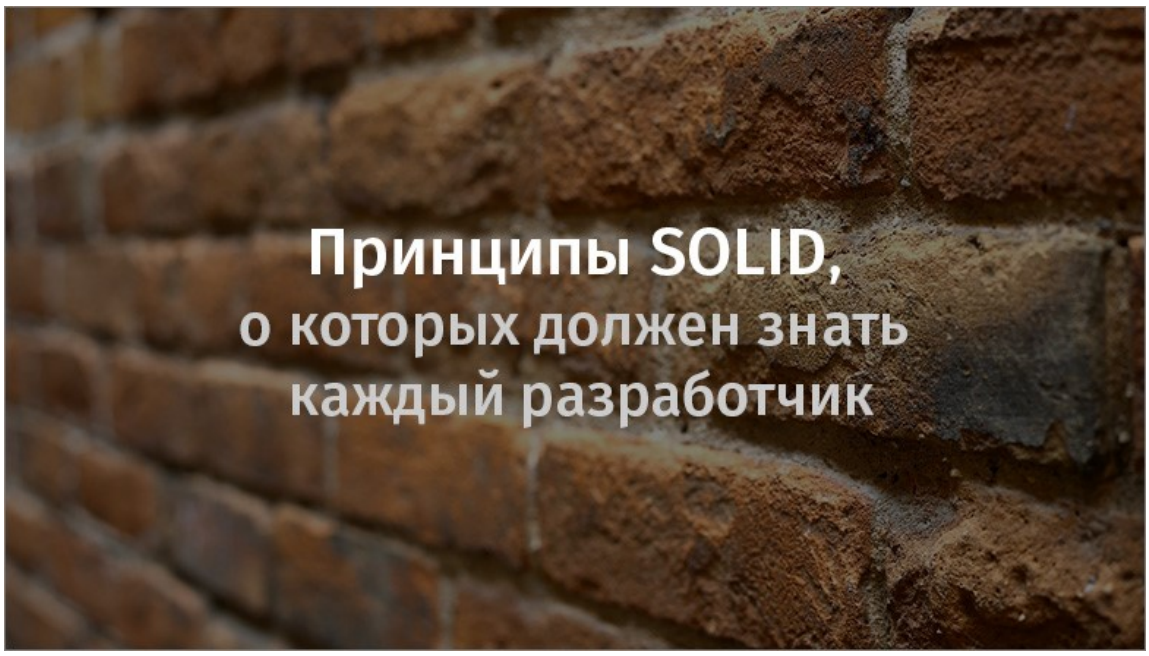
habr.com

Принципы SOLID, о которых должен знать каждый разработчик

ru_vds

17-24 минуты

Объектно-ориентированное программирование принесло в разработку ПО новые подходы к проектированию приложений. В частности, ООП позволило программистам комбинировать сущности, объединённые некоей общей целью или функционалом, в отдельных классах, рассчитанных на решение самостоятельных задач и независимых от других частей приложения. Однако само по себе применение ООП не означает, что разработчик застрахован от возможности создания непонятного, запутанного кода, который тяжело поддерживать. Роберт Мартин, для того, чтобы помочь всем желающим разрабатывать качественные ООП-приложения, разработал пять принципов объектно-ориентированного программирования и проектирования, говоря о которых, с подачи Майкла Фэзерса, используют акроним SOLID.



Материал, перевод которого мы сегодня публикуем, посвящён основам SOLID и предназначен для начинающих разработчиков.

Что такое SOLID?

Вот как расшифровывается акроним SOLID:

- S: Single Responsibility Principle (Принцип единственной ответственности).
- O: Open-Closed Principle (Принцип открытости-закрытости).
- L: Liskov Substitution Principle (Принцип подстановки Барбары Лисков).
- I: Interface Segregation Principle (Принцип разделения интерфейса).
- D: Dependency Inversion Principle (Принцип инверсии зависимостей).

Сейчас мы рассмотрим эти принципы на схематичных примерах. Обратите внимание на то, что главная цель примеров заключается в том, чтобы помочь читателю понять принципы SOLID, узнать, как их применять и как следовать им, проектируя приложения. Автор материала не стремился к тому, чтобы выйти на работающий код, который можно было бы использовать в реальных проектах.

Принцип единственной ответственности

«Одно поручение. Всего одно.» — Локи говорит Скурджу в фильме «Тор: Рагнарёк». Каждый класс должен решать лишь одну задачу.

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.

Обратите внимание на то, что этот принцип применим не только к классам, но и к компонентам программного обеспечения в более широком смысле.

Например, рассмотрим этот код:

```
class Animal {
    constructor(name: string){ }
    getAnimalName() { }
    saveAnimal(a: Animal) { }
}
```

Класс `Animal`, представленный здесь, описывает какое-то животное. Этот класс нарушает принцип единственной ответственности. Как именно нарушается этот принцип?

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

Как такая структура класса может привести к проблемам?

Если изменится порядок работы с хранилищем данных, используемым приложением, то придётся вносить изменения во все классы, работающие с хранилищем. Такая архитектура не отличается гибкостью, изменения одних подсистем затрагивают другие, что напоминает эффект домино.

Для того чтобы привести вышеприведённый код в соответствие с принципом единственной ответственности, создадим ещё один класс, единственной задачей которого является работа с хранилищем, в частности — сохранение в нём объектов класса `Animal`:

```
class Animal {
    constructor(name: string){ }
    getAnimalName() { }
}
class AnimalDB {
    getAnimal(a: Animal) { }
    saveAnimal(a: Animal) { }
}
```

Вот что по этому поводу говорит Стив Фентон: «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».

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

Принцип открытости-закрытости

Программные сущности (классы, модули, функции) должны быть открыты для расширения, но не для модификации.

Продолжим работу над классом `Animal`.

```
class Animal {
    constructor(name: string){ }
    getAnimalName() { }
}
```

Мы хотим перебрать список животных, каждое из которых представлено объектом класса `Animal`, и узнать о том, какие звуки они издают. Представим, что мы решаем эту задачу с помощью функции `AnimalSounds`:

```
//...
const animals: Array<Animal> = [
    new Animal('lion'),
    new Animal('mouse')
];
function AnimalSound(a: Array<Animal>) {
    for(int i = 0; i <= a.length; i++) {
        if(a[i].name == 'lion')
            return 'roar';
        if(a[i].name == 'mouse')
            return 'squeak';
    }
}
AnimalSound(animals);
```

Самая главная проблема такой архитектуры заключается в том, что функция определяет то, какой звук издаёт то или иное животное, анализируя конкретные объекты. Функция `AnimalSound` не соответствует принципу открытости-закрытости, так как, например, при появлении новых видов животных, нам, для того, чтобы с её помощью можно было бы узнавать звуки, издаваемые ими, придётся её изменить.

Добавим в массив новый элемент:

```
//...
const animals: Array<Animal> = [
    new Animal('lion'),
    new Animal('mouse'),
    new Animal('snake')
]
//...
```

После этого нам придётся поменять код функции `AnimalSound`:

```
//...
function AnimalSound(a: Array<Animal>) {
    for(int i = 0; i <= a.length; i++) {
        if(a[i].name == 'lion')
            return 'roar';
        if(a[i].name == 'mouse')
            return 'squeak';
        if(a[i].name == 'snake')
            return 'hiss';
    }
}
AnimalSound(animals);
```

Как видите, при добавлении в массив нового животного придётся дополнять код функции. Пример это очень простой, но если подобная архитектура используется в

реальном проекте, функцию придётся постоянно расширять, добавляя в неё новые выражения `if`.

Как привести функцию `AnimalSound` в соответствие с принципом открытости-закрытости? Например — так:

```
class Animal {
        makeSound();
        //...
}
class Lion extends Animal {
    makeSound() {
        return 'roar';
    }
}
class Squirrel extends Animal {
    makeSound() {
        return 'squeak';
    }
}
class Snake extends Animal {
    makeSound() {
        return 'hiss';
    }
}
//...
function AnimalSound(a: Array<Animal>) {
    for(int i = 0; i <= a.length; i++) {
        a[i].makeSound();
    }
}
AnimalSound(animals);
```

Можно заметить, что у класса `Animal` теперь есть виртуальный метод `makeSound`. При таком подходе нужно, чтобы классы, предназначенные для описания конкретных животных, расширяли бы класс `Animal` и реализовывали бы этот метод.

В результате у каждого класса, описывающего животного, будет собственный метод `makeSound`, а при переборе массива с животными в функции `AnimalSound` достаточно будет вызвать этот метод для каждого элемента массива.

Если теперь добавить в массив объект, описывающий новое животное, функцию `AnimalSound` менять не придётся. Мы привели её в соответствие с принципом открытости-закрытости.

Рассмотрим ещё один пример.

Представим, что у нас есть магазин. Мы даём клиентам скидку в 20%, используя такой класс:

```
class Discount {
    giveDiscount() {
        return this.price * 0.2
    }
}
```

Теперь решено разделить клиентов на две группы. Любимым (`fav`) клиентам даётся скидка в 20%, а VIP-клиентам (`vip`) — удвоенная скидка, то есть — 40%. Для того, чтобы реализовать эту логику, было решено модифицировать класс следующим образом:

```
class Discount {
    giveDiscount() {
        if(this.customer == 'fav') {
```

```
            return this.price * 0.2;
        }
        if(this.customer == 'vip') {
            return this.price * 0.4;
        }
    }
}
```

Такой подход нарушает принцип открытости-закрытости. Как видно, здесь, если нам надо дать некоей группе клиентов особую скидку, приходится добавлять в класс новый код.

Для того чтобы переработать этот код в соответствии с принципом открытости-закрытости, добавим в проект новый класс, расширяющий класс `Discount`. В этом новом классе мы и реализуем новый механизм:

```
class VIPDiscount: Discount {
    getDiscount() {
        return super.getDiscount() * 2;
    }
}
```

Если решено дать скидку в 80% «супер-VIP» клиентам, выглядеть это должно так:

```
class SuperVIPDiscount: VIPDiscount {
    getDiscount() {
        return super.getDiscount() * 2;
    }
}
```

Как видите, тут используется расширение возможностей классов, а не их модификация.

Принцип подстановки Барбары Лисков

Необходимо, чтобы подклассы могли бы служить заменой для своих суперклассов.

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

Рассмотрим применение этого принципа, вернувшись к примеру с классом `Animal`. Напишем функцию, предназначенную для возврата информации о количествах конечностей животного.

```
//...
function AnimalLegCount(a: Array<Animal>) {
    for(int i = 0; i <= a.length; i++) {
        if(typeof a[i] == Lion)
            return LionLegCount(a[i]);
        if(typeof a[i] == Mouse)
            return MouseLegCount(a[i]);
        if(typeof a[i] == Snake)
            return SnakeLegCount(a[i]);
    }
}
AnimalLegCount(animals);
```

Функция нарушает принцип подстановки (и принцип открытости-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. Как результат, при создании нового типа животного функцию придётся переписывать:

```
//...
class Pigeon extends Animal {

}
const animals[]: Array<Animal> = [
    //...,
    new Pigeon();
]
function AnimalLegCount(a: Array<Animal>) {
    for(int i = 0; i <= a.length; i++) {
        if(typeof a[i] == Lion)
            return LionLegCount(a[i]);
        if(typeof a[i] == Mouse)
            return MouseLegCount(a[i]);
         if(typeof a[i] == Snake)
            return SnakeLegCount(a[i]);
        if(typeof a[i] == Pigeon)
            return PigeonLegCount(a[i]);
    }
}
AnimalLegCount(animals);
```

Для того чтобы эта функция не нарушала принцип подстановки, преобразуем её с использованием требований, сформулированных Стивом Фентоном. Они заключаются в том, что методы, принимающие или возвращающие значения с типом некоего суперкласса (`Animal` в нашем случае) должны также принимать и возвращать значения, типами которых являются его подклассы (`Pigeon`).

Вооружившись этими соображениями мы можем переделать функцию `AnimalLegCount`:

```
function AnimalLegCount(a: Array<Animal>) {
    for(let i = 0; i <= a.length; i++) {
        a[i].LegCount();
    }
}
AnimalLegCount(animals);
```

Теперь эта функция не интересуется типами передаваемых ей объектов. Она просто вызывает их методы `LegCount`. Всё, что она знает о типах — это то, что обрабатываемые ей объекты должны принадлежать классу `Animal` или его подклассам.

Теперь в классе `Animal` должен появиться метод `LegCount`:

```
class Animal {
    //...
    LegCount();
}
```

А его подклассам нужно реализовать этот метод:

```
//...
class Lion extends Animal{
    //...
    LegCount() {
        //...
    }
}
//...
```

В результате, например, при обращении к методу `LegCount` для экземпляра класса `Lion` производится вызов метода, реализованного в этом классе, и возвращается именно то, что можно ожидать от вызова подобного метода.

Теперь функции `AnimalLegCount` не нужно знать о том, объект какого именно подкласса класса `Animal` она обрабатывает для того, чтобы узнать сведения о количестве конечностей у животного, представленного этим объектом. Функция просто вызывает метод `LegCount` класса `Animal`, так как подклассы этого класса должны реализовывать этот метод для того, чтобы их можно было бы использовать вместо него, не нарушая правильность работы программы.

Принцип разделения интерфейса

Создавайте узкоспециализированные интерфейсы, предназначенные для конкретного клиента. Клиенты не должны зависеть от интерфейсов, которые они не используют.

Этот принцип направлен на устранение недостатков, связанных с реализацией больших интерфейсов.

Рассмотрим интерфейс `Shape`:

```
interface Shape {
    drawCircle();
    drawSquare();
    drawRectangle();
}
```

Он описывает методы для рисования кругов (`drawCircle`), квадратов (`drawSquare`) и прямоугольников (`drawRectangle`). В результате классы, реализующие этот интерфейс и представляющие отдельные геометрические фигуры, такие, как круг (Circle), квадрат (Square) и прямоугольник (Rectangle), должны содержать реализацию всех этих методов. Выглядит это так:

```
class Circle implements Shape {
    drawCircle(){
        //...
    }
    drawSquare(){
        //...
    }
    drawRectangle(){
        //...
    }
}
class Square implements Shape {
    drawCircle(){
        //...
    }
    drawSquare(){
        //...
    }
    drawRectangle(){
        //...
    }
}
class Rectangle implements Shape {
    drawCircle(){
        //...
    }
    drawSquare(){
        //...
    }
    drawRectangle(){
        //...
    }
```

```
}
```

Странный у нас получился код. Например, класс `Rectangle`, представляющий прямоугольник, реализует методы (`drawCircle` и `drawSquare`), которые ему совершенно не нужны. То же самое можно заметить и при анализе кода двух других классов.

Предположим, мы решим добавить в интерфейс `Shape` ещё один метод, `drawTriangle`, предназначенный для рисования треугольников:

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

Это приведёт к тому, что классам, представляющим конкретные геометрические фигуры, придётся реализовывать ещё и метод `drawTriangle`. В противном случае возникнет ошибка.

Как видно, при таком подходе невозможно создать класс, который реализует метод для вывода круга, но не реализует методы для вывода квадрата, прямоугольника и треугольника. Такие методы можно реализовать так, чтобы при их выводе выбрасывалась бы ошибка, указывающая на то, что подобную операцию выполнить невозможно.

Принцип разделения интерфейса предостерегает нас от создания интерфейсов, подобных `Shape` из нашего примера. Клиенты (у нас это классы `Circle`, `Square` и `Rectangle`) не должны реализовывать методы, которые им не нужно использовать. Кроме того, этот принцип указывает на то, что интерфейс должен решать лишь какую-то одну задачу (в этом он похож на принцип единственной ответственности), поэтому всё, что выходит за рамки этой задачи, должно быть вынесено в другой интерфейс или интерфейсы.

В нашем же случае интерфейс `Shape` решает задачи, для решения которых необходимо создать отдельные интерфейсы. Следуя этой идее, переработаем код, создав отдельные интерфейсы для решения различных узкоспециализированных задач:

```
interface Shape {
    draw();
}
interface ICircle {
    drawCircle();
}
interface ISquare {
    drawSquare();
}
interface IRectangle {
    drawRectangle();
}
interface ITriangle {
    drawTriangle();
}
class Circle implements ICircle {
    drawCircle() {
        //...
    }
}
class Square implements ISquare {
    drawSquare() {
        //...
    }
```

```
}
class Rectangle implements IRectangle {
    drawRectangle() {
        //...
    }
}
class Triangle implements ITriangle {
    drawTriangle() {
        //...
    }
}
class CustomShape implements Shape {
   draw(){
      //...
   }
}
```

Теперь интерфейс `ICircle` используется лишь для рисования кругов, равно как и другие специализированные интерфейсы — для рисования других фигур. Интерфейс `Shape` может применяться в качестве универсального интерфейса.

Принцип инверсии зависимостей

Объектом зависимости должна быть абстракция, а не что-то конкретное.

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

В процессе разработки программного обеспечения существует момент, когда функционал приложения перестаёт помещаться в рамках одного модуля. Когда это происходит, нам приходится решать проблему зависимостей модулей. В результате, например, может оказаться так, что высокоуровневые компоненты зависят от низкоуровневых компонентов.

```
class XMLHttpService extends XMLHttpRequestService {}
class Http {
    constructor(private xmlhttpService: XMLHttpService) { }
    get(url: string , options: any) {
        this.xmlhttpService.request(url,'GET');
    }
    post() {
        this.xmlhttpService.request(url,'POST');
    }
    //...
}
```

Здесь класс `Http` представляет собой высокоуровневый компонент, а `XMLHttpService` — низкоуровневый. Такая архитектура нарушает пункт A принципа инверсии зависимостей: «Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций».

Класс `Http` вынужденно зависит от класса `XMLHttpService`. Если мы решим изменить механизм, используемый классом `Http` для взаимодействия с сетью — скажем, это будет Node.js-сервис или, например, сервис-заглушка, применяемый для целей тестирования, нам придётся отредактировать все экземпляры класса `Http`, изменив соответствующий код. Это нарушает принцип открытости-закрытости.

Класс `Http` не должен знать о том, что именно используется для организации сетевого соединения. Поэтому мы создадим интерфейс `Connection`:

```
interface Connection {
    request(url: string, opts:any);
```

```
}
```

Интерфейс `Connection` содержит описание метода `request` и мы передаём классу `Http` аргумент типа `Connection`:

```
class Http {
    constructor(private httpConnection: Connection) { }
    get(url: string , options: any) {
        this.httpConnection.request(url,'GET');
    }
    post() {
        this.httpConnection.request(url,'POST');
    }
    //...
}
```

Теперь, вне зависимости от того, что именно используется для организации взаимодействия с сетью, класс `Http` может пользоваться тем, что ему передали, не заботясь о том, что скрывается за интерфейсом `Connection`.

Перепишем класс `XMLHttpService` таким образом, чтобы он реализовывал этот интерфейс:

```
class XMLHttpService implements Connection {
    const xhr = new XMLHttpRequest();
    //...
    request(url: string, opts:any) {
        xhr.open();
        xhr.send();
    }
}
```

В результате мы можем создать множество классов, реализующих интерфейс `Connection` и подходящих для использования в классе `Http` для организации обмена данными по сети:

```
class NodeHttpService implements Connection {
    request(url: string, opts:any) {
        //...
    }
}
class MockHttpService implements Connection {
    request(url: string, opts:any) {
        //...
    }
}
```

Как можно заметить, здесь высокоуровневые и низкоуровневые модули зависят от абстракций. Класс `Http` (высокоуровневый модуль) зависит от интерфейса `Connection` (абстракция). Классы `XMLHttpService`, `NodeHttpService` и `MockHttpService` (низкоуровневые модули) также зависят от интерфейса `Connection`.

Кроме того, стоит отметить, что следуя принципу инверсии зависимостей, мы соблюдаем и принцип подстановки Барбары Лисков. А именно, оказывается, что типы `XMLHttpService`, `NodeHttpService` и `MockHttpService` могут служить заменой базовому типу `Connection`.

Итоги

Здесь мы рассмотрели пять принципов SOLID, которых следует придерживаться каждому ООП-разработчику. Поначалу это может оказаться непросто, но если к этому стремиться, подкрепляя желания практикой, данные принципы становятся естественной частью рабочего процесса, что оказывает огромное положительное воздействие на качество приложений и значительно облегчает их поддержку.

Уважаемые читатели! Используете ли вы принципы SOLID в своих проектах?