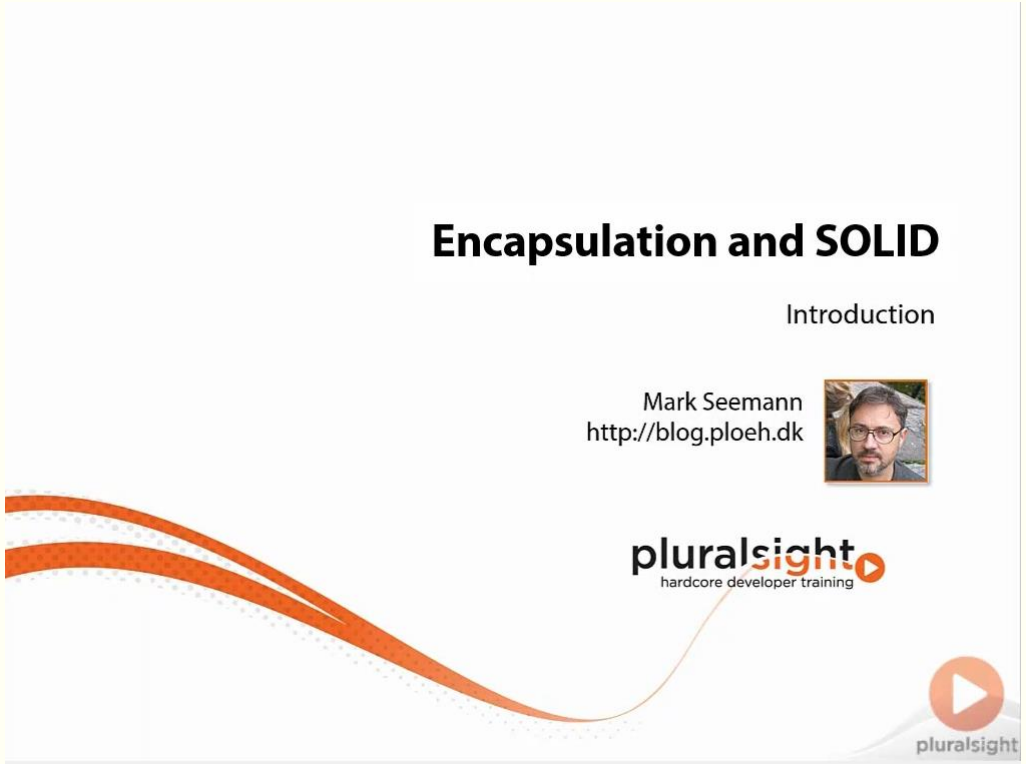
ПРИНЦИПИ ПОБУДОВИ ЯКІСНОГО ОБ'ЄКТНО-ОРІЄНТОВАНОГО КОДУ

Лекція 06
Об'єктно-орієнтоване програмування

План лекції

- Додатковий погляд на інкапсуляцію.
- Філософія об’єктно-орієнтованого проєктування програмного забезпечення.
- SOLID-принципи розробки об’єктно-орієнтованого коду.



ДОДАТКОВИЙ ПОГЛЯД НА ІНКАПСУЛЯЦІЮ

Питання 6.1.

Погляд на написання коду

- Фреймворки та бібліотеки є прикладами повторно використовуваних компонентів у програмних системах.
  - Наскільки регулярно Ви заглядаєте в їх первинний код при створенні своїх додатків?
  - Ці компоненти написані так, щоб не мати постійної потреби передивлятись їх код.
- Основне припущення – Ваш код буде передивлятись та використовувати неграмотний програміст, що не знає особливості саме Вашої реалізації.
  - Пишіть код таким чином, наче ним буде користуватись маніяк-психопат, який знає Вашу адресу.
- Розглянемо іграшковий приклад та спробуємо вгадати, за що відповідають члени класу:

```
public class FileStore
{
    public string WorkingDirectory { get; set; }

    public string Save(int id, string message)

    public event EventHandler<MessageEventArgs> MessageRead;

    public void Read(int id)
}
```

Погляд на написання коду

```
public class FileStore
{
    public string WorkingDirectory { get; set; }

    public string Save(int id, string message)

    public event EventHandler<MessageEventArgs> MessageRead;

    public void Read(int id)
}
```

Проблема коду: важко зрозуміти, що він робить, не читаючи реалізацій

- Чому метод Save() повертає значення типу string?
  - Що може бути в цьому рядку?
  - Ехо-вивід повідомлення? Рядкове представлення id? Статус-код, інкапсульований у рядок?
  - Без свіжої документації єдиний вихід – переглянути первинний код:

```
public string Save(int id, string message)
{
    var path = Path.Combine(this.WorkingDirectory, id + ".txt");
    File.WriteAllText(path, message);
    return path;
}
```

  - При об'ємній та складній реалізації доведеться розбиратись з її особливостями.
- Чому Read() повертає значення типу void?
  - Тут видно, що може бути зв'язок з подією MessageRead.

```
public void Read(int id)
{
    var path = Path.Combine(this.WorkingDirectory, id + ".txt");
    var msg = File.ReadAllText(path);
    this.MessageRead(this, new MessageEventArgs { Message = msg });
}
```

У чому код поганий?

- Мета: сповільнити деградацію кодової бази, спричиненою тим, що кілька програмістів з команди пишуть спагетті-код.
  - Падає продуктивність у тривалій перспективі, оскільки все важче буде додавати нові фічі.
  - Стає проблематичніше супроводжувати код, оскільки в ньому виникають все нові й нові баги.
- Часто програмісти більше читають код, ніж пишуть його.
  - Потрібно так писати код, щоб він був читабельний та легкий для розуміння.
  - Проблема в тому, що коли Ви реалізуєте фічу, ***Ви знаєте***, що вона повинна робити і про що Ви думали, коли реалізовували цю фічу.

Загальний погляд на інкапсуляцію

- Інкапсуляція оперує двома базовими поняттями:
  - ***Приховування інформації (information hiding)*** – усталена та не зовсім вірна назва. За своєю суттю, це ***приховування реалізації (implementation hiding)***. Наприклад, форми збереження паролів у класі.
  - ***Захист інваріантів (protection of invariants)***. *Забезпечення неможливості чи значне ускладнення* переходу об'єкта в некоректний внутрішній стан (invalid state). Ідея: додавати програмні перевірки передумов та постумов в сам клас.

- Разом з інкапсуляцією діють ще два принципи:
  - ***Відокремлення команд та запитів (Command Query Separation, CQS)*** – операція повинна бути або командою, або запитом, проте не обома.
    - Говоримо про операції, а не методи чи функції, оскільки принцип має загальне застосування, не лише в ООП.
    - ***Команда*** – це будь-яка операція, що має видимі в системі побічні ефекти: зміна біту пам'яті, збереження файлу тощо.
    - ***Запит (query)*** – це будь-яка операція, що повертає дані. Термін введено до широкого поширення реляційних БД.
  - Закон Постела (Postel's law) – визначає межі застосування принципу CQS.

Відокремлення команд та запитів (CQS)

- ***Команди змінюють (mutate) стан.*** Приклади:
  - void Save(Order order); // ймовірно, зберігає замовлення деяким чином
  - void Send(T message); // ймовірно, надсилає деяке повідомлення певним чином
  - void Associate(IFoo foo, Bar bar); // певним чином змінюється стан системи, щоб пов'язати foo і bar
  - Оскільки такі методи нічого не повертають, вони, очевидно, повинні вносити побічні ефекти.
  - Безпечно викликати запити всередині команд, проте не навпаки.

- ***Запити не змінюють стан.*** Приклади:
  - Order[] GetOrders(int userId); // ймовірно, повертає перелік замовлень користувача за його id
  - IFoo Map(Bar bar); // ймовірно, переформатовує інформацію з bar для реалізації інтерфейсу Ifoo
  - T Create(); // деяким чином створює об'єкт узагальненого типу T
  - Запити природним чином є ідемпотентними: повторне задавання питання не змінює відповіді.
  - Таким чином, запити безпечно викликати, що забезпечує базову *коректність* коду.
  - Запити не створюють побічні ефекти, тому безпечні для виклику.

Перша версія демонстраційного коду

```
public class FileStore
{
    public string WorkingDirectory { get; set; }

    public string Save(int id, string message)
    {
        var path = Path.Combine(this.WorkingDirectory, id + ".txt");
        File.WriteAllText(path, message);
        return path;
    }

    public event EventHandler<MessageEventArgs> MessageRead;

    public void Read(int id)
    {
        var path = Path.Combine(this.WorkingDirectory, id + ".txt");
        var msg = File.ReadAllText(path);
        this.MessageRead(this, new MessageEventArgs { Message = msg });
    }
}
```

- Де тут запит?
  - За логікою, це ***має бути*** метод Read().
  - Проте за поточною сигнатурою методу – виглядає як команда.
  - Це потрібно буде виправити, це один з проявів поганого коду.

```
public string Read(int id)
{
    var path = Path.Combine(this.WorkingDirectory, id + ".txt");
    var msg = File.ReadAllText(path);
    this.MessageRead(this, new MessageEventArgs { Message = msg });
    return msg;
}
```

- Чи отриманий метод-запит не має побічних ефектів?
  - Метод викликає подію – побічний ефект.
  - Обробники подій є методами-командами, тобто вносять побічні ефекти.
  - Видалимо подію цілком, нічого не втратимо, оскільки повідомлення повертається методом.

Рефакторинг коду

```
public class FileStore
{
    public string WorkingDirectory { get; set; }

    public string Save(int id, string message)
    {
        var path = Path.Combine(this.WorkingDirectory, id + ".txt");
        File.WriteAllText(path, message);
        return path;
    }

    public string Read(int id)
}
```

- Де тут команда?
  - Метод Save() за логікою має бути командою, проте виглядає і як запит (повертає значення), і як команда (утворює побічний ефект).
  - Це порушує принцип CQS.
- Виправимо порушення, прибравши повернення path із методу:

```
public void Save(int id, string message)
{
    var path = Path.Combine(this.WorkingDirectory, id + ".txt");
    File.WriteAllText(path, message);
}
```

  - Проте таким чином може втратитись важлива для викликаючої сторони інформація.
  - Додамо метод-запит GetFileName(), який не матиме побічних ефектів.
  - Виклик цього методу не змінює стан системи і є ідемпотентним. Клієнт може викликати метод при потребі.

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

Рефакторинг коду

```
public class FileStore
{
    public string WorkingDirectory { get; set; }

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

    public string Read(int id)
    {
        var path = this.GetFileName(id);
        var msg = File.ReadAllText(path);
        return msg;
    }

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

- Ініціалізуємо змінні path за допомогою методу GetFileName().
  - Також видно, що можна викликати запити всередині команд: GetFileName() всередині Save().
  - Подолання порушення принципу CQS полягало в декомпозиції суперечливої операції на команди та запити.
  - Це і є відокремлення команд та запитів.
- Строге слідування принципу в команді розробників дозволяє покладатись на роботу коду без повного його розуміння.
  - Маючи лише високорівневе бачення.
  - Це скорочує час на читання коду.

Закон Постела (принцип надійності, Robustness principle)

- Що можна зробити на рівні коду, щоб підвищити довіру до програмної системи?
  - Як довіряти тому, що команда *приймає* вхідні дані?
  - Як довіряти тому, що запит поверне *корисний* відгук (дані)?
- Закон Постела: будьте дуже консервативними в тому, що надсилаєте, та ліберальними в тому, що приймаєте.
  - Якомога строгіше гарантуйте клієнту результат (консервативна відповідь)
  - З іншого боку, якщо зрозуміло, що клієнт зі своїми вхідними даними мав на увазі, приймайте ці дані (ліберальний прийом).
  - Наслідок з правила: якщо ввід (input) від користувача взагалі не зрозумілий, потрібно негайно повідомити про це клієнту (*принцип fail-fast*) та запропонувати вихід із ситуації.

Що може піти не так у цьому коді?

```
public class FileStore
{
    public string WorkingDirectory { get; set; }

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

    public string Read(int id)
    {
        var path = this.GetFileName(id);
        var msg = File.ReadAllText(path);
        return msg;
    }

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

- Властивість WorkingDirectory може викликати проблеми наступним чином:
  - ***(1) WorkingDirectory може бути null***, що призведе до виникнення винятку в методі Path.Combine() – у всіх методах класу відбудеться збій.
  - Приклад збійного в рантаймі виклику:

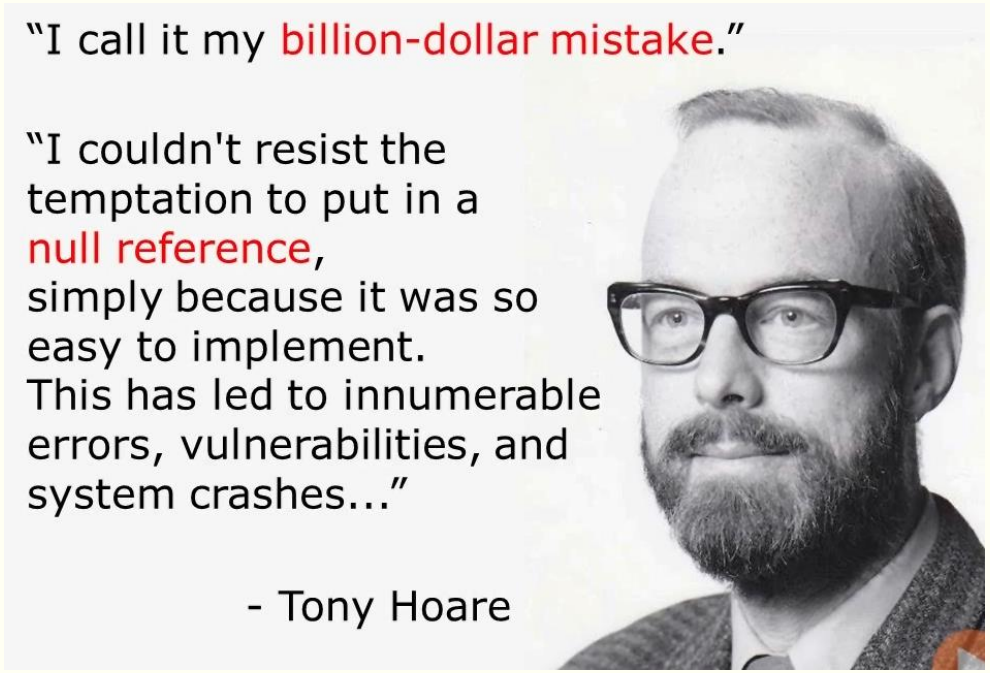
```
var fileStore = new FileStore();
fileStore.Save(42, "Hello world"); // Throws
```

  - Потрібний інваріант – принаймні, передумови для валідації.
  - ***(2) Шлях у WorkingDirectory може бути некоректним***.

Реалізуємо передумови

```
public class FileStore
{
    public FileStore(string workingDirectory)
    {
        if (workingDirectory == null)
            throw new ArgumentNullException("workingDirectory");

        this.WorkingDirectory = workingDirectory;
    }

    public string WorkingDirectory { get; private set; }

    public void Save(int id, string message)

    public string Read(int id)

    public string GetFileName(int id)
}
```

- Крок 1 – додаємо явний конструктор, щоб гарантувати передачу шляху.
- Крок 2: зробити сеттер WorkingDirectory приватним, щоб лише клас міг присвоїти значення.
- Крок 3: швидко реагувати (fail-fast) на передане в конструктор null-значення.
  - У більшості об'єктно-орієнтованих мов рядковий тип є нулабельним.



Реалізуємо передумови

```
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)

    public string Read(int id)

    public string GetFileName(int id)
}
```

- Проблему некоректного шляху також вирішуємо захисною перевіркою в конструкторі (fail-fast підхід).
  - Для демонстрації використано повідомлення «Boo», проте, очевидно, слід писати щось більш змістовне.

Реалізуємо постумови

```
public class FileStore
{
    public FileStore(string workingDirectory)

    public string WorkingDirectory { get; }

    public void Save(int id, string message)

    public string Read(int id)

    public string GetFileName(int id)
}
```

- Зараз клас має три методи-запити.
  - Властивість WorkingDirectory організує лише ехо-вивід шляху, тому нецікава в контексті постумов.
- Які гарантії щодо виводу дають методи Read() та GetFileName()?

```
public string Read(int id)
{
    var path = this.GetFileName(id);
    var msg = File.ReadAllText(path);
    return msg;
}

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

Реалізуємо постумови

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

- Почнемо з простішого методу – GetFileName().
  - Він повертає рядок, проте чи завжди?
  - Метод навіть не перевіряє, чи значення this.WorkingDirectory схоже на шлях до файлу.
  - Передумова, що WorkingDirectory не дорівнює null, гарантує, що метод GetFileName() не поверне null, а поверне нормальний рядок.
- Які гарантії для методу Read()?

```
public string Read(int id)
{
    var path = this.GetFileName(id);
    var msg = File.ReadAllText(path);
    return msg;
}
```

  - Метод GetFileName() завжди повертає string-значення, проте виклик File.ReadAllText() викине виняток, якщо переданий шлях не буде існувати.

```
public string Read(int id)
{
    var path = this.GetFileName(id);
    if (!File.Exists(path))
        throw new ArgumentException("You suck!", "id");
    var msg = File.ReadAllText(path);
    return msg;
}
```

  - Також може передаватись id, якому не відповідає повідомлення (це не обов'язково виняткова ситуація).

Реалізуємо постумови

```
public class FileStore
{
    public FileStore(string workingDirectory)

    public string WorkingDirectory { get; }

    public void Save(int id, string message)

    public string Read(int id)

    public string GetFileName(int id)
}
```

- Перефразуємо запитання: що повинен повертати метод Read(), коли йому переданий id без повідомлення?
  - Порожній рядок – теж повідомлення, тому не може стати індикатором цього стану.
  - Повернення null – теж не вихід, оскільки метод GetFileName() гарантує повернення рядка (хоч тип string і допускає null-значення).
    - Матимемо неузгоджений API: метод GetFileName() завжди повертатиме рядок, а метод Read() – ні.
    - Подальше захисне програмування з перевірками на null тільки ускладнить код.
- Існують різні способи перевірки того, чи можна виконати операцію:
  - Підхід «Tester/Doer» - найстаріший та найпростіший.
  - Підхід «TryRead» («TryParse»).
  - Підхід Maybe, запозичений з функціонального програмування.

Підхід «Tester/Doer»

```
public class FileStore
{
    public FileStore(string workingDirectory)

    public string WorkingDirectory { get; }

    public void Save(int id, string message)

    public bool Exists(int id)
    {
        var path = this.GetFileName(id);
        return File.Exists(path);
    }

    public string Read(int id)
    {
        var path = this.GetFileName(id);
        if (!File.Exists(path))
            throw new ArgumentException("You suck!", "id");
        var msg = File.ReadAllText(path);
        return msg;
    }

    public string GetFileName(int id)
}
```

- Якщо існує операція, щодо якої невідомо, чи можна її застосувати, слід створити окремий метод (тестер), який перевірятиме дану можливість.
  - Введемо метод Exists(), який перевірятиме можливість зчитування повідомлення по id.
  - Виклик відповідного API може бути таким:

```
string message = "";
if (fileStore.Exists(49))
    message = fileStore.Read(49);
```

- Основна проблема підходу – потоконебезпечність.
  - Навіть наприкінці 1990-х рр, коли проєктувався .NET Framework, потокобезпечність не була пріоритетом.
  - У короткий проміжок між викликами Exists() та Read() інший потік може видалити файл, що призведе до збою додатку.
  - В багатопоточних додатках підхід не спрацьовує.

Підхід «TryRead»

```
public class FileStore
{
    public FileStore(string workingDirectory)

    public string WorkingDirectory { get; }

    public void Save(int id, string message)

    public bool Exists(int id)

    public bool TryRead(int id, out string message)
    {
        message = null;
        var path = this.GetFileName(id);
        if (!File.Exists(path))
            return false;
        message = File.ReadAllText(path);
        return true;
    }

    public string GetFileName(int id)
}
```

- Якщо файл не існує, метод TryRead() поверне false, інакше буде зчитано текст з файлу в змінну message та повернуто true.
  - Потреби в методі Exists() тепер немає.
  - Клієнт, який використовує такий API, працює приблизно так:

```
string message = "";
bool exists = fileStore.TryRead(49, out message);
```

  - Виклик TryRead() відбувається атомарно.
  - Хоч гонитва даних можлива, це можна виправити всередині методу TryRead().
  - Проблема – API не дуже об'єктно-орієнтований та зручний: метод TryRead() руйнуватиме ланцюги викликів методів, оскільки повертає не потрібне значення (message), а булеве значення.
  - Більш об'єктно-орієнтований підхід для методів з out-параметрами – виокремити клас, який міститиме як об'єкт вихідного тип методу, так і об'єкт типу out-параметра. Проте теж далеко не ідеальний варіант.

Підхід Maybe

```
public class FileStore
{
    public FileStore(string workingDirectory)

    public string WorkingDirectory { get; }

    public void Save(int id, string message)

    public string Read(int id)
    {
        var path = this.GetFileName(id);
        if (!File.Exists(path))
            throw new ArgumentException("You suck!", "id");
        var msg = File.ReadAllText(path);
        return msg;
    }

    public string GetFileName(int id)
}
```

Проміжний крок

```
public class FileStore
{
    public FileStore(string workingDirectory)

    public string WorkingDirectory { get; }

    public void Save(int id, string message)

    public IEnumerable<string> Read(int id)
    {
        var path = this.GetFileName(id);
        if (!File.Exists(path))
            return new string[0];
        var message = File.ReadAllText(path);
        return new[] { message };
    }

    public string GetFileName(int id)
}
```

- Реалізація все ще перевіряє, чи існує файл.
  - Якщо не існує, то повертається масив з порожнім рядком, інакше – масив з повідомленням.
  - Проблема: розмиваються гарантії щодо вихідного результату роботи методу: може повертатись послідовність з довільною кількістю елементів. Це суперечить закону Постела.

Підхід Maybe

```
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();
    }
}
```

- Потрібно обмежити послідовність, яку повертає метод, одним елементом або їх відсутністю.
  - Це можна зробити, додавши в код написаний власноруч клас – Maybe.
  - Він матиме два перевантажених конструктори: безаргументний для порожньої послідовності та з одним параметром для заповнення послідовності одним елементом.
  - Для простоти тут не вводяться захисні перевірки.
- Нова версія методу Read():

```
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);
}
```

  - Вона може повертати, а може й не повертати рядок.
  - Сама концепція може повторно використовуватись, тут вона гарантує неможливість повернення null.

```
var message = fileStore.Read(49).DefaultIfEmpty("").Single();
```

ДЯКУЮ ЗА УВАГУ!

Наступне запитання: філософія об’єктно-орієнтованого проєктування програмного забезпечення