Лекція 3 **Розробка архітектури ПЗ. принципи Solid**

Слайд 2. Зміст Лекції 3.

Слайд 3. **3.1 Що таке SOLID?**

Що таке гарний дизайн (архітектура ПЗ)? За якими критеріями його оцінювати, і яких правил дотримуватися при розробці? Як забезпечити достатній рівень гнучкості, пов'язаності, керованості, стабільності і зрозумілості коду? Роберт Мартін склав список, що складається всього з п'яти правил хорошого проектування, які відомі, як принципи SOLID.

Досягти такої лаконічності вдалося, використавши невелику хитрість: справа в тому, що термін SOLID - це абревіатура, яка в свою чергу складається з абревіатур, за кожною з яких ховається цілий клас патернів. Нижче ми розглянемо кожен з них.

Слайд 4. Отже, як же розшифровується S.O.L.I.D .:

Single Responsibility Principle (Принцип єдиною обов'язки)
Open Closed Principle (Принцип відкритості / закритості)
Liskov's Substitution Principle (принцип підстановки лісков)
Interface Segregation Principle (Принцип поділу інтерфейсу)
Dependency Inversion Principle (Принцип інверсії залежностей)

Слайд 5.

Пять основных принципов дизайна классов (S.O.L.I.D.) в Java

info.javarush.ru

Название принципа	Описание
1. Single Responsibility Principle (Принцип единственной обязанности)	На каждый объект должна быть возложена одна единственная обязанность.
2. Open Closed Principle (Принцип открытости/закрытости)	Программные сущности должны быть открыты для расширения, но закрыты для изменения.
3. Liskov's Substitution Principle (Принцип подстановки Барбары Лисков)	Объекты в программе могут быть заменены их наследниками без изменения свойств программы.
4. Interface Segregation Principle (Принцип разделения интерфейса)	Много специализированных интерфейсов лучше, чем один универсальный.
5. Dependency Inversion Principle (Принцип инверсии зависимостей)	Зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Слайд 6.

3.2 Принцип єдиної відповідальності

Формулювання: не повинно бути більше однієї причини для зміни класу

Що є причиною зміни логіки роботи класу? Мабуть, зміна відносин між класами, введення нових вимог або скасування старих. Взагалі, питання про причини цих змін лежить в площині ответственності, яку ми поклали на наш клас. Якщо в об'єкта багато відповідальності, то і змінюватися він буде дуже часто. Таким чином, якщо клас має більше однієї відповідальності, то це веде до крихкості дизайну і помилок в несподіваних місцях при змінах коду.

SRP призначений для боротьби зі складністю. Насправді, будь-який принцип або патерн проектування призначений для цього. Коли в нашому додатку всього 200 рядків, то дизайн як такий взагалі не потрібен. Досить акуратно написати 5-7 методів і все буде добре. Проблеми виникають, коли система зростає і збільшується в масштабах. Оскільки складність зростає не лінійно при збільшенні розміру програми доводиться підходити до дизайну більш осмислено, згадувати всякі страшні слова, типу абстракції і приховування інформації, краще продумувати обов'язки кожного класу, щоб у розробника сьогодні і через рік була можливість зосередитися на головній проблемі класу / модуля і проігнорувати другорядні деталі.

Слайд 7. Приклади.

1. Active Record

Розглянемо приклад. Нехай у нас є клас, який реалізує деяку функціональність, пов'язану з банківським рахунком:class Account : ActiveRecord

```
{
   public Guid Id{ get{ ... } }
   public string Number { get{ ... } }
   public decimal CurrentBallance { get { ... } }
   public void Deposit(decimal amount){ ... }
   public void Withdraw(decimal amount){ ... }
   public void Transfer(decimal amount, Account recipient){ ... }
   public TaxTable CalculateTaxes(int year){ ... }
}
```

Як видно, клас несе відповідальність за:

1. Персистентность (збереження змін);

2. Логіку управління балансом;

3. Логіку розрахунку податків.

Слайд 8. Всі ці "місії" і є тими самими мотивами, які впливають на життєвий цикл класу. Провівши рефакторинг, можна отримати наступний код:

```
class Account
{
```

```
    public string Number {get {...}}
    public decimal CurrentBallance {get {...}}
    public void Deposit (decimal amount) {...}
    public void Withdraw (decimal amount) {...}
    public void Transfer (decimal amount, Account recipient) {...}
}
class AccountRepository
{
    public Account GetByNumber (string number) {...}
    public void Save (Account acc) {...}
}
class TaxCalculator
{
    public TaxTable CalculateTaxes (Account acc, int year) {...}
}
```

А складність у застосуванні даного принципу полягає в тому, що перш за все потрібно навчитися правильно відчувати кордону його використання. Адже навіть в наведеному прикладі ми перетворили патерн Active Record в антипаттерн, разом перекресливши всі приклади його успішного застосування.

Слайд 9. ***2. Валідація даних***

Проблема

Якщо ви зробили хоча б один проект, то перед вами напевно стояла проблема валідації даних. Наприклад, перевірка введеного адреси ел. пошти, довжини імені користувача, складності пароля і т.п.

Рішення

Стало очевидно, що при подальшому використанні об'єкта логіка валідації його даних буде змінюватися і ускладнюватися. Мабуть пора віддати відповідальність за затвердження даних іншому об'єкту. Причому треба зробити так, щоб сам об'єкт не залежав від конкретної реалізації його валідатора. Маємо об'єкт окремо, а будь-яку кількість усіляких валідаторів окремо.

Слайд 10. ***3. God object***

Проблема

Межа порушення принципу єдиності відповідальності - God object. Цей об'єкт знає і вміє робити все, що тільки можна. Наприклад, він робить запити до бази даних, до файлової системи, спілкується по протоколам в мережу і содержить тонну бізнес-логіки.

Здається, що межі відповідальності у нього взагалі немає. Він може зберігати в базу даних, причому знає правила призначення зображень користувачам. Може завантажувати зображення. Знає, як зберігаються файли зображень і може працювати з файлової системою.

Кожна відповідальність цього класу веде до його потенційної зміни. Виходить, що цей клас буде дуже часто міняти свою поведінку, що утруднить

його тестування і тестування компонентів, які його використовують. Такий підхід знизить працездатність системи і підвищить вартість її супроводу.

Рішення

Рішенням є розділити цей клас за принципом єдиної відповідальності: один клас на одну відповідальність**.**

Слайд 11.

3.3 Принцип відкритості / закритості

Формулювання: програмні сутності (класи, модулі, функції і т.д.) повинні бути відкриті для розширення, але закриті для зміни

Таким чином у модулів є дві основні характеристики:

• **Вони відкриті для розширення.** Це означає, що поведінка модуля можна розширити. Коли вимоги до додатка змінюються, ми додаємо в модуль нову поведінку, що відповідає зміненим вимогам. Іншими словами, ми можемо змінити склад функцій модуля**.**

• **Вони закриті для модифікації.** Розширення поведінки модулів не пов'язане зі змінами в початковому або довічним коді модуля. **Двійкове виконується уявлення модуля, будь то компонований бібліотека, DLL або EXE-файл, залишається незмінним.**

Слайд 12. Яку мету ми переслідуємо, коли застосовуємо цей принцип? Як відомо програмні проекти протягом свого життя постійно змінюються. Зміни можуть виникнути, наприклад, через нові вимог замовника або перегляду старих. В кінцевому підсумку буде потрібно змінити код згідно з поточною ситуацією.

З одного боку внесення змін вимагає часу програмістів і тестувальників, яке є дуже дорогим ресурсом у виробництві програмного забезпечення. З іншого, бізнес повинен досить швидко реагувати на ринкові зміни і час тут представляється дуже важливою конкурентною перевагою.

Звідси можна зробити висновок, що нашою метою є розробка системи, яка буде досить просто і безболісно змінюватися. Іншими словами, система повинна бути гнучкою. Наприклад, внесення змін в бібліотеку загальну для 4х проектів не повинно бути довгим («довгим» є різним проміжком часу для конкретної ситуації) і вже точно не повинно вести до змін в цих 4х проектах.

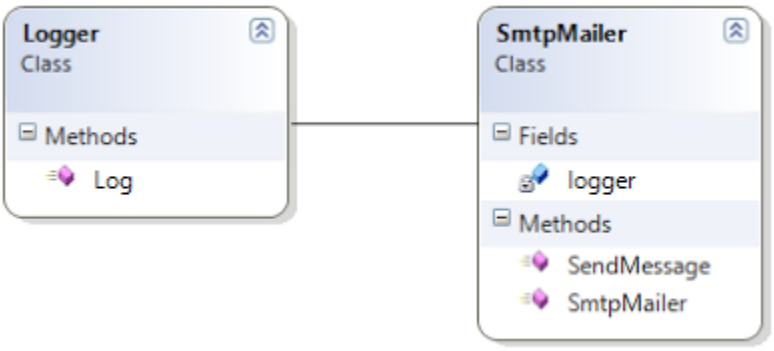
Принцип відкритості / закритість якраз і дає розуміння того, як залишатися досить гнучкими в умовах постійно мінливих вимог.

Слайд 13. **Приклади**

Без абстракцій

Проблема

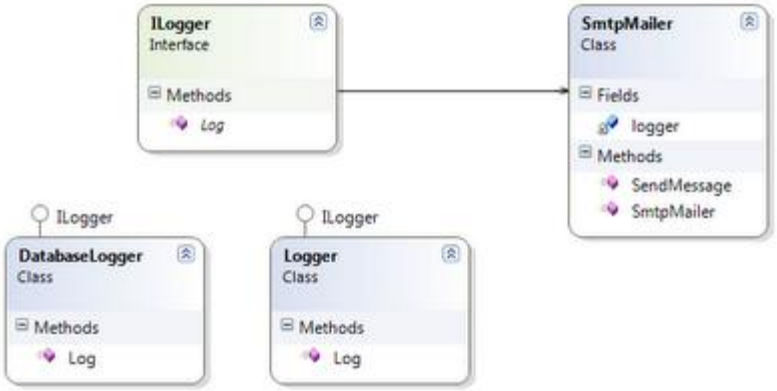
Найпростіший приклад порушення принципу відкритості / закритості - використання конкретних об'єктів без абстракцій. Припустимо, що у нас є об'єкт SmtpMailer. Для логування своїх дій він використовує Logger, який записує інформацію в текстові файли.



Така конструкція цілком життєздатна до тих, поки ми не вирішимо записувати лог SmptMailer'a в базу даних. Для цього нам треба створити клас, який буде записувати всі логи не в текстовий файл, а в базу даннях. А тепер найцікавіше. Ми повинні змінити клас SmptMailer через зміненого бізнес-вимоги. Але ж за принципом єдиності відповідальності не SmptMailer відповідає за логирование, чому зміни дійшли і до нього? Тому що порушений наш принцип відкритості / закритості. SmptMailer не закрите для модифікації. Нам довелося його змінити, щоб поміняти спосіб зберігання його логів.

Слайд 14. **Рішення**

В даному випадку захистити SmtpMailer допоможе виділення абстракції. Нехай SmtpMailer залежить від інтерфейсу ILogger:



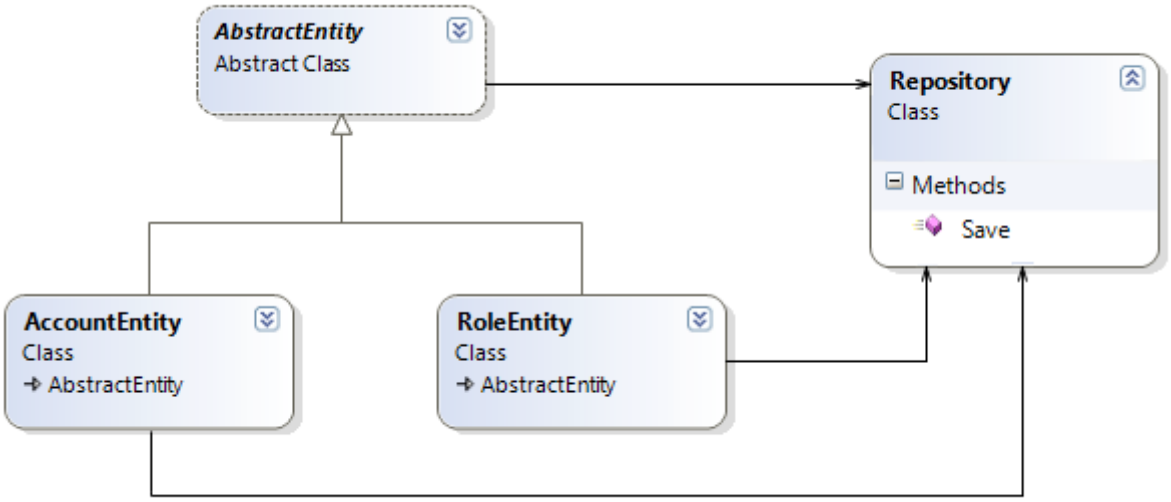
Тепер зміна логіки логування вже не буде вести до модифікації SmtpMailer'a.

Слайд 15. **Перевірка типу абстракції**

Проблема

Цей приклад в різних варіаціях все не раз бачили в коді. Його хоч в рамку можна вішати, як найпопулярніше порушення проектування. У нас є ієрархія об'єктів з абстрактним батьківським класом AbstractEntity і клас Repository, який використовує абстракцію. При цьому викликаючи метод Save у Repository

ми будуємо логіку в залежності від типу вхідного параметра:



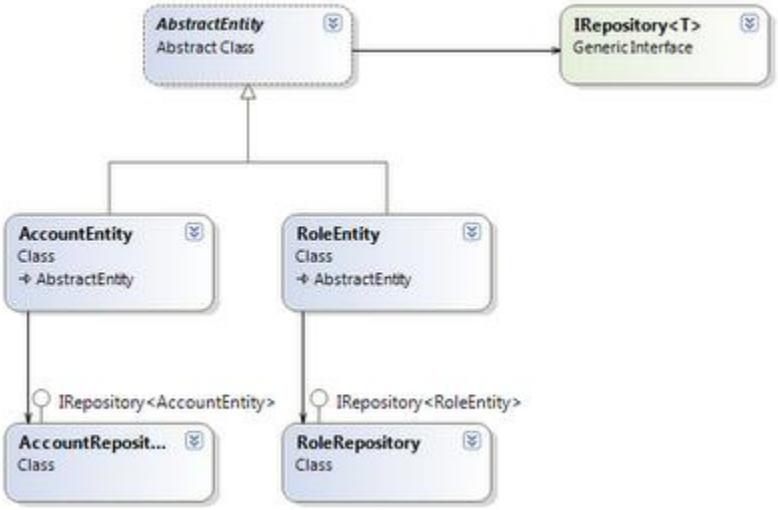
Слайд 16.

```
public abstract class AbstractEntity
{
}

public class AccountEntity : AbstractEntity
{
}

public class RoleEntity : AbstractEntity
{
}

public class Repository
{
    public void Save(AbstractEntity entity)
    {
        if (entity is AccountEntity)
        {
            // специфические действия для AccountEntity
        }
        if (entity is RoleEntity)
        {
            // специфические действия для RoleEntity
        }
    }
}
```

З коду видно, що об'єкт Repository доведеться міняти кожен раз, коли ми додаємо в ієрархію об'єктів з базовим класом AbstractEntity нових спадкоємців або видаляємо існуючих. Умовні оператори будуть множиться в методі Save і тим самим ускладнювати його.

Слайд 17. **Рішення**

Конкретизуючи класи методом is або typeof ми повинні відразу зрозуміти, що наш код почав «тхнути». Щоб вирішити дану проблему, необхідно логіку збереження конкретних класів з ієрархії AbstractEntity винести в конкретні класи Repository. Для цього ми повинні виділити інтерфейс IRepository і створити сховища AccountRepository і RoleRepository:



Тепер наші зміни будуть локалізовані в конкретних об'єктах.

Слайд 18.

3.4 Принцип заміщення Лісков

Формулювання №1: **если для кожного об'єкта o1 типу S існує об'єкт o2 типу T, який для всіх програм P визначено в термінах T, то поведінка P не зміниться, якщо o2 замінити на o1 за умови, що S є підтипом T.**

Формулювання №2: **Функції, які використовують посилання на базові класи, повинні мати можливість використовувати об'єкти похідних класів, не знаючи про це.**

Слайд 19. Цей принцип є варіацією принципу відкритості / закритості, про який говорилося раніше. У ньому йдеться:

Об'єкти в програмі можуть бути замінені їхніми спадкоємцями без зміни властивостей програми.

Це означає, що клас, розроблений на підставі вашого базового класу шляхом розширення, повинен працювати в додатку без збоїв. Тобто, якщо розробник розширює ваш клас і використовує його в додатку, він не повинен порушувати роботу програми або створювати фатальні помилки для всього програми.

Цього легко домогтися, якщо пам'ятати одне просте правило: Якщо ваш базовий клас робить строго одна справа, розробник отримає при використанні класу тільки одну проблему. Це може привести до деяких помилок в одній області, але все програма не буде працювати неправильно.

Слайд20 **Принцип поділу інтерфейсу**

Формулювання: клієнти не повинні залежати від методів, які вони не використовують

Як і при використанні інших принципів проектування класів ми намагаємося позбутися непотрібних залежностей в коді, зробити код легко читаним і легко змінним.

Слайд21 **Принцип поділу інтерфейсів** говорить про те, що занадто «товсті» інтерфейси необхідно розділяти на більш дрібні і специфічні, щоб клієнти маленьких інтерфейсів знали тільки про методи, які необхідні їм у роботі. У підсумку, при зміні методу інтерфейсу не повинні змінюватися клієнти, які цей метод не використовують. Розглянемо приклад. Розробник Алекс створив інтерфейс «звіт», і додав два методи: generateExcel () і generatedPdf (). Тепер клієнт А хоче використовувати цей інтерфейс, але він має намір використовувати звіти тільки в PDF форматі, а не в Excel. Чи влаштує його така функціональність?

Ні. Він повинен буде реалізувати два методи, один з яких за великим рахунком не потрібен, і існує тільки завдяки Алексу - дизайнеру програмного забезпечення. Клієнт скористається або іншим інтерфейсом, або залишить поле для Excel порожнім.

Так в чому ж рішення? Рішення полягає в поділі існуючого інтерфейсу на два дрібніших. Один - звіт в форматі PDF, другий - звіт в форматі Escel. Це дасть користувачеві можливість використовувати тільки необхідний для нього функціонал.

Слайд22 **Принцип інверсії залежності**

***формулювання*:**

• Модулі верхнього рівня не повинні залежати від модулів нижнього рівня. Обидва повинні залежати від абстракції.

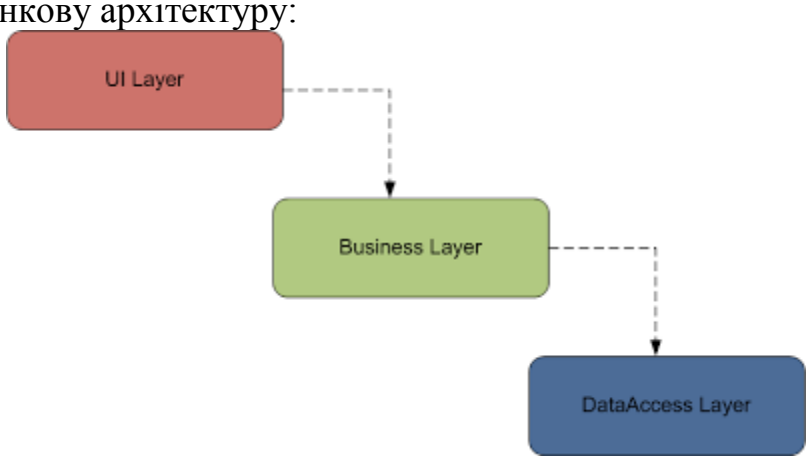
• Абстракції не повинні залежати від деталей. Деталі повинні залежати від абстракцій.

Іншими словами, слід розробляти програмне забезпечення таким чином, що різні модулі були автономними, і з'єднувалися один з одним за допомогою абстракції.

Слайд23 **Приклад.**

інвертована архітектура

Погляньмо на архітектуру програми. Давайте розглянемо класичну триланкову архітектуру:



Слайд24 Високорівневі модулі програми не відділені від низькорівневих реалізацій. Абстракції не відокремлені від деталей. Зміна логіки в шарі доступу

до даних може несподівано призвести до поломки в модулі відображення. Тестувати таку систему буде дуже складно. Навіть якщо вийде написати модульні тести, то будь-яка зміна в системі призведе до того, що ці тести доведеться переписувати. В результаті ця система володіє характеристиками:

1. Жорсткість

Важко змінювати систему, тому що кожна зміна зачіпає дуже багато різних її частин.

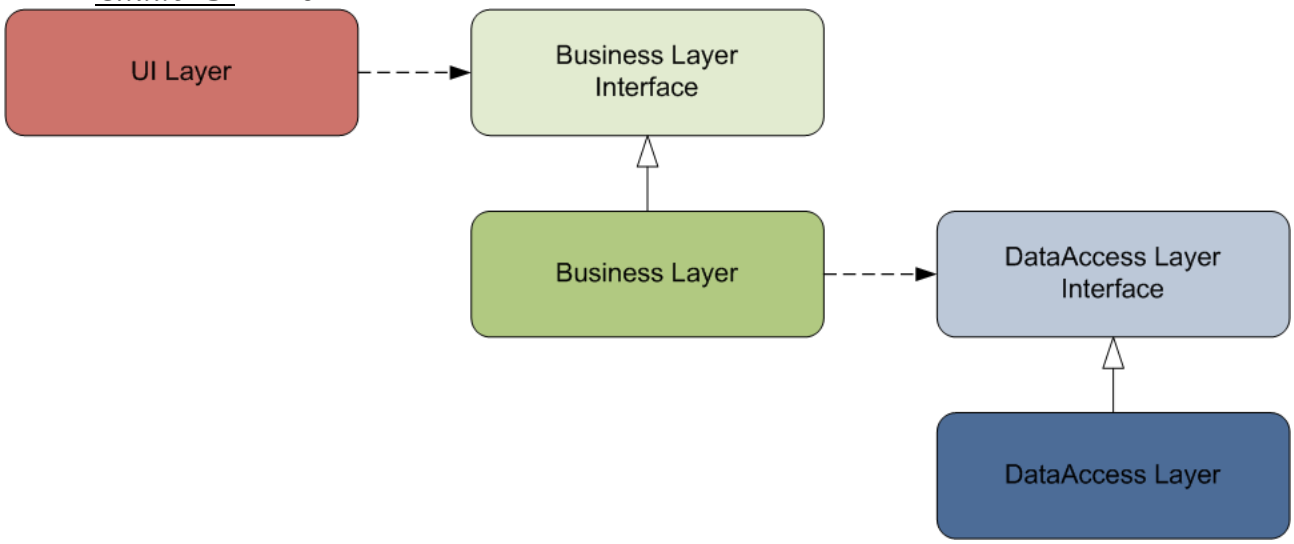
2. Крихкість

Коли ви вносите зміни в одну частину системи, то в несподіваному місці ламається інша.

3. Нерухомість

Дуже складно повторно використовувати код в іншому додатку, тому що модулі сильно пов'язані між собою.

Слайд25 **Рішення**



Запитання до Лекції 3.

1. Принципи SOLID при розробці архітектури ПЗ.
2. Що таке принципи SOLID? Перерахуйте та опишіть їх.
3. Опишіть Single Responsibility Principle (Принцип єдиності обов'язків).
4. Опишіть Open Closed Principle (Принцип відкритості / закритості).
5. Опишіть Liskov's Substitution Principle (принцип підстановки лісков).
6. Опишіть Interface Segregation Principle (Принцип поділу інтерфейсу).

7. Опишіть Dependency Inversion Principle (Принцип інверсії залежностей).
8. Надайте визначення та охарактеризуйте принцип єдиної відповідальності.
9. Надайте визначення та охарактеризуйте принцип відкритості / закритості.
10. Валідація даних в процесі проектування архітектури ПЗ.
11. В чому полягає межа порушення принципу єдиності відповідальності God object.
12. Логування своїх дій в об'єктах SmtpMaile.
13. Яким чином відбувається перевірка типу абстракції?
14. Скільки і яких Ви знаєте формулюваннь принципу заміщення Лісков?
15. Опишіть принцип поділу інтерфейсу.
16. Опишіть принцип інверсії залежності.
17. Які існують характеристики високорівневих модулів програми, що не відділені від низькорівневих реалізацій?