SOLID-ПРИНЦИПИ РОЗРОБКИ ОБ’ЄКТНО-ОРІЄНТОВАНОГО КОДУ

Питання 6.3.

Розуміння SOLID

- Вимоги до ПЗ ніколи не записуються в повному та остаточному вигляді.
  - SOLID-принципи пропонують так організовувати код, щоб внесення змін у нього через зміну вимог було максимально безболісним.
- SOLID НЕ є фреймворком чи бібліотекою, тобто відсутня прив'язаність до конкретних технологічних стеків.
  - Також це НЕ шаблон проектування.
  - НЕ є загальною метою зробити код у повністю SOLID-ним, оскільки неможливо виміряти ступінь відповідності SOLID-принципам.
- Багато сучасного коду, написаного за допомогою об'єктно-орієнтованих мов програмування, все-таки мають процедурно-орієнтовану природу.
  - SOLID-принципи стосуються саме об'єктно-орієнтованого проектування та можуть суттєво відрізнятись від Вашого коду, написаного об'єктно-орієнтованою мовою.
  - Мета SOLID – зробити код більш супроводжуваним (maintainable) на основі декомпозиції та декаплінгу (розділення, decoupling).

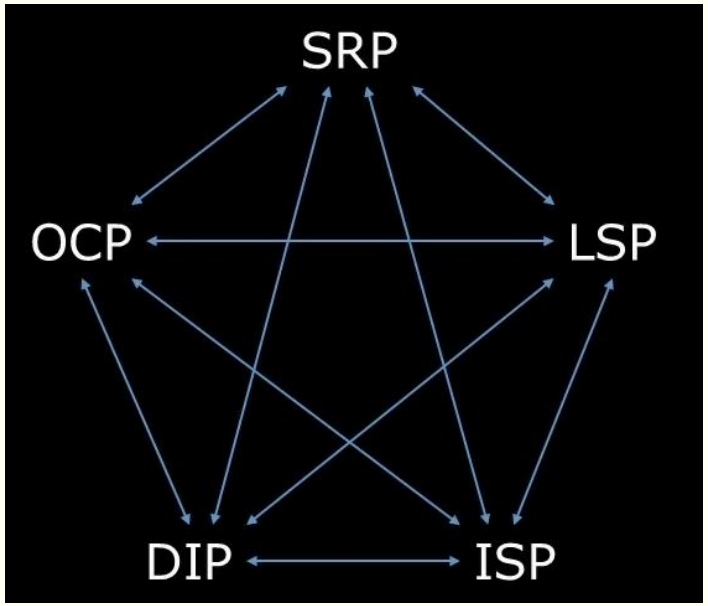
Розуміння SOLID

- Застосування SOLID-принципів відбувається на основі виявлення порушень об'єктно-орієнтованого дизайну (дизайну з душком, design smells).

- Недоліки коду, які адресуються до SOLID-принципів:
  - ***Жорсткість (Rigidity)***: в програмну систему складно вносити зміни в багатьох напрямках
  - ***Крихкість (нестійкість, fragility)***: незначні зміни в одній частині програми викликають проблеми в інших частинах системи, навіть напряму не пов'язаних зі зміненим компонентом.
  - ***Нерухомість (immobility):*** код складно повторно використовувати, що може стати проблемою.
  - ***В'язкість (viscosity):*** внесення змін щодо певного аспекту програмної системи ускладнене і може порушувати закладені в проект принципи.
  - ***Непотрібна складність (needless complexity, overdesign):*** намагання узагальнити код відразу в процесі його первинного створення, що не підкріплено вимогами чи потребами системи.

- Подальші демонстрації SOLID-принципів здійснюються в дуже спрощеному коді.

Розуміння SOLID



- **S**ingle responsibility Principle (принцип єдиної відповідальності, SRP) – кожний клас повинен мати тільки одну відповідальність.
- **O**pen Closed Principle (принцип відкритості-закритості, OCP) – клас повинен бути відкритим для розширення, проте закритим до змін.
- **L**iskov Substitution Principle (принцип підстановки Барбари Лісков, LSP) – описує, як поліморфізм має (не обов'язково повинен) працювати.
- **I**nterface Segregation Principle (принцип відокремлення інтерфейсу, ISP) – описує, як повинні проектуватись інтерфейси взаємодії.
- **D**ependency Inversion Principle (принцип інверсії залежностей, DIP) – описує зв'язки між абстрактними та конкретними типами.
  - Першість принципу в абревіатурі НЕ свідчить про його більшу важливість.
  - Принципи пов'язані між собою, тому впровадження тільки одного з них надасть обмежені переваги. Вони працюють єдиним пакетом.

Код перед рефакторингом

```
public class Maybe<T> : IEnumerable<T>
{
    private readonly IEnumerable<T> values;

    public Maybe()
    {
        this.values = new T[0];
    }

    public Maybe(T value)
    {
        this.values = new[] { value };
    }

    public IEnumerator<T> GetEnumerator()
    {
        return this.values.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return this.GetEnumerator();
    }
}
```

```
public class FileStore
{
    public FileStore(string workingDirectory)
    {
        if (workingDirectory == null)
            throw new ArgumentNullException("workingDirectory");
        if (!Directory.Exists(workingDirectory))
            throw new ArgumentException("Boo", "workingDirectory");

        this.WorkingDirectory = workingDirectory;
    }

    public string WorkingDirectory { get; private set; }

    public void Save(int id, string message)
    {
        var path = this.GetFileName(id);
        File.WriteAllText(path, message);
    }

    public Maybe<string> Read(int id)
    {
        var path = this.GetFileName(id);
        if (!File.Exists(path))
            return new Maybe<string>();
        var message = File.ReadAllText(path);
        return new Maybe<string>(message);
    }

    public string GetFileName(int id)
    {
        return Path.Combine(this.WorkingDirectory, id + ".txt");
    }
}
```

Код перед рефакторингом

```
public void Save(int id, string message)
{
    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("Reading message {id}.", id);
    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);
}
```

- Додамо логування до методів Save() і Read().
  - Також доповнимо код кешуванням (4й рядок методу Save() та для змінної message у методі Read()).
  - Подібні доповнення досить поширення в кодових базах C#-проектів.
  - Зауважте, що це лише спрощена версія коду, в реальності вона виглядатиме складніше.
  - Таким чином, початкова реалізація «розмиється» в супроводжуючому коді.

Принцип єдиної відповідальності

- Відповідальність (responsibility) – це причина для зміни.
  - Клас повинен мати єдину причину для зміни.
  - Опираємось на ідею розподілу відповідальностей (separation of concerns): відокремлення базової перевірки коректності вводу від кешування чи іншої логіки в класі.
  - Кожний клас повинен робити одну річ, і робити її добре!
  - За таким принципом побудовані UNIX-подібні ОС.

- Причини для зміни в попередньому коді:
  - Логування
  - Кешування
  - Механізм зберігання повідомлень (зчитування/запис з інших джерел)
- Неявною причиною для зміни є координування (orchestration) діяльності в контексті попередніх причин для зміни.
  - Встановлює порядок виконання послідовності дій.
  - Також можлива деяка умовна логіка для здійснення окремих дій.
  - Запропонований код не включає бізнес-логіку, яка теж може входити в загальну інфраструктуру.

Принцип єдиної відповідальності

```
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.Debug("Reading message {id}.", id);
    }

    public void DidNotFind(int id)
    {
        Log.Debug("No message {id} found.", id);
    }

    public void Returning(int id)
    {
        Log.Debug("Returning message {id}.", id);
    }
}
```

- Виділимо логіку логування в окремий клас StoreLogger.
  - Особливо корисно, якщо фреймворк для логування може змінитись з часом.
  - Тоді всі зміни можна вносити в межах одного класу.

```
public void Save(int id, string message)
{
    new StoreLogger().Saving(id);
    var file = this.GetFileInfo(id);
    File.WriteAllText(file.FullName, message);
    this.cache.AddOrUpdate(id, message, (i, s) => message);
    new StoreLogger().Saved(id);
}

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

Принцип єдиної відповідальності

- Створення нових об’єктів StoreLogger у методах класу є надмірним, тому введемо приватне, тільки для зчитування поле log.
  - Набагато кращим даний код поки що не став, його об’єм у класі FileStore залишився рівноцінним.

```
public class FileStore
{
    private readonly ConcurrentDictionary<int, string> 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 ConcurrentDictionary<int, string>();
        this.log = new StoreLogger();
    }

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

```
public void Save(int id, string message)
{
    this.log.Saving(id);
    var file = this.GetFileInfo(id);
    File.WriteAllText(file.FullName, message);
    this.cache.AddOrUpdate(id, message, (i, s) => 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);
}
```

Принцип єдиної відповідальності

```
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);
    }
}
```

- Застосуємо принцип до інших відповідальностей.
  - Виділимо клас StoreCache для визначення процесу кешування.
  - Словник з конкурентним доступом (тип ConcurrentDictionary<TKey, TValue>) є причиною таких дивних методів для роботи з кешем.
  - Код класу FileStore на даний момент практично не вимагає змін: слід лише змінити тип об’єкта cache на StoreCache замість ConcurrentDictionary<int, string>.
  - Словник з конкурентним доступом не є хорошою реалізацією для кешування, а показаний лише як демонстрація роботи.

Принцип єдиної відповідальності

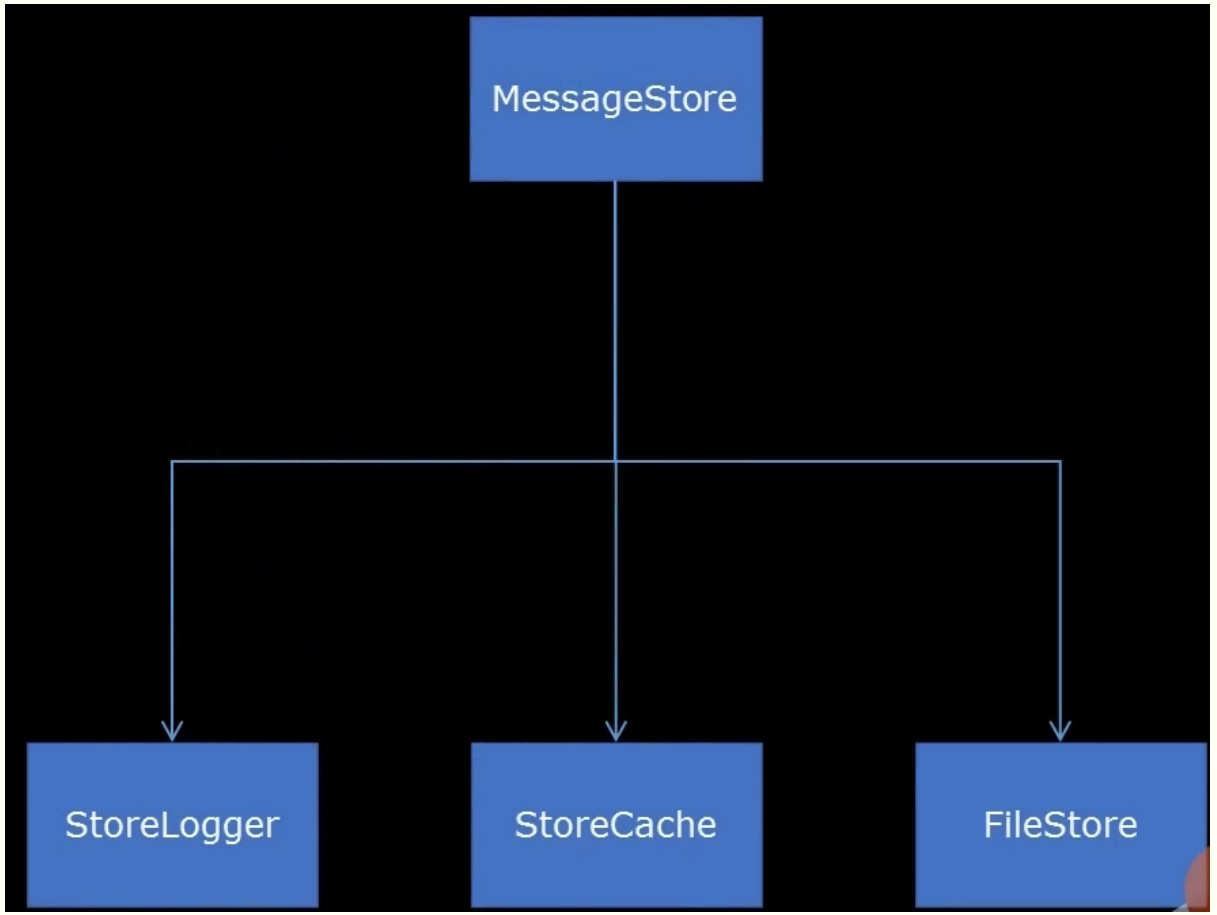
```
public class MessageStore
{
    private readonly StoreCache cache;
    private readonly StoreLogger log;

    public MessageStore(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 DirectoryInfo WorkingDirectory { get; private set; }

    public void Save(int id, string message)
    {
        this.log.Saving(id);
```

- Застосуємо принцип до інших відповідальностей.
  - Замінимо назву класу FileStore на MessageStore, щоб відобразити загальність опису механізму зберігання повідомлень (не тільки у файлову систему, але, можливо, в БД чи інше сховище даних).
  - Також перейменування дасть можливість створити клас FileStore, націлений на роботу саме з файлами:

```
public class FileStore
{
    public void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }

    public string ReadAllText(string path)
    {
        return File.ReadAllText(path);
    }

    public FileInfo GetFileInfo(int id, string workingDirectory)
    {
        return new FileInfo(
            Path.Combine(workingDirectory, id + ".txt"));
    }
}
```

Архітектура у результаті застосування SRP

```
public void Save(int id, string message)
{
    this.log.Saving(id);
    var file = this.GetFileInfo(id);
    this.fileStore.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, _ => this.fileStore.ReadAllText(file.FullName));
    this.log.Returning(id);
    return new Maybe<string>(message);
}
```



Ще раз про проблему непотрібного ускладнення

- Отриманий код у деякій мірі надмірно ускладнений, оскільки принцип єдиної відповідальності застосовувався у відриві від решти SOLID-принципів.
- Загалом, розробники часто намагаються розв'язати конкретні завдання за допомогою універсальних рішень.
  - Це веде до посилення зв'язаності (каплінгу, coupling) та підвищення складності.
  - Замість бути універсальним, код має вирішувати конкретну задачу.
  - Кожний конкретний клас повинен описувати вирішення конкретного завдання, як у попередньому коді.
- А якщо універсальність потрібна?
  - Висока конкретика може вести до дублювання коду.
  - Якщо в коді знайдено дублювання частини функціональності в різних місцях, можна виділити ***інтерфейс***.

Header Interfaces vs Role Interfaces

- Мартін Фаулер виділяє 2 різновиди інтерфейсів:
- ***Header-інтерфейс*** – це явний інтерфейс, який імітує неявний публічний інтерфейс взаємодії з класом.
  - Описується шляхом взяття всіх публічних методів класу та їх оголошення в інтерфейсі.
  - Після цього можна постачати альтернативну реалізацію для класу.
- ***Рольовий інтерфейс*** визначається при конкретній взаємодії між постачальниками та споживачами, зазвичай має небагато методів.
  - Компонент-постачальник зазвичай реалізуватиме кілька рольових інтерфейсів, зазвичай, по одному на кожний паттерн взаємодії.
  - Це відрізняє рольовий інтерфейс від заголовкового (header), в якому постачальник матиме єдиний інтерфейс
- Детальніше порівняння різновидів інтерфейсів.
- Чи потрібно уникати заголовкових інтерфейсів?

```
public class FileStore : IStoreWriter, IStoreReader, IFileLocator
{
    public void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }

    public string ReadAllText(string path)
    {
        return File.ReadAllText(path);
    }

    public FileInfo GetFileInfo(int id, string workingDirectory)
    {
        return new FileInfo(
            Path.Combine(workingDirectory, id + ".txt"));
    }
}
```

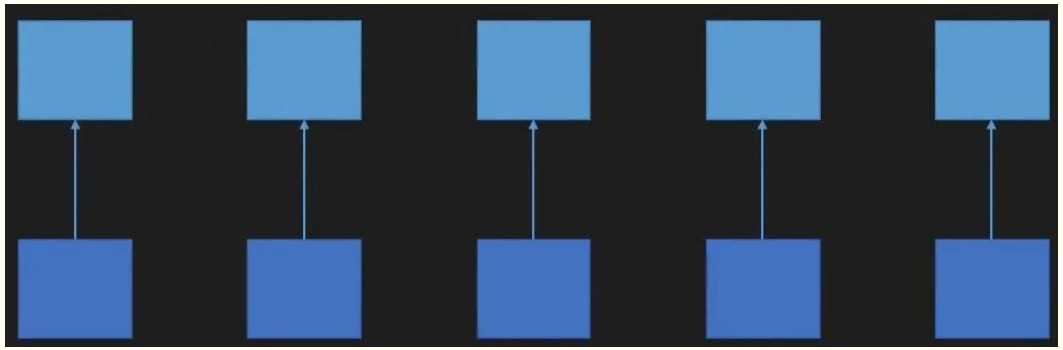
- Виділимо в коді рольові інтерфейси
  - IStoreWriter, який передбачатиме реалізацію лише методу WriteAllText();
  - IStoreReader, який передбачатиме реалізацію лише методу ReadAllText();
  - IFileLocator, який передбачатиме реалізацію лише методу GetFileInfo().
- Потреба в них проявилась у ході рефакторингу первинної версії коду.
  - Це є проявом ***принципу повторного використання абстракцій (Reused Abstractions Principle, RAP)***, який не є частиною SOLID-принципів, але тісно з ними пов'язаний.
  - Якщо у вас є абстракції (тут – інтерфейси чи базові абстрактні класи), які не використовуються повторно (реалізуються) іншими конкретними класами, скоріше за все, у вас погані абстракції (poor abstractions).

Ще один погляд на абстракцію

“Abstraction is the elimination of the irrelevant and the amplification of the essential.”

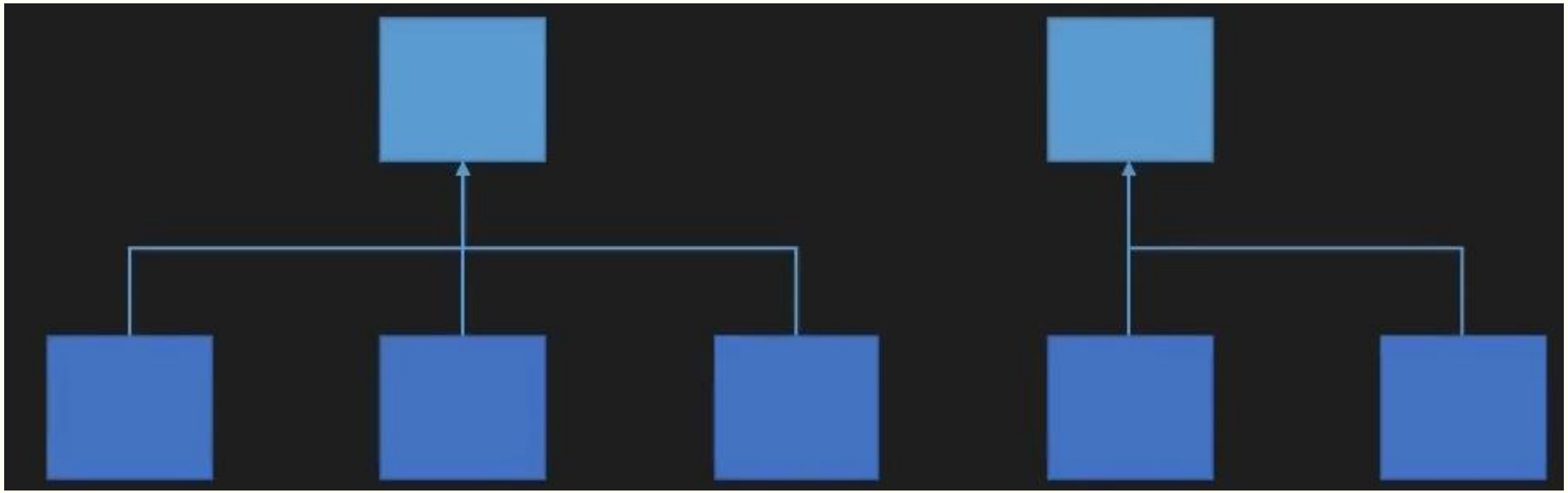
— Robert C. Martin, Agile Principles, Patterns, and Practices in C#

- Абстрагування дозволяє виконувати перегляд коду, за бажання не вникаючи в особливості конкретних реалізацій абстракцій.
  - Абстракції працюють на вищому рівні проєктування.
- Багато програмних проєктів мають подібну архітектуру коду, де згори знаходятьсяабстракції (інтерфейси чи абстрактні базові класи), а внизу – конкретні класи.
  - Повторного використання абстракцій немає.
  - Порушується RAP.



Ще один погляд на абстракцію

- Пропонується виділяти спільні характеристики та поведінку для класів.



  - Точка зору: інтерфейси/АБК не проєктуються, вони виявляються в ході розростання програмної системи.
  - Ідея: починати з конкретних реалізацій, а потім виявляти спільні абстракції.
  - ***«Правило трьох»:*** не виділяти абстракцій, поки не буде знайдено принаймні три появи аналогічного коду в різних частинах програмної системи.
  - Чим більше прикладів спільної реалізації, тим чіткіше видно потрібну для введення абстракцію.

Принцип відкритості-закритості (OCP)

- Ідея: якщо клас пішов у продакшн, клієнти цього класу залежать від нього.
  - Вам більше не дозволено вносити зміни в цей клас.
  - Для заміни поведінки класу краще описувати його відкритим для розширення: будь-хто зможе успадкувати цей клас та переозначити поведінку.
  - Єдиний виняток щодо можливості внесення змін у клас – знаходження багів у ньому.
- Принцип відкритості-закритості спочатку розглядався в контексті наслідування.
  - Пізніше дійшли висновку, що наслідування – не кращий спосіб моделювання предметної області.
  - Слід віддавати перевагу композиції.
  - Один зі способів – використання шаблону проєктування Стратегія (Strategy).
  - Альтернативні застосування: шаблон проєктування Компонувальник (Composite) або шаблон проєктування Декоратор (Decorator).
-

Принцип відкритості-закритості (OCP)

- Візьмемо конкретний клас FileStore.
  - Для відкриття класу до змін (розширення) у C# потрібно вказати, що всі методи будуть віртуальними.
  - Це підхід, що базується на наслідуванні.
  - Аналогічні дії виконаємо з методами AddOrUpdate() і GetOrAdd() з класу StoreCache та з методами класу StoreLogger.

```
public class FileStore
{
    public virtual void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }

    public virtual string ReadAllText(string path)
    {
        return File.ReadAllText(path);
    }

    public virtual FileInfo GetFileInfo(int id, string workingDirectory)
    {
        return new FileInfo(
            Path.Combine(workingDirectory, id + ".txt"));
    }
}
```

Принцип відкритості-закритості (OCP)

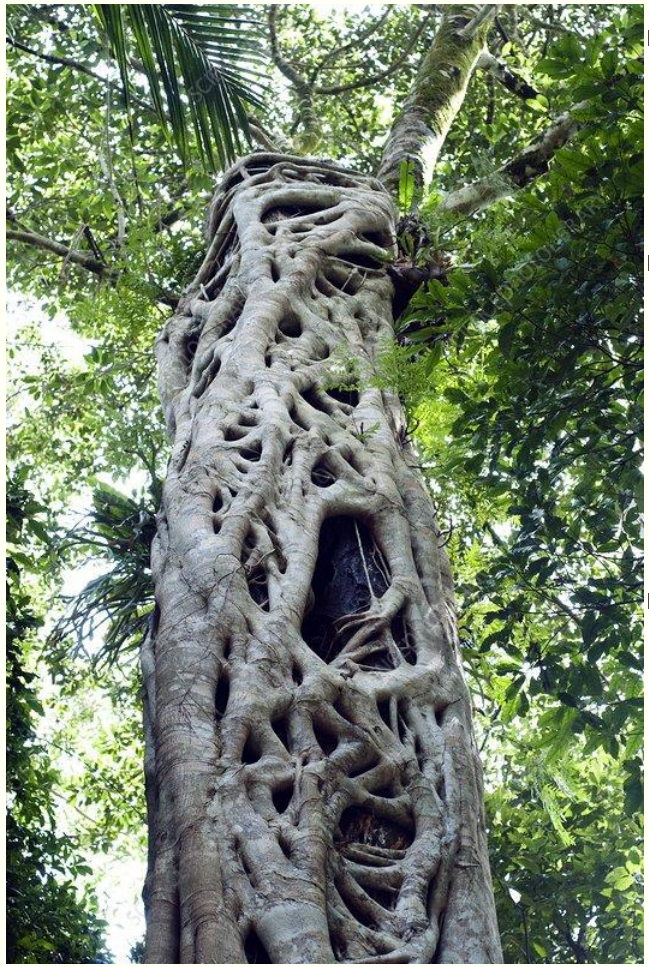
- Припустимо, що для логування буде застосовуватись інша .NET-бібліотека.
  - Наприклад, створимо відповідний клас LogForNetStoreLoggerClass, успадкований від StoreLogger.
  - Методи Save() і Read() за допомогою поліморфізму допустять роботу з об'єктом типу LogForNetStoreLoggerClass, проте проблема з'явиться в конструкторі класу MessageStore:

```
public MessageStore(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();
    this.fileStore = new FileStore();
}
```

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

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

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

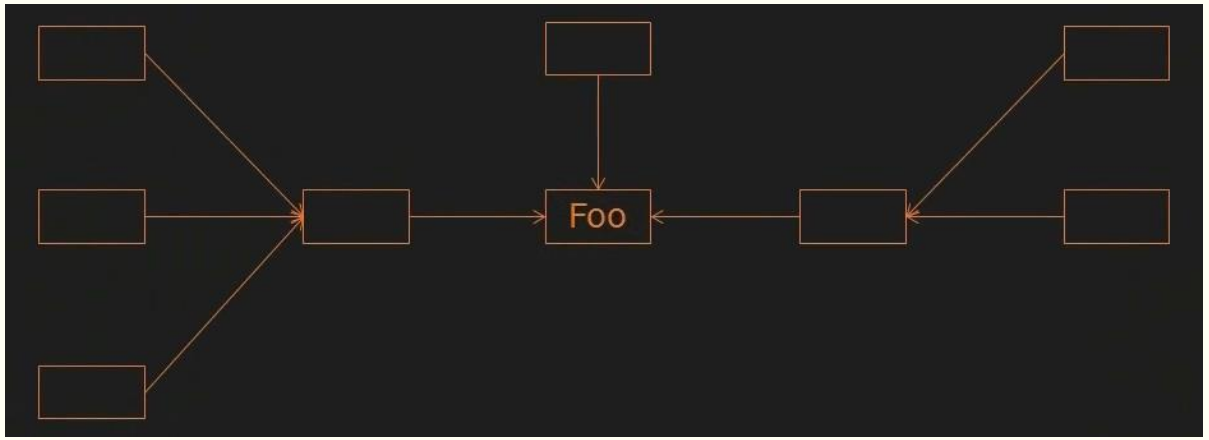
  - Змінити це можна за допомогою шаблону проєктування Фабричний метод (Factory Method).
  - Він дозволяє створити екземпляр поліморфного класу за допомогою введення в клас фабричних методів (тут це будуть властивості-геттери Store, Cache, StoreLogger).
  - Проте весь даний підхід побудований на наслідуванні і не є найкращим підходом для проєктування.

Принцип відкритості-закритості (OCP)

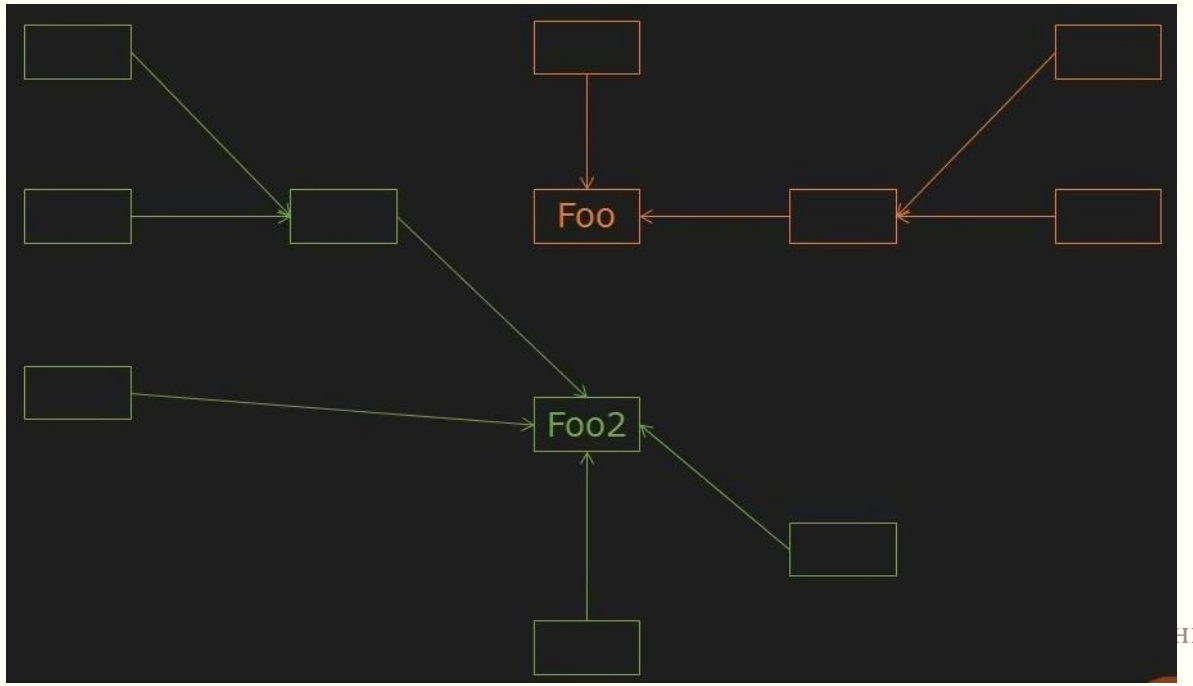


- Серед CRUD-операцій принцип відкритості-закритості каже, що класи можна лише створювати та читати їх первинний код.
  - Закритість до змін (незмінюваність) надає нові можливості.
- Як здійснювати еволюцію кодої бази?
  - Застосовується архітектурний шаблон Душитель (Strangler).
  - Ідея роботи запозичена у фікусів-душителів: навіть після смерті дерев-господарів фікуси зберігають їх форму та стають деревоподібними.
  - Таким чином, фікус є новим організмом, що має форму іншого організму.
- У проєктуванні ПЗ пропонується надбудова нової поведінки в новому класі, створеному навколо вже існуючого класу.
  - Клієнти існуючого класу поступово будуть переводити взаємодію на новий клас до повного переходу.

Принцип відкритості-закритості (OCP)



- Схема роботи:
  - Ви пишете клас і запускаєте його в продакшн
  - Інші клієнти починають використовувати цей клас
  - Ще інші клієнти працюють з існуючими клієнтами і т. д.



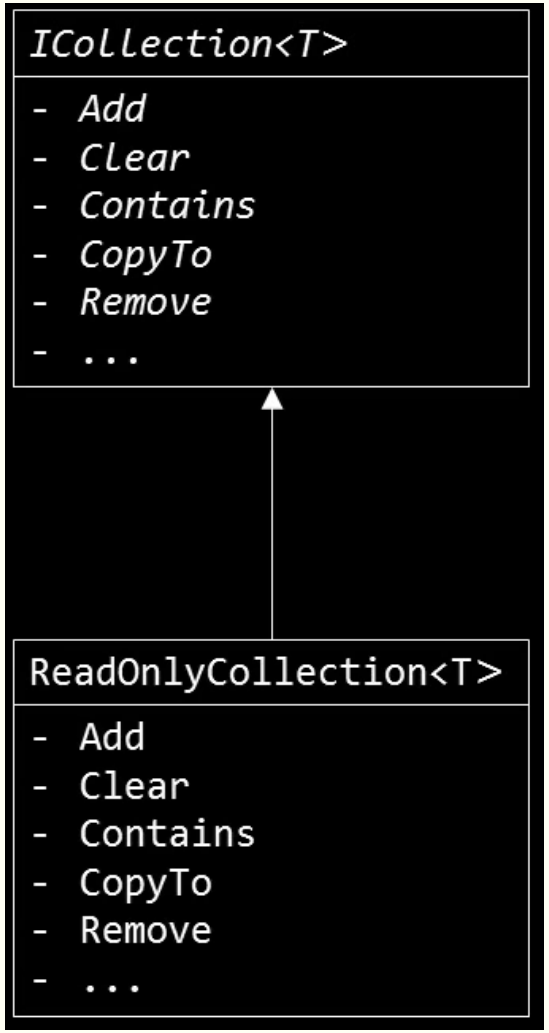
- Зміна класу спричинить хвильовий ефект.
  - Альтернатива: створити новий клас, який дублюватиме потрібну поведінку старого класу та вводитиме нову.
  - Клієнтам старого класу рекомендується поступово переходити на новішу версію.
  - Наприклад, можна застосувати атрибут Obsolete.
  - Буде виводитись попередження, проте деякий час робота зі старим класом залишатиметься такою ж.

```
[Obsolete("Please use Foo2 instead.", true)]
public class Foo
{
    // ...
}
```

Принцип підстановки Барбари Лісков (LSP)

- Описаний математично в 1980-х рр. Барбарою Лісков.
  - Роберт Мартін сформулював принцип так: підтипи повинні бути замінними на свої базові типи.
  - Більш зрозуміла трактовка: клієнт працює з різними реалізаціями абстракції без порушення *коректності* роботи системи.
- Коректність системи стосується конкретного додатку та описує множину коректних станів системи.
  - Поліморфізм (різна поведінка) може вносити варіації в поведінку системи в цілому, а коректність обмежує набір підтримуваних станів системи.
  - Якщо робота з однією реалізацією абстракції не призводить до некоректного стану, а з іншою – призводить, відбувається порушення LSP.
  - LSP часто порушується при спробах видалення фіч.
- Ймовірні ознаки порушення принципу підстановки:
  - Викидання NotSupportedException – існують деякі непідтримувані реалізації.
  - Наявність великої кількості понижуючих зведень (downcast) типів.
  - Наявність виділених (extracted) інтерфейсів – найбільш поширений індикатор порушення.

Приклад NotSupportedException у колекціях .NET



- Для ReadOnlyCollection<T> не підтримуються операції очистки, додавання чи видалення елементів, незважаючи на реалізацію інтерфейсу ICollection<T>.
  - Це порушує LSP.
  - Існує ризик, що якщо раніше система працювала зі списками (інтерфейс IList<T>), а потім вирішили змінити реалізацію на ReadOnlyColletction<T>, система почне збоїти в деяких випадках.
  - Це призводить до появи некоректних станів (викидання винятків), яких раніше не було.
- Часто відповідність вимогам принипу повторного використання абстракцій (RAP) також вказує на відповідність LSP.
  - Якщо продемонстровано створення багатьох інтерфейсів, які реалізуються багатьма класами, Ваша система вже показує, що може коректно працювати з багатьма реалізаціями.

Рефакторинг коду для підтримки принципу підстановки Барбари Лісков (LSP)

- Уявімо, що змінюється сховище для повідомлень – на реляційну базу даних.
  - Зараз сховище описується конкретним базовим класом FileStore.

```
public class FileStore
{
    public virtual void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }

    public virtual string ReadAllText(string path)
    {
        return File.ReadAllText(path);
    }

    public virtual FileInfo GetFileInfo(int id, string workingDirectory)
    {
        return new FileInfo(
            Path.Combine(workingDirectory, id + ".txt"));
    }
}
```

Рефакторинг коду для підтримки принципу підстановки Барбари Лісков (LSP)

- Створимо новий клас SqlStore, породжений від FileStore.
  - Тут спеціально залишено третій метод з викликом методу з базового класу, оскільки він, врешті решт, порушить LSP.
  - Ієрархія наслідування теж дивна.

```
public class SqlStore : FileStore
{
    public override void WriteAllText(string path, string message)
    {
        // Write to database here
    }

    public override string ReadAllText(string path)
    {
        // Read and return from database here
    }

    public override FileInfo GetFileInfo(int id, string workingDirectory)
    {
        return base.GetFileInfo(id, workingDirectory);
    }
}
```

Рефакторинг коду для підтримки принципу підстановки Барбари Лісков (LSP)

```
public interface IStore
{
    void WriteAllText(string path, string message);

    string ReadAllText(string path);

    FileInfo GetFileInfo(int id, string workingDirectory);
}
```

- Спробуємо виділити спільний інтерфейс – стандартна практика, що порушить LSP.
  - Для перекомпіляції потрібно замінити в класі MessageStore оголошення FileStore-об'єкта на IStore-об'єкт.
  - У конструкторі класу залишаємо виклик конструктора FileStore, оскільки FileStore реалізує IStore.

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

    public MessageStore(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();
        this.store = new FileStore();
    }
```

```
public class SqlStore : IStore
{
    public void WriteAllText(string path, string message)
    {
        // Write to database here
    }

    public string ReadAllText(string path)
    {
        // Read and return from database here
    }

    public FileInfo GetFileInfo(int id, string workingDirectory)
    {
        throw new NotSupportedException();
    }
}
```

Рефакторинг коду для підтримки принципу підстановки Барбари Лісков (LSP)

```
public void Save(int id, string message)
{
    this.log.Saving(id);
    var file = this.GetFileInfo(id);
    this.fileStore.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, _ => this.fileStore.ReadAllText(file.FullName));
    this.log.Returning(id);
    return new Maybe<string>(message);
}
```

- Реалізація класу SqlStore проблематична ще й тому, що в методах Save() і Read() з класу MessageStore немає обробки винятків при виклику методу GetFileInfo.
  - Це суттєво змінює коректність системи, оскільки методи Save() і Read() теж, врешті решт, викидатимуть NotSupportedException.
  - Тому це не буде рішенням.
- Можливо, створити фейковий FileInfo об’єкт?
  - Він не буде пов’язаний з БД, яка зараз є основним сховищем даних, і завжди виконуватиметься if-умова з методу Read().
  - Навіть якщо файл буде реальним, запису в БД відповідно до його вмісту немає.
  - Таким чином межі коректності суттєво зміняться, проте не буде викидатись NotSupportedException.
- Загалом, чим менше членів у виділеному інтерфейсі, тип менша ймовірність порушення LSP.
  - Очевидна відповідь – видалення методу GetFileInfo() з інтерфейсу, проте оскільки цей код уже продакшені, це буде не так просто.

Рефакторинг коду для підтримки принципу підстановки Барбари Лісков (LSP)

```
public class FileStore : IStore
{
    private readonly DirectoryInfo workingDirectory;

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

        this.workingDirectory = workingDirectory;
    }

    public virtual void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }
    public virtual string ReadAllText(string path)
    {
        return File.ReadAllText(path);
    }

    public virtual FileInfo GetFileInfo(int id, string workingDirectory)
    {
        return new FileInfo(
            Path.Combine(this.workingDirectory.FullName, id + ".txt"));
    }
```

- Ще одна проблема неуважного рефакторингу: поле workingDirectory у класі FileStore.
  - Введемо відповідне поле в клас разом із конструктором.
  - Тоді методу GetFileInfo() не потрібно передавати другий параметр, що порушує вимогу інтерфейсу IStore.
  - Тому змінимо інтерфейс IStore. Це можливо, оскільки FileStore – єдина коректна реалізація, з якої інтерфейс і було видобуто.

```
public interface IStore
{
    void WriteAllText(string path, string message);

    string ReadAllText(string path);

    FileInfo GetFileInfo(int id);
}
```

Рефакторинг коду для підтримки принципу підстановки Барбари Лісков (LSP)

- Ще одна проблема: методи інтерфейсу працюють на різних рівнях абстракції.
  - GetFileInfo() приймає на вхід ідентифікатор, проте решта методів з цим ідентифікатором не працює.
  - Id – більш абстрактне поняття, ніж шлях до конкретного файлу – path.
- Перепишемо інтерфейс IStore і клас FileStore з використанням id.
  - Це безпечно, оскільки в класі FileStore є конкретна реалізація, яка не обов'язково стосується інтерфейсу

```
public class FileStore : IStore
{
    public FileStore(DirectoryInfo workingDirectory)

    public virtual void WriteAllText(int id, string message)
    {
        var path = this.GetFileInfo(id).FullName;
        File.WriteAllText(path, message);
    }

    public virtual string ReadAllText(string path)

    public virtual FileInfo GetFileInfo(int id)
    {
        return new FileInfo(
            Path.Combine(this.workingDirectory.FullName, id + ".txt"));
    }
}
```

```
public interface IStore
{
    void WriteAllText(int id, string message);

    string ReadAllText(string path);

    FileInfo GetFileInfo(int id);
}
```

Рефакторинг коду для підтримки принципу підстановки Барбари Лісков (LSP)

```
public class MessageStore
{
    public MessageStore(DirectoryInfo workingDirectory)

    public DirectoryInfo WorkingDirectory { get; }

    public void Save(int id, string message)
    {
        this.Log.Saving(id);
        var file = this.GetFileInfo(id);
        this.Store.WriteAllText(id, message);
        this.Cache.AddOrUpdate(id, message);
        this.Log.Saved(id);
    }

    public Maybe<string> Read(int id)

    public FileInfo GetFileInfo(int id)

    protected virtual IStore Store { get; }

    protected virtual StoreCache Cache { get; }

    protected virtual StoreLogger Log { get; }
}
```

- Це потребує внесення змін у MessageStore.
  - Тепер у методі Save() видно, що змінна file ніде потім не використовується, і її можна видалити разом з викликом GetFileInfo().
- Оновимо інтерфейс IStore для роботи з id методу ReadAllText().

```
public interface IStore
{
    void WriteAllText(int id, string message);

    Maybe<string> ReadAllText(int id);

    FileInfo GetFileInfo(int 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, _ => this.Store.ReadAllText(id).Single());
    this.Log.Returning(id);
    return new Maybe<string>(message);
}
```

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

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

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

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

Рефакторинг коду для підтримки принципу підстановки Барбари Лісков (LSP)

- При зверненні до кешу очікується отримання значення типу string, проте нинішня реалізація методу ReadAllText() повертає Maybe<string>.
  - Тому дописуємо .Single(), щоб повертати очікуване значення.
  - На даний момент реалізація безпечна, проте лише в контексті FileStore-повідомлень, оскільки в основі лежить об'єкт-файл, отриманий від методу GetFileInfo().
- У класі StorageCache метод GetOrAdd() передбачає отримання повідомлення, пов'язаного з id, або виклик messageFactory(), щоб створити повідомлення та додати в словник.

Рефакторинг коду для підтримки принципу підстановки Барбари Лісков (LSP)

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

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

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

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

- Виділимо новий інтерфейс IStoreCache, який теж буде не в фінальній своїй версії

```
public interface IStoreCache
{
    void AddOrUpdate(int id, string message);

    string GetOrAdd(int id, Func<int, string> messageFactory);
}
```

- Щоб не викликати метод Single(), можемо задати для методу GetOrAdd() другий параметр типу Func<int, Maybe<string>>.
  - Проте таким чином лише загнали проблему всередину класу StoreCache.
  - Крім того, тепер немає безпечного контексту для виклику Single().

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

Рефакторинг коду для підтримки принципу підстановки Барбари Лісков (LSP)

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

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

    public virtual void AddOrUpdate(int id, string message)
    {
        var m = new Maybe<string>(message);
        this.cache.AddOrUpdate(id, m, (i, s) => m);
    }

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

- На даному етапі немає потреби прив'язуватись до типу значень у словнику.
  - Можна вказати, щоб GetOrAdd() повертав значення типу Maybe<string>, які й будуть зберігатись у словнику.
  - Тепер виклик Single() не потрібний, проте GetFileInfo() все ще викликається в методі Read() класу Maybe (потрібно виконувати різні речі залежно від існування файлу).

```
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, _ => this.Store.ReadAllText(id));
    this.Log.Returning(id);
    return message;
}
```

Рефакторинг коду для підтримки принципу підстановки Барбари Лісков (LSP)

- Насправді, нас більше цікавить, чи існує повідомлення, а не файл.
  - Тому цей код не належить даній абстракції, і його слід перенести в клас FileStore.

```
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, _ => this.Store.ReadAllText(id));
    this.Log.Returning(id);
    return message;
}
```

```
public class FileStore : IStore
{
    public FileStore(DirectoryInfo workingDirectory)

    public virtual void WriteAllText(int id, string message)

    public virtual Maybe<string> ReadAllText(int id)
    {
        var file = this.GetFileInfo(id);
        if (!file.Exists)
            return new Maybe<string>();
        var path = file.FullName;
        return new Maybe<string>(File.ReadAllText(path));
    }

    public virtual FileInfo GetFileInfo(int id)
    {
        return new FileInfo(
            Path.Combine(this.workingDirectory.FullName, id + ".txt"));
    }
}
```

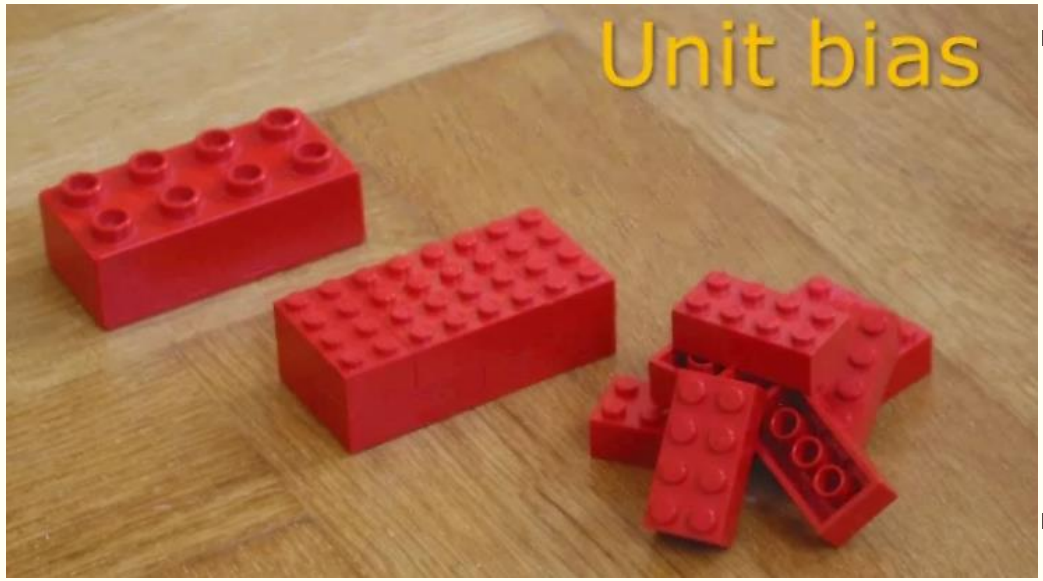
Рефакторинг коду для підтримки принципу підстановки Барбари Лісков (LSP)

```
public void Save(int id, string message)
{
    this.Log.Saving(id);
    this.Store.WriteAllText(id, message);
    this.Cache.AddOrUpdate(id, message);
    this.Log.Saved(id);
}

public Maybe<string> Read(int id)
{
    this.Log.Reading(id);
    var message = this.Cache.GetOrAdd(
        id, _ => this.Store.ReadAllText(id));
    if (message.Any())
        this.Log.Returning(id);
    else
        this.Log.DidNotFind(id);
    return message;
}

public FileInfo GetFileInfo(int id)
{
    return this.Store.GetFileInfo(id);
}
```

- Частина коду зі споживанням повідомлення тепер виглядатиме так.
  - Методи Save() і Read() більше не залежать від методу GetFileInfo(), а метод GetFileInfo() залежить від this.Store.GetFileInfo(), який реалізує інтерфейс IStore.
  - Таким чином коректність роботи програмної системи не змінюється.

```
public class SqlStore : IStore
{
    public void WriteAllText(int id, string message)
    {
        // Write to database here
    }

    public Maybe<string> ReadAllText(int id)
    {
        // Read and return from database here
    }

    public FileInfo GetFileInfo(int id)
    {
        throw new NotSupportedException();
    }
}
```

Принцип відокремлення інтерфейсу (ISP)



- Застосовуючи принципи єдиної відповідальності та відкритості-закритості, у проєкті накопичуються сотні класів.
  - Це може злякати програмістів, які тільки-но прийшли на проєкт, через когнітивне спотворення – ухил убік цілого (Unit bias) – схильність віддавати перевагу цілісним об'єктам та уникати незавершеності.
  - Більше маленьких класів дає вищий рівень гранулярності.
- Принцип відокремлення інтерфейсу допомагає усувати недоліки коду, залишені після принципу підстановки Барбари Лісков.
  - Основна ідея: клієнти не повинні залежати від методів, які вони не використовують.
  - Цікаве питання: хто «володіє» інтерфейсами?
  - Насправді, інтерфейси потрібні *клієнту,* щоб визначити коло підтримуваних дій та можливостей.

Принцип відокремлення інтерфейсу (ISP)

- На інтерфейси слід дивитись з точки зору клієнта, який їх споживає.
  - Якщо клієнт визначає інтерфейс, він повинен визначати лише те, що йому потрібно.
  - Тому говорять про віддавання переваги рольовим інтерфейсам над заголовковими інтерфейсами: рольовий інтерфейс визначатиме тільки ті методи, які потрібні клієнту.
  - В екстремальній формі рольовий інтерфейс визначає лише 1 метод, причому такі інтерфейси дозволяють простіше звести код до відповідності LSP.
  - У прикладі з інтерфейсом IStore, що складається всього з трьох методів, уже спостерігались проблеми з підтримкою LSP.
  - Якщо інтерфейс визначає єдиний метод, буде відсутня взаємодія між методами інтерфейсу, що убезпечує від великої категорії проблем, пов'язаних з LSP.
- Способів додавання нової функціональності в коді багато: наслідування, асоціація, інтерфейси, узагальнені параметри тощо.
  - Проте способів видалення функціональності небагато.
  - ISP допомагає видаляти функціональність у тому плані, що він передбачає їх початкове визначення з високим рівнем гранулярності: якщо функціональність не потрібна, не реалізуйте цей інтерфейс!

Принцип відокремлення інтерфейсу (ISP)

- Застосуємо ISP до попереднього коду класу SqlStore.
  - Виділяючи IStore з конкретного класу, ми створили заголовковий інтерфейс.
  - Зараз клас Store – єдине місце, де застосовується метод GetFileInfo().

```
public class SqlStore : IStore
{
    public void WriteAllText(int id, string message)
    {
        // Write to database here
    }

    public Maybe<string> ReadAllText(int id)
    {
        // Read and return from database here
    }

    public FileInfo GetFileInfo(int id)
    {
        throw new NotSupportedException();
    }
}
```

```
public void Save(int id, string message)
{
    this.Log.Saving(id);
    this.Store.WriteAllText(id, message);
    this.Cache.AddOrUpdate(id, message);
    this.Log.Saved(id);
}

public Maybe<string> Read(int id)
{
    this.Log.Reading(id);
    var message = this.Cache.GetOrAdd(
        id, _ => this.Store.ReadAllText(id));
    if (message.Any())
        this.Log.Returning(id);
    else
        this.Log.DidNotFind(id);
    return message;
}

public FileInfo GetFileInfo(int id)
{
    return this.Store.GetFileInfo(id);
}
```

Принцип відокремлення інтерфейсу (ISP)

- Замість виділення заголовкового інтерфейсу дамо клієнту формувати інтерфейс.
  - Визначимо інтерфейс IFileLocator з єдиним методом:

```
public interface IFileLocator
{
    FileInfo GetFileInfo(int id);
}
```

- Кожний з методів класу MessageStore використовує різні методи з інтерфейсу IStore:
  - Це порушення ISP: методи реалізують інтерфейс IStore, проте також залежать від методів, які не використовують.
  - Методу GetFileInfo() тепер немає потреби реалізувати інтерфейс IStore.
  - Замінимо на нову реалізацію:

```
public FileInfo GetFileInfo(int id)
{
    return this.FileLocator.GetFileInfo(id);
}
```

  - Тепер метод GetFileInfo() в інтерфейсі IStore взагалі непотрібний!

```
public void Save(int id, string message)
{
    this.Log.Saving(id);
    this.Store.WriteAllText(id, message);
    this.Cache.AddOrUpdate(id, message);
    this.Log.Saved(id);
}

public Maybe<string> Read(int id)
{
    this.Log.Reading(id);
    var message = this.Cache.GetOrAdd(
        id, _ => this.Store.ReadAllText(id));
    if (message.Any())
        this.Log.Returning(id);
    else
        this.Log.DidNotFind(id);
    return message;
}

public FileInfo GetFileInfo(int id)
{
    return this.Store.GetFileInfo(id);
}
```

Принцип відокремлення інтерфейсу (ISP)

```
public interface IStore
{
    void WriteAllText(int id, string message);

    Maybe<string> ReadAllText(int id);
}
```

```
public class SqlStore : IStore
{
    public void WriteAllText(int id, string message)
    {
        // Write to database here
    }

    public Maybe<string> ReadAllText(int id)
    {
        // Read and return from database here
    }
}
```

- Тепер LSP не порушується.
  - Можна застосувати ISP для подальшого введення рольових інтерфейсів.

```
public void Save(int id, string message)
{
    this.Log.Saving(id);
    this.Store.WriteAllText(id, message);
    this.Cache.AddOrUpdate(id, message);
    this.Log.Saved(id);
}

public Maybe<string> Read(int id)
{
    this.Log.Reading(id);
    var message = this.Cache.GetOrAdd(
        id, _ => this.Store.ReadAllText(id));
    if (message.Any())
        this.Log.Returning(id);
    else
        this.Log.DidNotFind(id);
    return message;
}

public FileInfo GetFileInfo(int id)
{
    return this.FileLocator.GetFileInfo(id);
}
```

Принцип відокремлення інтерфейсу (ISP)

- Подивимось на метод Save() як на клієнта:
  - Всі 4 методи всередині приймають параметр id.
  - Усі 4 методи є командами.
  - Два методи приймають параметр message.

```
public void Save(int id, string message)
{
    this.Log.Saving(id);
    this.Store.WriteAllText(id, message);
    this.Cache.AddOrUpdate(id, message);
    this.Log.Saved(id);
}
```

- Поки що інтерфейс IStoreLogger – заголовковий.
  - Додавши параметр message у методи Saving() та Saved(), ми нічого не порушимо:

```
public interface IStoreLogger
{
    void Saving(int id);

    void Saved(int id);

    void Reading(int id);

    void DidNotFind(int id);

    void Returning(int id);
}
```

```
public interface IStoreLogger
{
    void Saving(int id, string message);

    void Saved(int id, string message);

    void Reading(int id);

    void DidNotFind(int id);

    void Returning(int id);
}
```

Принцип відокремлення інтерфейсу (ISP)

- Тепер усі методи приймають однакові параметри, проте мають різні назви.
  - При пошуку спільності для виокремлення абстракцій назви є шумом.
  - Назви методів при виділенні інтерфейсу в свій час відображали особливості логіки роботи цього класу.
  - Технічно, тепер можна задавати довільну назву, наприклад, Save():

```
public void Save(int id, string message)
{
    this.Log.Saving(id, message);
    this.Store.WriteAllText(id, message);
    this.Cache.AddOrUpdate(id, message);
    this.Log.Saved(id, message);
}
```

```
public interface IStore
{
    void WriteAllText(int id, string message);

    Maybe<string> ReadAllText(int id);
}
```

```
public interface IStore
{
    void Save(int id, string message);

    Maybe<string> ReadAllText(int id);
}
```

```
public interface IStoreCache
{
    void AddOrUpdate(int id, string message);

    Maybe<string> GetOrAdd(int id, Func<int, Maybe<string>> messageFacto
}
```

```
public interface IStoreCache
{
    void Save(int id, string message);

    Maybe<string> GetOrAdd(int id, Func<int, Maybe<string>> messageFacto
}
```

Принцип відокремлення інтерфейсу (ISP)

```
public void Save(int id, string message)
{
    this.Log.Saving(id, message);
    this.Store.Save(id, message);
    this.Cache.Save(id, message);
     this.Log.Saved(id, message);
}
```

- Перейменувати однаково перший і четвертий метод ми не можемо, оскільки вони знаходяться в одному інтерфейсі.
  - Наближаємось до спільної абстракції: видно подібність сигнатур методів навіть з сигнатурою основного метода Save()!

- Виділимо рольовий інтерфейс IStoreWriter:
  - Проміжним етапом також стане виокремлення невеличких класів, що реалізуватиме інтерфейс:

```
public interface IStoreWriter
{
    void Save(int id, string message);
}
```

```
public class LogSavingStoreWriter : IStoreWriter
{
    public void Save(int id, string message)
    {
        Log.Information("Saving message {id}.", id);
    }
}
```

```
public class LogSavedStoreWriter : IStoreWriter
{
    public void Save(int id, string message)
    {
        Log.Information("Saved message {id}.", id);
    }
}
```

Принцип відокремлення інтерфейсу (ISP)

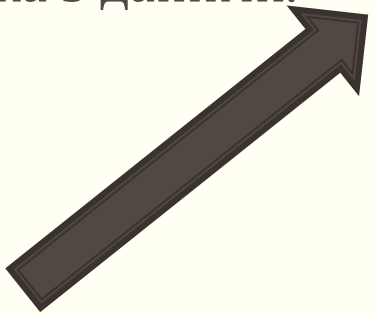
```
public void Save(int id, string message)
{
    new LogSavingStoreWriter().Save(id, message);
                    this.Store.Save(id, message);
                    this.Cache.Save(id, message);
     new LogSavedStoreWriter().Save(id, message);
}
```

- Основна проблема: для використання цих класів потрібно конструювати об’єкти.
  - Проте чудово видно подібність викликів всередині методу Save().
  - У той же час, порушується принцип відкритості-закритості – тепер не можна змінювати реалізацію LogSavingStoreWriter та LogSavedStoreWriter.
  - Вирішення даної проблеми за допомогою фабричних методів призведе до «вибухової» появи нових фабричних властивостей, тому знову постає питання, як замінити наслідування на композицію.

Розглянемо функціональний погляд на код

- Об’єкт – це дані з поведінкою.
- Замикання (closure) – це поведінка з даними.

```
public class FileStore : IMessageQuery
{
    private readonly DirectoryInfo workingDirectory;

    public FileStore(DirectoryInfo workingDirectory)
    {
        this.workingDirectory = workingDirectory;
    }

    public string Read(int id)
    {
        var path = Path.Combine(
            this.workingDirectory.FullName,
            id + ".txt");
        return File.ReadAllText(path);
    }
}
```



```
var workingDirectory =
    new DirectoryInfo(Environment.CurrentDirectory);

Func<int, string> read = id =>
    {
        var path = Path.Combine(
            workingDirectory.FullName,
            id + ".txt");
        return File.ReadAllText(path);
    };
```

```
[CompilerGenerated]
private sealed class <>c__DisplayClass3
{
    public DirectoryInfo workingDirectory;

    public string <UseClosure>b__2(int id)
    {
        return File.ReadAllText(
            Path.Combine(
                this.workingDirectory.FullName,
                id + ".txt"));
    }
}
```

Принцип інверсії залежностей (DIP)

- Говорить: високорівневі модулі мають не залежати від низьорівневих модулів.
  - Обидва мають залежати від абстракцій!
  - Абстракції не мають залежати від деталей – деталі залежать від абстракцій.
- Знову про переваги композиції над наслідуванням.
  - Одиничне наслідування накладає дуже строгі обмеження на успадкований клас: він не може наслідуватись від будь-якого іншого класу.
  - Замість такої негнучкої поведінки можна використовувати інтерфейси та композицію, причому без втрати функціональності.
  - Множинне наслідування знімає проблему обмеженої гнучкості, проте багато об’єктно-орієнтованих мов його не підтримують.

Принцип інверсії залежностей (DIP)

- Застосуємо шаблон проєктування Компонувальник (Composite) для вирішення проблеми порушення принципу відкритості-закритості в коді:

```
public void Save(int id, string message)
{
    new LogSavingStoreWriter().Save(id, message);
    this.Store.Save(id, message);
    this.Cache.Save(id, message);
    new LogSavedStoreWriter().Save(id, message);
}
```

- Компонувальник є специфічною реалізацією інтерфейсу.

```
public interface IStoreWriter
{
    void Save(int id, string message);
}
```

  - Тут клас реалізує інтерфейс IStoreWriter, проте також містить довільну кількість IStoreWriter-реалізацій.
  - Способів впровадження Компонувальника багато.
  - Тут обрано передачу params-масиву в конструктор класу.

```
public class CompositeStoreWriter : IStoreWriter
{
    private readonly IStoreWriter[] writers;

    public CompositeStoreWriter(params IStoreWriter[] writers)
    {
        this.writers = writers;
    }

    public void Save(int id, string message)
    {
        foreach (var w in this.writers)
            w.Save(id, message);
    }
}
```

Принцип інверсії залежностей (DIP)

```
public void Save(int id, string message)
{
    new LogSavingStoreWriter().Save(id, message);
    this.Store.Save(id, message);
    this.Cache.Save(id, message);
    new LogSavedStoreWriter().Save(id, message);
}
```

```
public void Save(int id, string message)
{
    this.Writer.Save(id, message);
}
```

- Властивість this.Writer буде введена в конструкторі класу MessageStore.
  - Оскільки CompositeStoreWriter реалізує IStoreWriter, можливо створити екземпляр CompositeStoreWriter та присвоїти його полю типу IStoreWriter.
  - Усі передані аргументи теж реалізують IStoreWriter.
  - Таким чином, ми відокремили (decoupled) ядро методу від способу компонування графа об'єктів.
  - При потребі тепер можна наслідуватись від класу MessageStore та заміщати фабричну властивість Writer та переозначити компонування store writer (іншими логерами, кешами і т. д.).

```
public MessageStore(DirectoryInfo workingDirectory)
{
    this.WorkingDirectory = workingDirectory;
    var c = new StoreCache();
    this.cache = c;
    this.log = new StoreLogger();
    var fileStore = new FileStore(workingDirectory);
    this.store = fileStore;
    this.fileLocator = fileStore;
    this.writer = new CompositeStoreWriter(
        new LogSavingStoreWriter(),
        fileStore,
        c,
        new LogSavedStoreWriter());
}
```

Принцип інверсії залежностей (DIP)

```
public class StoreCache : IStoreCache, IStoreWriter
{
    private readonly ConcurrentDictionary<int, Maybe<string>> cache;

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

    public virtual void Save(int id, string message)
    {
        var m = new Maybe<string>(message);
        this.cache.AddOrUpdate(id, m, (i, s) => m);
    }

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

- Компонувальник добре працює з методами-командами, оскільки можна компонувати довільну кількість команд та не переживати за вихідні типи.
  - З методами-запитами реалізація Компонувальника ускладнюється, проте інший шаблон – Декоратор – значно простіше реалізувати для них.
  - Декоратор доречний і для методів-команд.
- Для перетворення storage-класу в клас-декоратор, StorageCache повинен не тільки реалізовувати інтерфейс IStoreWriter, а й компонуватись з ним.
  - Спосіб компонування – додавання IStoreWriter-об’єкта через конструктор.

Декорування методу Save()

```
public class StoreCache : IStoreCache, IStoreWriter
{
    private readonly ConcurrentDictionary<int, Maybe<string>> cache;
    private readonly IStoreWriter writer;

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

    public virtual void Save(int id, string message)
    {
        this.writer.Save(id, message);
        var m = new Maybe<string>(message);
        this.cache.AddOrUpdate(id, m, (i, s) => m);
    }

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

- У методі Save() тепер викличемо відповідний метод внутрішнього writer-об’єкт.
  - Після цього можна кешувати повідомлення.
  - Отримується аналогічна наслідуванню функціональність, проте більш гнучка.
  - Клас StoreCache став декоратором для IStoreWriter.
- Реалізація методу Save() залишається тією ж, проте композиція коду змінилась.

Декорування методу Save()

```
public MessageStore(DirectoryInfo workingDirectory)
{
    this.WorkingDirectory = workingDirectory;
    var fileStore = new FileStore(workingDirectory);
    var c = new StoreCache(fileStore);
    this.cache = c;
    this.log = new StoreLogger();
    this.store = fileStore;
    this.fileLocator = fileStore;
    this.writer = new CompositeStoreWriter(
        new LogSavingStoreWriter(),
        c,
        new LogSavedStoreWriter());
}

public DirectoryInfo WorkingDirectory { get; private set; }

public void Save(int id, string message)
{
    this.Writer.Save(id, message);
}
```

- Реалізація методу Save() залишається тією ж, проте композиція коду змінилась.
  - Змінна c тепер має тип StoreCache без FileStore, оскільки останній тип теж реалізує інтерфейс IStoreWriter.
- Подібні операції можна застосувати і до класу StoreLogger.
  - Проте реалізувати IStoreWriter у цьому класі не вдасться, оскільки ми не зможемо обрати між методами Saving() та Saved().

Принцип інверсії залежностей (DIP)

```
public class StoreLogger : IStoreLogger, IStoreWriter
{
    private readonly IStoreWriter writer;

    public StoreLogger(IStoreWriter writer)
    {
        this.writer = writer;
    }

    public void Save(int id, string message)
    {
        Log.Information("Saving message {id}.", id);
        this.writer.Save(id, message);
        Log.Information("Saved message {id}.", id);
    }

    public virtual void Saving(int id, string message)
    {
        Log.Information("Saving message {id}.", id);
    }

    public virtual void Saved(int id, string message)
    {
        Log.Information("Saved message {id}.", id);
    }
```

```
public MessageStore(DirectoryInfo workingDirectory)
{
    this.WorkingDirectory = workingDirectory;
    var fileStore = new FileStore(workingDirectory);
    var c = new StoreCache(fileStore);
    this.cache = c;
    var l = new StoreLogger(c);
    this.log = l;
    this.store = fileStore;
    this.fileLocator = fileStore;
    this.writer = new CompositeStoreWriter(l);
}

public DirectoryInfo WorkingDirectory { get; private set; }

public void Save(int id, string message)
{
    this.Writer.Save(id, message);
}

public Maybe<string> Read(int id)
{
    this.Log.Reading(id);
```

- Клас StoreLogger є декоратором IStoreWriter і декорує змінну c
  - Конструювання writer за допомогою CompositeStoreWriter() стає надмірним, тому можемо його прибрати.
  - Отримуємо код-матрьошку.

Рефакторинг методу Read()

```
public Maybe<string> Read(int id)
{
    this.Log.Reading(id);
    var message = this.Cache.GetOrAdd(
        id, _ => this.Store.ReadAllText(id));
    if (message.Any())
        this.Log.Returning(id);
    else
        this.Log.DidNotFind(id);
    return message;
}
```

```
public interface IStoreCache
{
    void Save(int id, string message);

    Maybe<string> GetOrAdd(int id,
        Func<int, Maybe<string>> messageFactory);
}
```

- Усі методи всередині Read() отримують в якості параметру id.
  - Деякі з цих методів – команди, а деякі – запити.
  - Найбільша проблема – метод GetOrAdd(), оскільки він також приймає ще один параметр-делегат.
- Потреба в делегаті визначалась тим, що слід додавати елементи в кеш, якщо їх там ще немає.
  - У поточній формі нам достатньо глянути результат роботи методу – значення типу Maybe<string>.
  - Якщо значення порожнє, то такого елементу немає в кеші, а інакше для поточного id повідомлення в кеші присутнє.
  - Спробуємо видалити делегат та розглянемо наслідки цього рішення.

Рефакторинг методу Read()

```
public Maybe<string> Read(int id)
{
    this.Log.Reading(id);
    var message = this.Cache.GetOrAdd(id);
    if (!message.Any())
    {
        message = this.Store.ReadAllText(id);
        if (message.Any())
            this.Cache.Save(id, message.Single());
    }
    if (message.Any())
        this.Log.Returning(id);
    else
        this.Log.DidNotFind(id);
    return message;
}
```

- Змінна message матиме тип Maybe<string>.
  - Якщо повідомлення наявне (message.Any() == true), зберігаємо його в кеш у безпечній формі (викликаючи Single()).
  - У результаті, якщо повідомлення містить текст, записуємо його в лог, інакше записуємо повідомлення, що текст не знайдено.
- Назва методу GetOrAdd() беззмістовна відносно реальної роботи методу, виконаємо перейменування:

```
public interface IStoreCache
{
    void Save(int id, string message);

    Maybe<string> Read(int id);
}
```

```
public interface IStore
{
    void Save(int id, string message);

    Maybe<string> ReadAllText(int id);
}
```

```
public interface IStore
{
    void Save(int id, string message);

    Maybe<string> Read(int id);
}
```

Рефакторинг методу Read()

```
public Maybe<string> Read(int id)
{
    this.Log.Reading(id);
    var message = this.Cache.Read(id);
    if (!message.Any())
    {
        message = this.Store.Read(id);
        if (message.Any())
            this.Cache.Save(id, message.Single());
    }
    if (message.Any())
        this.Log.Returning(id);
    else
        this.Log.DidNotFind(id);
    return message;
}
```

- Розглянувши початковий заголовковий інтерфейс IStore, виділимо з нього рольовий інтерфейс IStoreReader.

```
public interface IStore
{
    void Save(int id, string message);

    Maybe<string> Read(int id);
}
```

```
public interface IStoreReader
{
    Maybe<string> Read(int id);
}
```

- Тепер зможемо застосувати шаблон Декоратор.

Рефакторинг методу Read()

```
public class StoreCache : IStoreCache, IStoreWriter, IStoreReader
{
    private readonly ConcurrentDictionary<int, Maybe<string>> cache;
    private readonly IStoreWriter writer;
    private readonly IStoreReader reader;

    public StoreCache(IStoreWriter writer, IStoreReader reader)
    {
        this.cache = new ConcurrentDictionary<int, Maybe<string>>();
        this.writer = writer;
        this.reader = reader;
    }

    public virtual void Save(int id, string message)

    public virtual Maybe<string> Read(int id)
    {
        Maybe<string> retVal;
        if (this.cache.TryGetValue(id, out retVal))
            return retVal;
        return new Maybe<string>();
    }
}
```

- У поточній реалізації метод Read() намагається отримати значення з кешу.
  - Якщо значення є, воно повертається, інакше повертається значення типу Maybe.
  - Маючи внутрішнє значення reader, можна звернутись до нього, коли в кеші значення не знайдено:

```
public virtual Maybe<string> Read(int id)
{
    Maybe<string> retVal;
    if (this.cache.TryGetValue(id, out retVal))
        return retVal;

    retVal = this.reader.Read(id);
    if (retVal.Any())
        this.cache.AddOrUpdate(id, retVal, (i, s) => retVal);

    return retVal;
}
```

  - Комбінування writer і reader демонструє аналог множинного наслідування на основі композиції.

Рефакторинг методу Read()

- Маючи декоратор, можна спростити метод Read():

```
public Maybe<string> Read(int id)
{
    this.Log.Reading(id);
    var message = this.Cache.Read(id);
    if (message.Any())
        this.Log.Returning(id);
    else
        this.Log.DidNotFind(id);
    return message;
}
```

```
public MessageStore(DirectoryInfo workingDirectory)
{
    this.WorkingDirectory = workingDirectory;
    var fileStore = new FileStore(workingDirectory);
    var c = new StoreCache(fileStore, fileStore);
    this.cache = c;
    var l = new StoreLogger(c);
    this.log = l;
    this.store = fileStore;
    this.fileLocator = fileStore;
    this.writer = l;
}
```

- Якщо така логіка зчитування з кешу та інших джерел даних зустрічається в багатьох місцях програми, даний код стає ще кориснішим.
- Кеш тепер конструюється з двох однакових параметрів: реалізуються як IStoreReader, так і IStoreWriter, проте StoreCache не знає про це, а знає лише, що залежить від reader і writer.

Далі зробимо клас StoreLogger декоратором IStoreReader

- IStoreReader оголошує метод-запит

```
public class StoreLogger : IStoreLogger, IStoreWriter, IStoreReader
{
    private readonly IStoreWriter writer;
    private readonly IStoreReader reader;

    public StoreLogger(IStoreWriter writer, IStoreReader reader)
    {
        this.writer = writer;
        this.reader = reader;
    }

    public void Save(int id, string message)

    public Maybe<string> Read(int id)
    {
        Log.Debug("Reading message {id}.", id);
        var retVal = this.reader.Read(id);
        if (retVal.Any())
            Log.Debug("Returning message {id}.", id);
        else
            Log.Debug("No message {id} found.", id);
        return retVal;
    }
    public virtual void Saving(int id, string message)

    public virtual void Saved(int id, string message)

    public virtual void Reading(int id)

    public virtual void DidNotFind(int id)

    public virtual void Returning(int id)
}
```

```
public MessageStore(DirectoryInfo workingDirectory)
{
    this.WorkingDirectory = workingDirectory;
    var fileStore = new FileStore(workingDirectory);
    var c = new StoreCache(fileStore, fileStore);
    this.cache = c;
    var l = new StoreLogger(c, c);
    this.log = l;
    this.store = fileStore;
    this.fileLocator = fileStore;
    this.writer = l;
    this.reader = l;
}

public DirectoryInfo WorkingDirectory { get; private s

public void Save(int id, string message)
{
    this.Writer.Save(id, message);
}

public Maybe<string> Read(int id)
{
    return this.Reader.Read(id);
}
```

Фінальні кроки

```
public class MessageStore
{
    private readonly IStoreCache cache;
    private readonly IStoreLogger log;
    private readonly IStore store;
    private readonly IFileLocator fileLocator;
    private readonly IStoreWriter writer;
    private readonly IStoreReader reader;

    public MessageStore(DirectoryInfo workingDirectory)
    {
        this.WorkingDirectory = workingDirectory;
        var fileStore = new FileStore(workingDirectory);
        var c = new StoreCache(fileStore, fileStore);
        this.cache = c;
        var l = new StoreLogger(c, c);
        this.log = l;
        this.store = fileStore;
        this.fileLocator = fileStore;
        this.writer = l;
        this.reader = l;
    }

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

- Інтерфейси IStoreCache, IStoreLogger, IStore – заголовкові та більше не використовуються.
  - Також потрібно усунути 6 фабричних властивостей, які покладаються на наслідування, а не композицію.

```
public Maybe<string> Read(int id)
{
    return this.Reader.Read(id);
}

public FileInfo GetFileInfo(int id)
{
    return this.FileLocator.GetFileInfo(id);
}

protected virtual IStore Store
{
    get { return this.store; }
}

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

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

```
protected virtual IFileLocator FileLocator
{
    get { return this.fileLocator; }
}

protected virtual IStoreWriter Writer
{
    get { return this.writer; }
}

protected virtual IStoreReader Reader
{
    get { return this.reader; }
}
}
```

Чистіша версія

```
public class MessageStore
{
    private readonly IFileLocator fileLocator;
    private readonly IStoreWriter writer;
    private readonly IStoreReader reader;

    public MessageStore(
        IStoreWriter writer,
        IStoreReader reader,
        IFileLocator fileLocator)
    {
        if (writer == null)
            throw new ArgumentNullException("writer");
        if (reader == null)
            throw new ArgumentNullException("reader");
        if (fileLocator == null)
            throw new ArgumentNullException("fileLocator");

        this.fileLocator = fileLocator;
        this.writer = writer;
        this.reader = reader;
    }
```

- Змінився конструктор класу, оскільки для реалізації принципу інверсії залежностей важливий спосіб компонування інтерфейсів.
  - Тепер сторонні компоненти, які потребують MessageStore, мають постачати дані три залежності через конструктор.
  - Конструктор захищає свої інваріанти за допомогою захисних перевірок кожної з залежностей.
  - У такому вигляді клас може бути надмірним, проте це місце для бізнес-логіки, не введеної для спрощення коду.

```
    public void Save(int id, string message)
    {
        this.writer.Save(id, message);
    }

    public Maybe<string> Read(int id)
    {
        return this.reader.Read(id);
    }

    public FileInfo GetFileInfo(int id)
    {
        return this.fileLocator.GetFileInfo(id);
    }
}
```

Чистіша версія

- Потреба в наслідуванні тепер повністю відпала, оскільки реалізація тепер покладається на композицію.
  - За бажання можна навіть видалити позначки віртуальності для методів відповідних класів:

```
public class StoreCache : IStoreWriter, IStoreReader
{
    public StoreCache(IStoreWriter writer, IStoreReader reader)

    public virtual void Save(int id, string message)

    public virtual Maybe<string> Read(int id)
}
```

```
public class FileStore : IFileLocator, IStoreWriter, IStoreReader
{
    public FileStore(DirectoryInfo workingDirectory)

    public virtual void Save(int id, string message)

    public virtual Maybe<string> Read(int id)

    public virtual FileInfo GetFileInfo(int id)
}
```

Чистіша версія

```
public class StoreLogger : IStoreWriter, IStoreReader
{
    private readonly ILogger log;
    private readonly IStoreWriter writer;
    private readonly IStoreReader reader;

    public StoreLogger(
        ILogger log, IStoreWriter writer, IStoreReader reader)
    {
        this.log = log;
        this.writer = writer;
        this.reader = reader;
    }

    public void Save(int id, string message)
    {
        this.log.Information("Saving message {id}.", id);
        this.writer.Save(id, message);
        this.log.Information("Saved message {id}.", id);
    }

    public Maybe<string> Read(int id)
    {
        this.log.Debug("Reading message {id}.", id);
        var retVal = this.reader.Read(id);
```

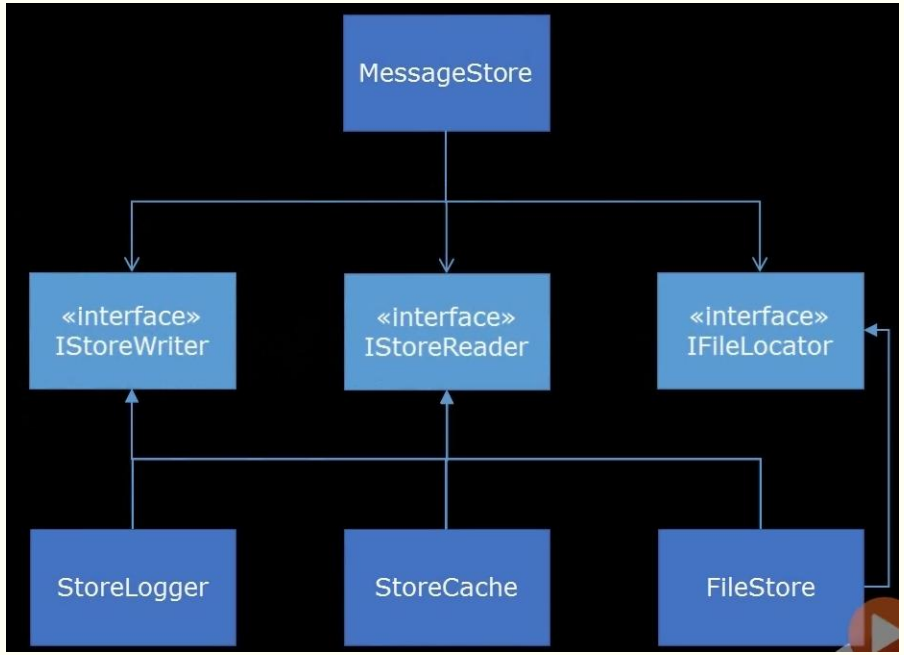
- У старішій версії для логування використовувались статичні методи Log.Debug() та Log.Information().
  - Слідуючи принципу інверсії залежностей, краще впровадити екзмепляр інтерфейсу ILogger.
  - Тоді клієнтський код у класі MessageStore можна буде компонувати так:

```
var logger = new LoggerConfiguration().CreateLogger();
var fileStore =
    new FileStore(
        new DirectoryInfo(
            Environment.CurrentDirectory));
var cache = new StoreCache(fileStore, fileStore);
var log = new StoreLogger(logger, cache, cache);
var msgStore = new MessageStore(
    log,
    log,
    fileStore);
```

  - За бажання можна перекомпонувати MessageStore, і для цього досить змінити лише цей код.

Чистіша версія

```
var logger = new LoggerConfiguration().CreateLogger();
var fileStore =
    new FileStore(
        new DirectoryInfo(
            Environment.CurrentDirectory));
var cache = new StoreCache(fileStore, fileStore);
var log = new StoreLogger(logger, cache, cache);
var msgStore = new MessageStore(
    log,
    log,
    fileStore);
```



- Наприклад, якщо не подобається поточна реалізація StoreCache (використання простого словника), і потрібно застосувати доречну реалізацію кеша, можна створити новий клас-адаптер, який реалізуватиме доречні інтерфейси.
  - Рекомендації стратегій кешування від Microsoft
- Відповідно до сформованої архітектури, всі конкретні деталі залежать від абстракцій, проте жодна абстракція не залежить від деталей реалізації.
  - Це результат застосування принципу інверсії залежностей до нашої кодової бази.

ДЯКУЮ ЗА УВАГУ!

Наступне питання: Колекції в мові програмування C#.