GeekBrains

Архитектура ПО

Урок 3

Принципы SOLID

Оглавление

На этом уроке

1. Узнаем, какими бывают программные структуры нижнего и среднего уровней.
2. Рассмотрим принципы, составляющие подход проектирования ПО SOLID.

Введение

Проектирование — один из этапов создания программного приложения или системы программных приложений. Этот этап есть в большинстве методологий разработки программного обеспечения.

Сущность блочно-иерархического подхода к проектированию — в разделении описания объекта по разным иерархическим уровням, которые соответствуют степени детализации свойств объекта или его части. То есть в начале первой стадии структура объекта носит характер закрытой, неизвестной, требующей детализации и уточнения, структуры.

Описание первого уровня даёт определённую степень детализации, выявляет наличие связей между блоками и устанавливает иерархическую связь с другим уровнем. Затем таким же образом детализируется и прорабатывается второй и последующие уровни.

Детализация считается достаточной, когда блоки последнего уровня демонстрируют простую и прозрачную структуру.

На каждом уровне преобладают свои формы документации, собственный математический аппарат, способствующий построению алгоритмов, своя совокупность средств, а именно: моделей, методов и прочих, присущих этому уровню.

1. Программные структуры нижнего и среднего уровней

Программные структуры нижнего уровня — это классы и модули.

Классы — шаблоны, на основе которых создаётся конкретный программный объект. Класс описывает свойства и методы, определяющие поведение объектов этого класса.

Модули — наборы отдельных, не зависящих друг от друга, блоков кода при построении программного приложения.

Принцип модульности служит средством упрощения задачи проектирования программного обеспечения (ПО) и распределения процесса разработки между группами разработчиков. При разбиении ПО на модули для каждого модуля указывается реализуемая им функциональность, а также связи с другими модулями. Удобство использования модульной архитектуры заключается в возможности обновления (замены) модуля, без изменения остальной системы.

Программные структуры среднего уровня — компоненты. Это множества классов и языковых конструкций, объединённых по общему признаку и организованных в соответствии с определёнными правилами и ограничениями.

Компоненты — это единицы развёртывания. Они представляют наименьшие сущности, развёртываемые в составе системы.

1. В Java — jar-файлы.
2. В Ruby — gem-файлы.
3. В .NET — библиотеки DLL.
4. В компилирующих языках — комплексы двоичных файлов.
5. В интерпретирующих языках — комплексы файлов с исходным кодом — например, Python .py.

2. Принципы SOLID

Хорошая программная система начинается с чистого и понятного кода, спроектированного по определённым принципам.

SOLID — это акроним пяти важных принципов проектирования при выполнении ООП (объектно-ориентированного программирования). Они связаны с проектированием и сопровождением программных систем.

Сами принципы:

1. SRP: Single Responsibility Principle. Принцип единственной ответственности.
2. OCP: Open-Closed Principle. Принцип открытости или закрытости.
3. LSP: Liskov Substitution Principle. Принцип подстановки Барбары Лисков.
4. ISP: Interface Segregation Principle. Принцип разделения интерфейсов.
5. DIP: Dependency Inversion Principle. Принцип инверсии зависимости.

Эти пять принципов ввёл Роберт Мартин в своей работе 2000 года «Принципы проектирования и паттерны проектирования». Он начал собирать их в конце 1980-х годов, обсуждая принципы проектирования программного обеспечения с другими пользователями Usenet — ранняя разновидность социальной сети. В течение многих лет принципы смещались и изменялись. Некоторые исчезали, другие объединялись, а какие-то добавлялись. Окончательный вид они приобрели лишь в 2000 году.

Однако на самом деле аббревиатуру SOLID ввёл в 2004 году Майкл Фезерс. Он сообщил, что если переупорядочить принципы, то из первых букв можно составить слово SOLID.

Цель этих принципов — сделать дизайн структуры программного обеспечения более понятным, более простым в сопровождении и более лёгким в расширении, которые:

- терпимы к изменениям;
- просты и понятны;
- просто поддерживаются, легко расширяются и внедряются;
- образуют основу для компонентов, используемых во многих программных системах.

💡 Инженеру-программисту эти пять принципов надо твёрдо понимать и знать!

Принципы SOLID указывают, как объединять функции и структуры данных в классы, и как эти классы должны сочетаться друг с другом.

Рассмотрим каждый принцип на примерах.

2.1. SRP: Single Responsibility Principle. Принцип единственной ответственности

Класс должен быть ответственен лишь за что-то одно. Если класс отвечает за решение нескольких задач, его подсистемы, реализующие решение этих задач, оказываются связанными друг с другом. Изменения в одной такой подсистеме ведут к изменениям в другой.

Принцип единственной ответственности описывается таким образом:

1. «Модуль должен иметь одну и только одну причину для изменения».
2. «Модуль должен отвечать за одного и только за одну сущность, за один актор».

Актор — примитивная единица, получающая сообщения и делающая примитивные вычисления, основанные на этих сообщениях.

Акторы изолированы друг от друга и имеют внутреннее состояние, которое не может быть изменено извне.

Если разные люди работают с одной программой, то изменение части при взаимодействии с одним человеком не повлияет на часть, с которой работает другой.

Пример 1

```
class Animal {
    Animal(string name);
    string getAnimalName();
    void saveAnimal(Animal a);
}
```

Листинг 1. Описание класса Animal

В соответствии с принципом единственной ответственности класс должен решать лишь какую-то одну задачу. Но он решает две, занимаясь работой с хранилищем данных в методе saveAnimal и манипулируя свойствами объекта в конструкторе и в методе getAnimalName.

Если изменится порядок работы с хранилищем данных, используемым приложением, то придётся вносить изменения во все классы, работающие с хранилищем.

Чтобы привести код в соответствии с принципом единственной ответственности, создадим ещё один класс, единственная задача которого — работа с хранилищем, в частности — сохранение в нём объектов класса.

```
class Animal {
    Animal(string name);
    string getAnimalName();
}

class AnimalGS {
    string getAnimal(Animal a);
    void saveAnimal(Animal a);
}
```

Листинг 2. Описание классов Animal и AnimalGDS

Объединение двух сущностей, которые меняются по разным причинам в разное время, считается плохим дизайном.

Принцип **SRP** применяется:

- когда надо сделать код гибким и легко изменяемым;
- сложно заранее определить вектор изменений;
- разделение функционала — долгая и сложная операция, нежели его объединение, поэтому отдавайте предпочтение декомпозиции. То есть старайтесь сразу создать архитектуру из небольших, независимых друг от друга сущностей.

Не используйте этот принцип, когда:

- заранее известно о неизменяемости кода в конкретном месте;
- решение сильно усложняет разработку и поддержку кода. Всегда надо сохранять баланс между количеством классов и их необходимостью.

💡 «Проектируя классы, мы должны стремиться к тому, чтобы объединять родственные компоненты, то есть такие, изменения в которых происходят по одним и тем же причинам. Нам следует стараться разделять компоненты, изменения в которых вызывают различные причины», — Стив Фентон.

Правильное применение принципа единственной ответственности приводит к высокой степени связности элементов внутри модуля, а именно: к тому, что задачи, решаемые внутри него, хорошо соответствуют его главной цели.

💡 Этот принцип применим не только к классам, но и к компонентам программного обеспечения в более широком смысле.

2.2. OCP: Open-Closed Principle. Принцип открытости или закрытости

Принцип открытости-закрытости

Программные сущности (классы, модули, функции) открыты для расширения, но закрыты для модификации. Это означает, что должна быть возможность изменять внешнее поведение класса, не внося физических изменений в сам класс. Следуя этому принципу, классы разрабатываются так, чтобы для подстройки класса к конкретным условиям применения было достаточно расширить его и переопределить некоторые функции.

Главная концепция этого принципа — то, что класс открыт для расширений, но закрыт от модификаций.

Мы должны разработать свой модуль так, чтобы новая функциональность добавлялась только при создании новых требований.

Статус «Закрыт для модификации» означает, что мы уже разработали класс, и он прошёл модульное тестирование. Не меняем его, пока не найдём ошибки. Как говорится, класс должен быть открытым только для расширений, и мы можем использовать для этого наследование.

Открыт для расширения

То есть класс создаётся таким образом, чтобы поведение класса можно было дополнить в любой момент. По мере изменения требований мы должны иметь возможность дополнить класс новой функциональностью.

Закрыт для модификации

При добавлении новой функциональности нельзя вносить изменения в исходный код такого класса. То есть весь новый код не должен затрагивать старый.

Суть принципа

Важно, чтобы изменения в программе происходили при написании нового кода, а не модификации старого.

Принцип открытости или закрытости сформулировал Бертран Мейер в 1988 году: «Программные сущности должны быть открыты для расширения и закрыты для изменения». Важно расширять поведение программных сущностей без их изменения. Если строго соблюдать этот принцип, можно регулировать поведение кода без изменения самого исходника.

Пример 1

Продолжим работу с классом Animal.

```
class Animal {
    Animal(string name);
    string getAnimalName();
}
```

Листинг 3. Описание класса Animal

На этот раз требуется перебрать список животных и указать, какие звуки они издают.

Для этого создадим функцию AnimalSound.

```
Animal animals[2] = {
   Animal("Lion"),
   Animal("Cat")
};

string AnimalSound(Animal animal)
```

```
{
        if (animal.getAnimalName() == "Lion")
             return "roar";
        else if (animal.getAnimalName() == "Cat")
             return "meow";
}

for (int i = 0; i < 2; i++)
    printf("%s", AnimalSound(animals[i]))
```

Листинг 4. Описание функции AnimalSound

Добавим в массив новый элемент.

```
Animal animals[3] = {
   Animal("Lion"),
   Animal("Cat",),
   Animal("Mouse")
};
```

Листинг 5. Пример исходного массива

Придётся добавить новое условие в функции AnimalSound.

```
string AnimalSound(Animal animal)
{
        if (animal.getAnimalName() == "Lion")
             return "roar";
        else if (animal.getAnimalName() == "Cat")
             return "meow";
        else if (animal.getAnimalName() == "Mouse")
             return "squeak";
}

for (int i = 0; i < 3; i++)
    printf("%s", AnimalSound(animals[i]));
```

Листинг 6. Описание функции AnimalSound при добавлении нового элемента в массив

В этом примере достаточно просто исправить ситуацию, например, добавив одно условие. Но если подобная архитектура используется в реальном

проекте, функцию придётся постоянно расширять, добавляя в неё новые выражения условия.

Приведём функцию AnimalSound к соответствию принципу открытости или закрытости.

```
class Animal {
    //...
    string getSound();
    //...
}

class Lion : public Animal {
    //...
    string getSound() {
        return("roar");
    }
    //...
}

class Cat: public Animal {
    //...
    string getSound() {
        return("meow");
    }
    //...
}

class Mouse: public Animal {
    //...
    string getSound() {
        return("squeak");
    }
    //...
}

for (int i = 0; i < 3; i++)
    printf("%s ", AnimalSound(animals[i]));
```

Листинг 7. Применение принципа открытости или закрытости

Принцип открытости или закрытости — одна из движущих сил в архитектуре систем. Его цель — сделать систему легко расширяемой и обезопасить её от

влияния изменений. Эта цель достигается делением системы на компоненты и упорядочением их зависимостей в иерархию, защищающую компоненты уровнем выше от изменений в компонентах уровнем ниже.

OCP применяется:

- когда место в программе подвержено частым изменениям;
- в программу входит множество блоков if/case else;
- можно использовать абстракции для построения более общих классов, которые подвержены изменениям.

Не используйте OCP:

- когда на придумывание абстракций уходит много времени;
- неудачные абстракции засоряют код и усложняют рефакторинг;
- глобальное применение абстракций увеличивает сложность понимания и сопровождение проекта.

2.3. LSP: Liskov Substitution Principle. Принцип подстановки Барбары Лисков

В 1988 году Барбара Лисков написала следующие строки с формулировкой определения подтипов: «Если для каждого объекта o1 типа S существует такой объект o2 типа T, что для всех программ P, определённых в терминах T, поведение P не изменяется при подстановке o1 вместо o2, то S является подтипом T».

Барбара Лисков — американский учёный в области информатики, исследователь проблемы абстракции данных, руководитель группы по разработке языка программирования Клу (*CLU)*, лауреат премии Тьюринга 2008 года.

Принцип подстановки Барбары Лисков

Важно, чтобы подклассы служили заменой для своих суперклассов.

Цель этого принципа заключаются в том, чтобы классы-наследники использовались вместо родительских классов, от которых они образованы, не нарушая работы программы. Если оказывается, что в коде проверяется тип класса, значит, принцип подстановки нарушается.

Опишем его так: объекты в программе можно заменить их наследниками без изменения свойств программы.

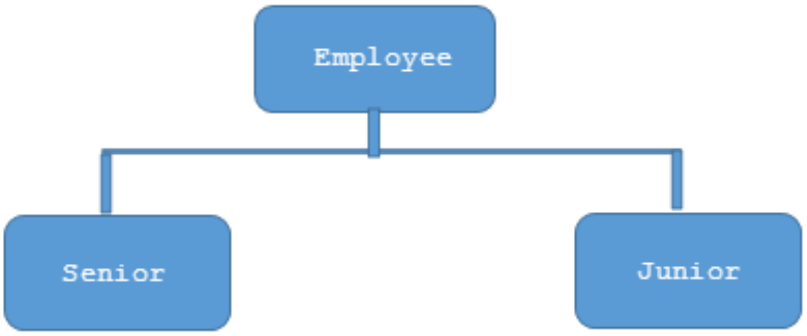
Одна из самых важных идей состоит в том, что у различных реализаций некоего интерфейса совпадает не только интерфейс, но и поведение полностью: с точностью до логики генерации исключительных ситуаций и политики обработки ошибок.

Это означает, что класс, разработанный путём расширения на основании базового класса, должен переопределять его методы так, чтобы не нарушалась функциональность, с точки зрения клиента. Другими словами, если разработчик расширяет ваш класс и использует его в приложении, он не должен изменять ожидаемое поведение переопределённых методов.

Подклассы должны переопределять методы базового класса так, чтобы не нарушалась функциональность, с точки зрения клиента.

Важно, чтобы класс Child не нарушал определение типа родительского класса и его поведение.

Чтобы понять, в чём смысл этого, рассмотрим следующее изображение:



Принцип гласит, что поведение методов в дочернем классе должно следовать принципам базового класса, а не изменять их. То есть дочерний класс, переопределяя методы или переменные, не меняет заложенную логику базового класса.

Пример 1

Продолжим работу с классом Animal.

Рассмотрим функцию AnimalLegCount, которая возвращает количество конечностей животного.

```
//...
int AnimalLegCount(Animal animal)
{
    if (typeid(animal).name() == "Lion")
        return LionLegCount();
    else if (typeid(animal).name() == "Cat")
        return CatLegCount();
    else if (typeid(animal).name() == "Mouse")
        return MouseLegCount();
}
//...
for (int i = 0; i < 3; i++)
    printf("%d ", AnimalLegCount(animals[i]));
```

Листинг 8. Описание функции AnimalLegCount

Функция нарушает принцип подстановки и принцип открытости или закрытости. Этот код должен знать о типах всех обрабатываемых им объектов и, в зависимости от типа, обращаться к соответствующей функции для

подсчёта конечностей конкретного животного. При создании нового типа животного функцию придётся переписывать.

Поэтому надо переписать функцию в соответствии с принципом подстановки. Функция не интересуется типами передаваемых ей объектов. Она просто вызывает методы LegCount.

```
int AnimalLegCount(Animal animal)
{
    return animal.LegCount();
}
//...
for (int i = 0; i < 3; i++)
    printf("%d ", AnimalLegCount(animals[i]));
```

Листинг 9. Применение принципа подстановки Барбары Лисков

Теперь в классе Animal появится метод LegCount, в подклассах его надо реализовать.

```
class Animal {
    //...
    int  LegCount();
    //...
}
//...
class Lion : Animal {
    //...
    int LegCount() {
        //...
    }
    //...
}
//...
```

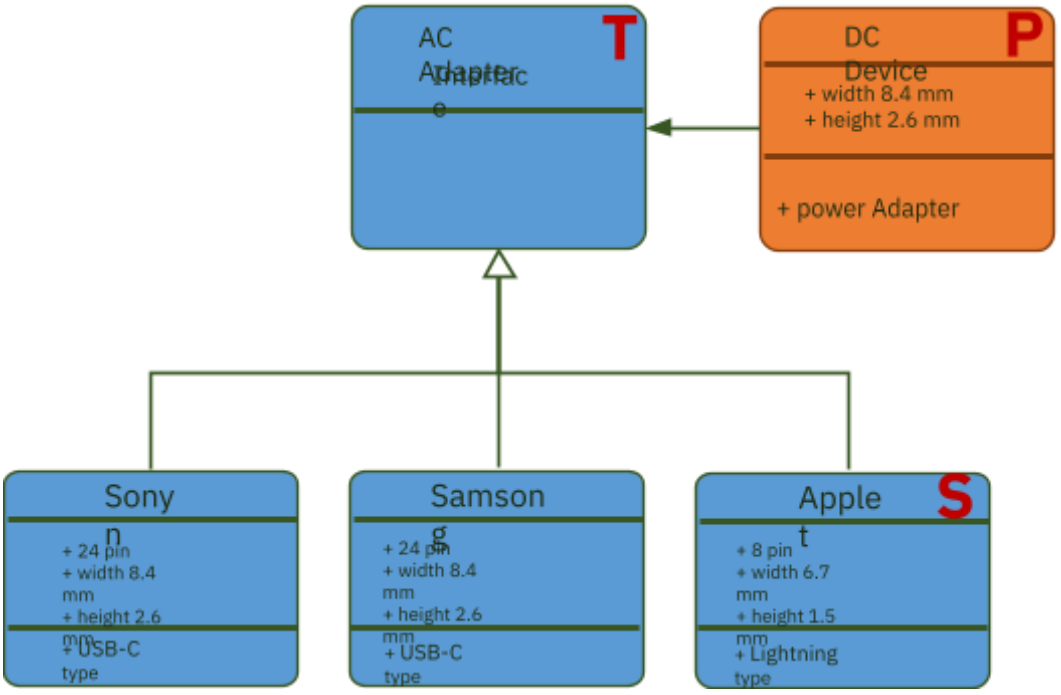
Листинг 10. Описание класса Animal и его подклассов в соответствии с принципом подстановки

Теперь функции AnimalLegCount не надо знать о том, объект какого именно подкласса класса Animal она обрабатывает, чтобы узнать сведения о количестве конечностей животного, представленного этим объектом. Функция просто вызывает метод LegCount класса Animal, так как подклассы такого

класса реализуют этот метод, чтобы использовать их вместо него, не нарушая правильности работы программы.

Пример нарушения LSP

S — **неподходящий** подтип T, если при замене T на S поведение P изменится.



2.4. ISP: Interface Segregation Principle. Принцип разделения интерфейсов

Принцип разделения интерфейса

Лучше создавать много отдельных узкоспециализированных интерфейсов, чем один, который включает в себя множество функций. Это позволит сделать архитектуру более гибкой и использовать интерфейсы по отдельности.

Этот принцип похож на первый, принцип единой ответственности.

💡 Клиенты не должны попадать в зависимость от методов, которыми они не пользуются.

Интерфейс считается абстракцией внешнего представления класса и описывает все доступные методы для работы с ним. В языках, где нет поддержки интерфейсов, они эмулируются через классы с пустыми методами. То есть интерфейсы — классы, у которых есть исключительно публичные методы, и у них нет реализации, они реализуются классами, их использующими.

Этот принцип призывает разработчиков программного обеспечения избегать зависимости от всего, что не используется. То есть слишком большие интерфейсы требуется разделять на более маленькие, чтобы программные сущности маленьких интерфейсов знали только о методах, которые требуются им в работе.

Цели принципа:

1. Принцип борется с толстыми интерфейсами — с интерфейсами на все случаи жизни.
2. По сути, это принцип персональной ответственности, но для интерфейсов, то есть каждый интерфейс должен делать одну свою задачу.
3. Интерфейс должен быть абстрактным — иметь универсальное имя и никому не принадлежать.

Принцип применяется

1. Для создания абстракций (повышение гибкости).
2. Используется в принципе DIP.

Опасности

1. Добавление метода в интерфейс заставляет реализовывать его в классах-наследниках.
2. Заранее хорошо продумывайте абстракции.
3. Заранее продумывайте интерфейс, а потом — класс.

Если происходит наоборот: сначала реализовывается класс, а потом часть его методов переходит в интерфейс, стоит задуматься, а всё ли вы делаете правильно. Это пагубная привычка и признак возможной проблемы.

Суть принципа

1. Много специализированных интерфейсов лучше, чем один универсальный.
2. Этот принцип призывает разработчиков программного обеспечения избегать зависимости от всего, что не используется. То есть слишком большие интерфейсы надо разделять на более маленькие, чтобы программные сущности маленьких интерфейсов знали только о методах, которые требуются им в работе.

Пример 1

Рассмотрим, например, действия архитектора, работающего над системой S. Он пожелал включить в систему некоторый фреймворк F. Авторы F связали его с поддержкой конкретной базы данных D. То есть S зависит от F, который зависит от D.

Некорректная архитектура

Допустим, D включает функции, которые не используются фреймворком F, и, соответственно, не используются системой S. Изменения в этих функциях внутри D могут вынудить повторно развернуть F и, соответственно, повторно развернуть S. Ошибка в одной из таких функций внутри D способна спровоцировать появление ошибок в F и S.

Пример 2

Рассмотрим интерфейс Shape, который описывает методы для рисования кругов (drawCircle), квадратов (drawSquare) и треугольников (drawTriangle).

```
interface Shape {
    void drawCircle();
    void drawSquare();
    void drawTriangle();
}
```

Листинг 11. Описание интерфейса Shape

В результате классы, реализующие этот интерфейс и представляющие отдельные геометрические фигуры, такие, как круг (Circle), квадрат (Square) и треугольник (Triangle), содержат реализацию всех этих методов.

```
class Circle implements Shape {
    void drawCircle(){ //... }
    void drawSquare(){ //... }
    void drawTriangle(){ //... }
}

class Square implements Shape {
    void drawCircle(){ //... }
    void drawSquare(){ //... }
    void drawTriangle(){ //... }
}

class Triangle implements Shape {
    void drawCircle(){ //... }
    void drawSquare(){ //... }
    void drawTriangle(){ //... }
}
```

Листинг 12. Описание классов, реализующих интерфейс

Таким образом, каждый класс реализует методы, которые не требуются в этом классе.

Предположим, что в интерфейс Shape надо добавить метод drawRectangle(), тогда в каждом классе потребуется реализовать ещё один дополнительный метод, иначе возникнет ошибка.

Принцип разделения интерфейса предостерегает от создания подобных интерфейсов.

```
interface IShape {
    void draw();
}

interface ICircle {
    void drawCircle();
}
```

```
interface ISquare {
    void drawSquare();
}

interface ITriangle {
    void drawTriangle();
}

class Circle implements ICircle{
    void drawCircle(){
    //...
    }
}
class Square implements ISquare {
    void drawSquare(){
    //...
    }
}

class Triangle implements ITriangle {
    void drawTriangle(){
    //...
    }
}

class Shape implements IShape {
    void draw(){
    //...
    }
}
```

Листинг 13. Применение принципа разделения интерфейсов

ISP применяется:

- когда требуются абстрактные классы для повышения гибкости и развития архитектуры;
- можно заранее продумать интерфейс, а затем спроектировать класс, который его имплементирует (реализует).

Не используйте ISP:

- когда добавление метода в интерфейс заставляет реализовывать его в классах-наследниках, которые к нему не относятся;
- ваши классы редко меняются и содержат мало абстракций.

2.5. DIP: Dependency Inversion Principle. Принцип инверсии зависимости

Принцип инверсии зависимости (Dependency Inversion Principle) утверждает, что наиболее гибкими получаются системы, в которых зависимости в исходном коде направлены на абстракции, а не на конкретные реализации.

1. Высокоуровневые модули не зависят от низкоуровневых.
2. Оба вида модулей зависят от абстракций. Абстракции не зависят от подробностей. Подробности зависят от абстракций.

Объект зависимости — абстракция, а не что-то конкретное.

1. Модули верхних уровней не зависят от модулей нижних уровней. Оба типа модулей зависят от абстракций.
2. Абстракции не зависят от деталей. Детали зависят от абстракций.

Классы высокого уровня реализуют бизнес-правила или логику в системе (приложении).

Низкоуровневые классы занимаются более подробными операциями, другими словами, они могут заниматься записью информации в базу данных или передачей сообщений в операционную систему или службы и т. д.

Говорят, что высокоуровневый класс, имеющий зависимость от классов низкого уровня или какого-либо другого класса и много знающий о других классах, с которыми взаимодействует, тесно связан. Когда класс явно знает о дизайне и реализации другого класса, возникает риск того, что изменения в одном классе нарушат другой класс.

Поэтому мы должны держать эти высокоуровневые и низкоуровневые классы слабосвязанными, насколько можем. Чтобы выполнить это, надо сделать их зависимыми от абстракций, а не друг от друга.

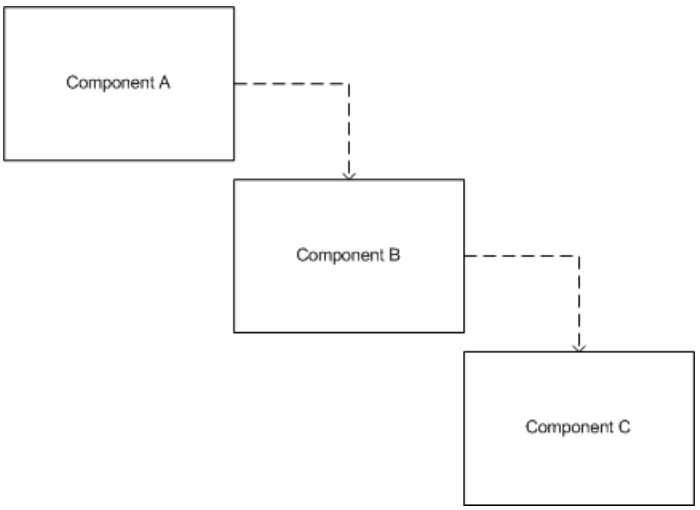
Особенности принципа

1. Изменения реализации модулей низкого уровня не изменяют модули высокого уровня — проблема абстракции или отсутствие инверсии.
2. Языки программирования предоставляют свои классы высокого уровня, такие как String, Date, Time и т. д. И их использование не создаёт никакой проблемы зависимостей. Потому что эти зависимости статичны, в них ничего не меняется.
3. В идеале интерфейс должен быть общим и не от кого не зависеть.
4. Всегда помните, изменяя интерфейс абстракции, меняется реализация нижнего слоя.

💡 Зависимости строятся относительно абстракций, а не деталей.

Модули верхних уровней не зависят от модулей нижних уровней. Если по-простому, то надо делать код так, чтобы он имел как можно меньше зависимостей, и не было круговых зависимостей: например, модуль A зависит от модуля B, а модуль B зависит от модуля A.

На картинке ниже отображения прямая зависимость компонентов.

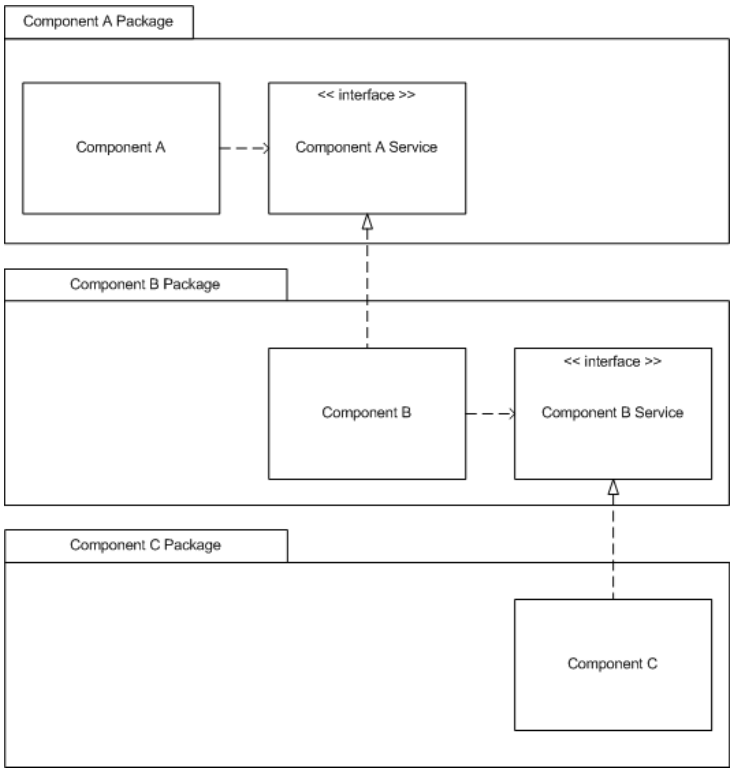


Признаки и проблемы прямой зависимости

1. В классическом программировании классы инициируются через new Class().
2. Подменить реализацию нижнего уровня невозможно.

3. Как результат, уровни зависят друг от друга, логику абстракций сложно удержать в одном месте и не размывать по разным слоям.
4. Невозможно написать хорошие юнит-тесты.

Чтобы решить эти проблемы, применяется инверсия зависимостей. Представим рисунок ниже в виде инверсии зависимостей:



Видно, что все наши зависимости стали инвертируемы. То есть теперь не верхние модули зависят от нижних, а нижние от верхних. На этом рисунке у каждого компонента есть собственный интерфейс, и этот компонент знает о нижнем классе только то, что может знать интерфейс. Модуль нижнего уровня при этом не может общаться с модулем верхнего уровня напрямую, только через его интерфейс.

В любой объектно-ориентированной программе есть зависимости (связи, отношения) между классами. Очевидно, что с ростом количества и силы зависимостей программа становится *менее гибкой*.

Принцип инверсии зависимостей направлен на повышение гибкости программы благодаря *ослаблению связности классов*. В ряде источников говорится, что суть DIP заключается в замене композиции агрегацией — рассмотрим это более детально.

Отношение композиции означает, что объекты одного из классов включают экземпляр другого класса. Такая зависимость считается более слабой, чем наследование, но всё равно очень сильной. Более слабым, а значит, гибким считается **отношение агрегации** — при этом объект-контейнер содержит *ссылку* на вложенный класс.

И композиция, и агрегация выражают **отношение «часть-целое»**. При использовании композиции зависимость между классами считается более сильной, так как мы не можем изменить её во время выполнения программы, но при использовании агрегации для этого достаточно изменить ссылку.

Это один из принципов, откуда выходят многие основополагающие архитектурные идеи.

Пример

```
class Book
{
    public string Text { get; set; }
    public ConsolePrinter Printer { get; set; }

    public void Print()
    {
        Printer.Print(Text);
    }
}

class ConsolePrinter
{
    public void Print(string text)
    {
        Console.WriteLine(text);
    }
}
```

Листинг 14. Описание классов Book и ConsolePrinter

Класс Book, представляющий книгу, использует для печати класс ConsolePrinter.

При подобном определении класс Book зависит от класса ConsolePrinter. Печатать книгу можно только на консоли, используя класс ConsolePrinter. Другие варианты в этом случае исключены. Абстракция печати книги не

отделяется от деталей класса ConsolePrinter. Всё это — нарушение принципа инверсии зависимостей.

```
interface IPrinter
{
    void Print(string text);
}

class Book
{
    public string Text { get; set; }
    public IPrinter Printer { get; set; }

    public Book(IPrinter printer)
    {
        this.Printer = printer;
    }

    public void Print()
    {
        Printer.Print(Text);
    }
}

class ConsolePrinter : IPrinter
{
    public void Print(string text)
    {
        Console.WriteLine("Печать на консоли");
    }
}

class HtmlPrinter : IPrinter
{
    public void Print(string text)
    {
        Console.WriteLine("Печать в html");
    }
}
```

Листинг 15. Применение принципа инверсии зависимости

Теперь абстракция печати книги отделена от конкретных реализаций. В итоге класс Book и класс ConsolePrinter зависят от абстракции IPrinter. Появилась возможность создавать дополнительные низкоуровневые реализации абстракции IPrinter и динамически применять их в программе.

```
Book book = new Book(new ConsolePrinter());
book.Print();
book.Printer = new HtmlPrinter();
book.Print();
```

Листинг 16. Пример вызова метода Print

Применяя этот принцип, одни модули можно легко заменять другими, всего лишь меняя модуль зависимости. И тогда никакие перемены в низкоуровневом модуле не повлияют на высокоуровневый.

Термины, используемые в лекции

Класс — шаблон, на основе которого создаётся конкретный программный объект. Такой шаблон описывает свойства и методы, определяющие поведение объектов этого класса.

Компонент — множество классов и языковых конструкций, объединённых по общему признаку и организованных в соответствии с конкретными правилами и ограничениями.

Модуль — часть реализации программного обеспечения. Обычно модуль состоит из файлов с исходным кодом, но иногда в него входят xml-файлы, текстовые конфигурационные файлы и другие элементы реализации.

Программное обеспечение — совокупность программ и соответствующая документация, позволяющая использовать вычислительную технику для решения различных задач.

Что можно почитать ещё?

1. Основы UML — диаграммы использования (use-case).

2. Элементы графической нотации диаграммы компонентов.

Используемая литература

1. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения.
2. Принципы SOLID, о которых должен знать каждый разработчик.
3. Барбара Лисков.
4. Мультитул.
5. Набор инструментов.