP and ISP) effectively solves them in a constructive way. For example, when storing in ”DBStore”, the function ”getInfo” in ”iStore” interface may convey no meaning. Thus implementing such a function for ”DBStore” is meaningless compared to ”FileStore” that contains information about file directory. Such problem addresses interface segregation, seemingly solved by ISP.