

Урок 2

Принципы проектирования

Принципы проектирования DRY, KISS, SOLID, GRASP.

Виды паттернов. Язык UML. Диаграммы классов и последовательностей

[Принципы проектирования](#)

[Общие принципы проектирования](#)

[Методы](#)

[Принцип Абстракции](#)

[Принцип DRY](#)

[Принцип KISS](#)

[Принципы SOLID](#)

[Принцип единственной ответственности \(SRP\)](#)

[Принцип открытости/закрытости \(OCP\)](#)

[Принцип подстановки Лисков \(LSP\)](#)

[Принцип разделения интерфейса \(ISP\)](#)

[Принцип инверсии зависимостей \(DIP\)](#)

[Принципы/шаблоны GRASP](#)

[Creator // Создатель](#)

[Information Expert // Информационный эксперт](#)

[Low Coupling // Низкая связанность](#)

[High Cohesion // Высокая связность \(или Высокое сцепление\)](#)

[Controller // Контроллер](#)

[Polymorphism // Полиморфизм](#)

[Protected variations // Устойчивость к изменениям](#)

[Indirection // Посредник](#)

[Pure Fabrication // Чистая синтетика \(Чистая выдумка\)](#)

[Принцип – не догма](#)

[UML](#)

[Диаграмма классов](#)

[Диаграмма последовательностей](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Принципы проектирования

Проектирование программного обеспечения – это процесс, посредством которого разработчик создает спецификацию предназначенного для достижения заявленных целей программного продукта, используя набор примитивных компонентов, имеющих заданные ограничения.

Общие принципы проектирования

- Разработчик должен рассмотреть альтернативные подходы, делая выводы о каждом из них на основе заявленных требований и ресурсов, доступных для выполнения работы.
- Проектирование должно соответствовать аналитической модели. В процессе декомпозиции сложная система делится на части, соответствие которых проектным требованиям легко проверить по нескольким критериям. В тоже время необходимо следить за целостностью системы, насколько заявленные требования удовлетворяются системой в целом.
- Не стоит изобретать велосипед. Система строится с использованием набора шаблонов проектирования, многие из которых, вероятно, встречались ранее. Эти шаблоны всегда должны выбираться как более выгодная альтернатива созданию чего либо с нуля. Ограниченное время и ресурсы должны быть инвестированы в представление действительно новых идей на основе использования существующих шаблонов, когда это применимо.
- Разработчик должен сокращать интеллектуальную дистанцию между программным продуктом и задачей как она существует в реальном мире. Иными словами, структура дизайна программного обеспечения должна, по возможности, имитировать структуру предметной области (см. выше «Проектирование на основе предметной области»).
- Дизайн должен демонстрировать единообразие и целостность. Для достижения этой цели в процессе разработки команда разработчиков должна следовать заранее оговоренным архитектурным принципам, лежащим в основе создаваемого приложения.
- Структура проектируемого продукта должна поддерживать внесение изменений.
- Следует различать проектирование и кодирование (реализацию проектных решений). Уровень абстракции проектных решений выше уровня реализации при кодировании. Проектирование – это стратегия, кодирование – тактика создания продукта.
- Проектные решения требуют независимой оценки в целях устранения концептуальных ошибок. Иногда разработчик сосредотачивается на мелочах и, как говорится, за деревьями леса не видит. Сначала проектная группа должна убедиться в корректности основных концептуальных элементов дизайна, и уже потом – переходить к проработке особенностей реализации.

Методы

- Абстракция.
- Декомпозиция.
- Модульность.
- Проектирование на основе предметной области (модель, основа коммуникаций с заинтересованными сторонами).

- Тестирование (избавляет от многих архитектурных проблем, требуя создавать модульный слабосвязанный код еще на стадии разработки).

Принцип Абстракции

«Каждая существенная область функциональности в программе должна быть реализована всего в одном месте программного кода. Если различные фрагменты кода реализуют аналогичную функциональность, то, как правило, имеет смысл слить их в один фрагмент, абстрагируя (abstracting out) различающиеся части», – Б.Пирс [1].

«Абстракция выделяет существенные характеристики объекта, которые отличают его от всех других видов объектов и таким образом обеспечивают четко определенные концептуальные границы относительно взгляда наблюдателя», – Г. Буч [2].

Абстракция в объектно-ориентированном программировании – это придание объекту характеристик, четко определяющих его концептуальные границы, отличая от всех других объектов. Основная идея состоит в том, чтобы отделить способ использования составных объектов данных от деталей их реализации в виде более простых объектов.

Рассмотрим пример. Предприятие оптовой торговли имеет дело с покупателями и продавцами, причем покупатель никогда не выступает в роли продавца, а продавец – в роли покупателя (эдакий типичный перепродавец-спекулянт). Из этих двух типов следует выделить Абстракцию-контрагент и Абстракцию-вид контрагента – это сделает каждую новую полученную абстракцию более цельной и универсальной.

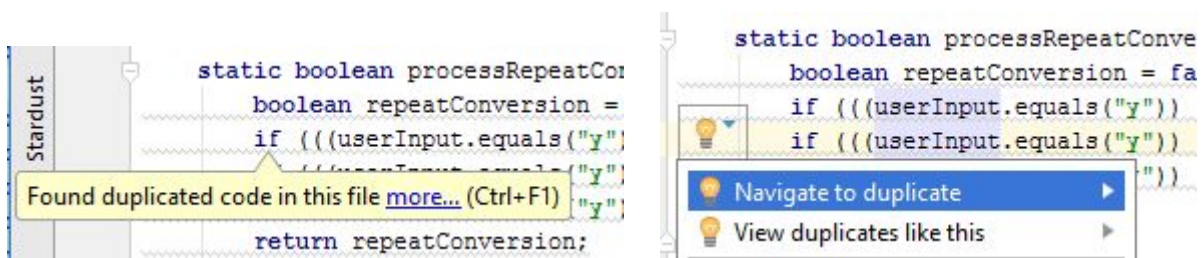
«Любая проблема проектирования может быть решена введением дополнительного абстрактного слоя, за исключением проблемы слишком большого количества дополнительных абстрактных слоев», – неизвестный автор.

Принцип DRY

Принцип DRY (Don't Repeat Yourself) – «не повторяйся». Впервые он был официально сформирован в книге Эндрю Ханта и Дэвида Томаса «Программист-прагматик» и звучит так: «каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы».

Если вы видите в вашем коде схожие места, значит, это место может быть выделено в отдельную функцию, класс, пакет и так далее. В дальнейшем при изменении внешних требований достаточно поменять код в одном месте, дабы измененный функционал стал доступен во всех его использующих частях программы.

Стоит отметить, что современные IDE, такие как IntelliJ IDEA, обладают достаточно развитым интеллектом, чтобы подсказывать разработчику места дублирования кода. Кроме того, имеется множество методов удобного рефакторинга кода с целью выделения части кода в отдельный метод, класс и так далее.



Принцип KISS

Принцип KISS (Keep It Simple Stupid) – «сделай проще». Этот принцип запрещает использование более сложных средств, чем те, что необходимы для решения задачи. Принцип декларирует простоту системы в качестве основной цели и/или ценности. Наиболее близким предтече KISS, скорее всего, является широко известный принцип Бритвы Оккама: «не следует множить сущее без необходимости».

Принципы SOLID

Пять принципов объектно-ориентированного проектирования, предложенные Робертом Мартином в начале 2000-х годов в [3]. S.O.L.I.D – акроним от сокращенных названий принципов ООД:

Инициал	Принцип ¹	Название, понятие
S	SRP	Принцип единственной ответственности (The Single Responsibility Principle): «существует лишь одна причина, приводящая к появлению класса».
O	OCP	Принцип открытости/закрытости (The Open Closed Principle): «программные сущности... должны быть открыты для расширения, но закрыты для модификации».
L	LSP	Принцип подстановки Лисков (The Liskov Substitution Principle): «объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы».
I	ISP	Принцип разделения интерфейса (The Interface Segregation Principle): «много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения».
D	DIP	Принцип инверсии зависимостей (The Dependency Inversion Principle): «зависимость на абстракциях. Нет зависимости на что-то конкретное».

Рассмотрим их более подробно.

Принцип единственной ответственности (SRP)

Принцип единой ответственности гласит, что каждый модуль или класс должен нести ответственность только за одну часть функциональности, предоставляемой программным обеспечением, и эта ответственность должна быть полностью инкапсулирована классом.

Существует лишь одна причина, приводящая к изменению класса [3] (стр.165).

Ответственность класса – это стержень, вокруг которого выстраиваются его методы. Если класс отвечает за две (или более) ответственностей, то и стержней становится более одного. При изменении одной из ответственностей такой класс уже может не соответствовать другой ответственности. Наглядная аналогия – слуга двух господ Фигаро.

Пример. Есть объект Order (заказ), включающий в себя следующие методы:

```
class Order {  
    public Items getItems();  
    public Money getTotal();  
    public bool validate();  
    public void save();  
    public void load();  
}
```

Здесь мы видим смешение бизнес-логики и персистентности (сохранения состояния). При изменении методов, отвечающих за персистентность, очень легко «задеть» бизнес-логику, что не желательно, поскольку клиентские классы (использующие класс Order) ни в коей мере не ожидают никаких изменений. Если не отделить логику от сохранения состояния, то приложение становится хрупким – изменения в одной его части нарушают работу других концептуально независимых частей приложения. Правильным решением будет отделить логику от сохранения:

```
class Order {  
    public Items getItems() {}  
    public Money getTotal() {}  
    public bool validate() {}  
}
```

и

```
class OrderRepository {  
    public void save() {}  
    public void load() {}  
}
```

Однако не следует чрезмерно переусердствовать в разделении обязанностей класса, так как можно впасть в другую крайность, при которой логика класса размывается между несколькими мелкими объектами.

Принцип открытости/закрытости (ОСР)

Широко известный от Р. Мартина [5] принцип открытости/закрытости имеет два варианта:

Программные сущности (классы, модули, функции и т.п.) должны быть открытыми для расширения, но закрытыми для модификации.

Первоначальная идея принадлежит Бертрану Мейеру [6].

Модули должны быть одновременно и открытыми и закрытыми.

Модуль будет считаться открытым, если он по-прежнему доступен для расширения. Например, должно быть возможно добавить поля к содержащимся в нем структурам данных или новые элементы к набору функций, которые он выполняет.

Модуль будет считаться закрытым, если он доступен для использования другими модулями. Это означает, что модулю (его интерфейсу в смысле скрытия информации)

дано четкое, окончательное описание. Модуль можно компилировать, сохранить в библиотеке и сделать доступным для использования другими модулями.

Рассмотрим этот принцип подробно на классическом примере с фигурами. Допустим, мы проектируем САПР, которая имеет дело с различными фигурами. Каждая фигура представлена своим классом и требуется некоторая функция для отрисовки набора фигур в пользовательском интерфейсе.

```
// Фигура
class Shape {
}
```

```
// Круг
class Circle extends Shape {
}
```

```
// Треугольник
class Triangle extends Shape {
}
```

```
// САПР
public class CAD {
    public void drawAll(Shape[] shapes) {
        for (Shape shape : shapes) {
            // выбор поведения в зависимости от типа входного объекта
            if (shape instanceof Circle) {
                drawCircle((Circle) shape);
            }
            else if (shape instanceof Triangle) {
                drawTriangle((Triangle) shape);
            }
        }
    }
}

// рисуем круг
public void drawCircle(Circle circle) {}

// рисуем треугольник
public void drawTriangle(Triangle triangle) {}
}
```

В чем тут проблема? Чтобы при необходимости добавить, допустим, квадрат (Square), нам придется: во-первых, изменить поведение метода CAD.drawAll при выборе поведения и, во-вторых, реализовать отрисовку квадрата в отдельном методе этого класса. Говоря строго: класс CAD не соответствует принципу закрытости/открытости, т.к. не закрыт от расширения типов наследников Shape.

Что мы можем сделать? Надо абстрагировать отрисовку фигур. Сделать класс Shape абстрактным объявив абстрактный метод draw();

```
// Абстрактная фигура
abstract class Shape {
    abstract void draw();
}
```

```
// Круг
class Circle extends Shape {
    @Override
    public void draw() {}
}
```

```
// Треугольник
class Triangle extends Shape {
    @Override
    public void draw() {}
}
```

```
// САПР
public class CAD {
    public void drawAll(Shape[] shapes) {
        for (Shape shape : shapes) {
            shape.draw();
        }
    }
}
```

Новое требование к системе: круги должно рисовать первыми, треугольники после кругов. Решение «в лоб» приведет нас туда, откуда мы только что ушли: зависимости класса CAD от наследников Shape.

```
public class CAD {
    // решение упорядочиванием, уже лучше
    public void drawAllOrdered(Shape[] shapes) {
        for (Shape shape : shapes) {
            if(shape instanceof Circle)
                shape.draw();
        }
        for (Shape shape : shapes) {
            if(shape instanceof Triangle)
                shape.draw();
        }
    }
}
```

Дополнительный уровень Абстракции, в данном случае абстрагирующий порядок отрисовки, помогает, но помещать его в иерархию Shape будет опрометчиво, ибо мы нарушим закрытость

наследников Shape относительно друг друга и вдобавок нарушаем принцип SRP относительно ответственности потомков Shape, заставляя их отвечать за порядок сортировки.

```
// Круг
class Circle extends Shape {
    @Override
    int compareOrder(Shape shape) {
        // нарушение закрытости относительно других потомков Shape
        return shape instanceof Circle ? 1 : -1;
    }
    ....
}
```

Идея правильная, однако место для абстрагирования выбрано неудачно. Давайте вынесем абстракцию порядка за рамки иерархии Shape, для чего объявим класс, отвечающий только за порядок сортировки.

```
// САПР
public class CAD {
    // упорядоченная отрисовка
    public void drawAllOrdered(Shape[] shapes) {
        // копируем исходный массив, т.к. порядок элементов будет изменен
        Shape[] clone = shapes.clone();
        // сортировка, используем готовую внешнюю абстракцию компаратора
        // java.util.Arrays.Comparator, имплементированную в классе ShapeOrderComparator
        java.util.Arrays.sort(clone, new ShapeOrderComparator());
        drawAll(clone);
    }
    ...
}
```

```
// компаратор фигур относительно порядка отрисовки
public class ShapeOrderComparator implements Comparator<Shape> {
    @Override
    public int compare(Shape shape1, Shape shape2) {
        // реализация опущена
        return ...
    }
    ...
}
```

Соблюдение принципа открытости/закрытости достигается применением дополнительного уровня абстракции и размещением его в открытой для расширения части модуля. Используя абстракции, невозможно создать такую систему, которая будет удовлетворять все точки зрения. Мастерство разработчика заключается в способности предугадать наиболее возможные из них и заложить эти абстракции в основу расширения системы.

Принцип подстановки Лисков (LSP)

Принцип подстановки Лисков (предложен Барбарой Лисков в 1987 году):

Пусть $q(x)$ является свойством, верным относительно объектов x некоторого типа T . Тогда $q(y)$ также должно быть верным для объектов y типа S , где S является подтипом типа T .

Р. Мартин в [5] формулирует его так:

Должна быть возможность вместо базового типа подставить любой его подтип.

Рассмотрим классический пример. Допустим в системе САПР имеется класс Прямоугольников:

```
// Прямоугольник
public class Rectangle {
    private int width;
    private int height;

    public int getWidth() {return width;}

    public void setWidth(int width) {this.width = width;}

    public int getHeight() {return height;}

    public void setHeight(int height) { this.height = height; }

    public int getArea() { return width * height;}
}
```

Через какое-то время по разным причинам потребовался класс, представляющий квадраты, т.к. математически квадрат — это частный случай прямоугольника, то, наследуя прямоугольник, мы получаем квадрат. Так как у квадрата все стороны равны, мы перегружаем методы присвоения длин сторон.

```
// Квадрат
public class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public void setHeight(int height) {
        super.setHeight(height);
        super.setWidth(height);
    }
}
```

А теперь попробуем проверить реализацию Square на соответствие принципу LSP:

```
class SquareTest {
    @Test
    void testArea() {
        Rectangle rectangle = new Square();
        // нарушение принципа LSP, мы не ожидаем тут, что и высота ТОЖЕ изменится
        rectangle.setWidth(5);
        rectangle.setHeight(4);
        // тест провален, актуальное значение 16
        assert rectangle.getArea() == 20;
    }
}
```

Тест провален, ибо реализация Square нарушает принцип LSP в методах установки длин сторон, ибо совершенно неожиданно для прямоугольника (коим тут является квадрат) квадрат модифицирует соседнее значение размера фигуры.

Варианты решения данной проблемы:

1. Проектирование по контракту, которое включает в себя предусловия и постусловия.
 - a. Предусловия (то, что должно быть выполнено вызывающей стороной перед вызовом метода) не могут быть усилены в классе Наследнике.
 - b. Постусловия (то, что гарантируется вызываемым методом) не могут быть ослаблены в классе Наследнике.

В данном примере квадрат нарушает постусловия для прямоугольника, модифицируя одновременно два поля базового класса.

К сожалению, Java не поддерживает программирование по контракту из коробки, но вы можете воспользоваться сторонней библиотекой, например, этой <https://opensource.googleblog.com/2011/02/contracts-for-java.html>, вдохновленной языком Eiffel [9].

2. Использование Immutable (неизменных) классов, например, так:

```
// Прямоугольник // неизменяемый
public class Rectangle {
    private final int height;
    private final int width;

    public Rectangle(int h, int w ) {
        height = h;
        width = w;
    }

    public int getArea() { return width * height;}
}
```

```
// Квадрат неизменяемый
public class Square extends Rectangle {
    public Square(int side) {
        super(side, side);
    }
}
```

В таком случае принцип LSP будет соблюден, поскольку отсутствует предусловие по зависимости/независимости изменения длин прямоугольника и квадрата, ибо они назначаются единожды при инициализации.

3. Наследовать оба класса от какой-то более общей абстракции, развивая их основное отличие: все стороны Квадрата равны, что неверно для прямоугольника.

Принцип разделения интерфейса (ISP)

Формулировка:

Клиенты не должны вынужденно зависеть от методов, которыми не пользуются.

Пример. Вернемся к нашим фигурам. Предположим, теперь нам надо выводить фигуры на печать. Естественно, мы добавим в абстрактный класс фигуры метод `plot()`, который конкретные фигуры будут реализовывать.

```
// Фигура
interface Shape {
    void draw();
    void plot();
}
```

```
// Круг
class Circle implements Shape {
    @Override
    public void draw() { /* нарисовать */ }

    @Override
    public void plot() { /* напечатать */ }
}
```

Все бы хорошо, но рано или поздно у нас появятся фигуры, которые не выводятся на печать, но присутствуют на экране: например, оси, направляющие и так далее. Естественно, их реализация будет пустой:

```
// направляющая, непечатная
class GuideLine implements Shape {
    @Override
    public void draw() { /* нарисовать */ }

    // пустой метод, вынужденная реализация
    @Override
    public void plot() {}
}
```

Следовательно, в соответствии с принципом ISP, из интерфейса фигуры методы, относящиеся к печати, должны быть выделены в отдельный интерфейс, который будут реализовывать только те типы фигур, которым это действительно необходимо:

```
// Фигура
interface Shape {
    void draw();
}
```

```
// Нечто печатаемое
interface Plotable {
    void plot();
}
```

```
// Круг, печатаемый, реализует интерфейс печатаемого
class Circle implements Shape, Plotable {
    @Override
    public void draw() { /* нарисовать */ }
    @Override
    public void plot() { /* напечатать */ }
}
```

```
// Направляющая, просто фигура, непечатная
class GuideLine implements Shape {
    @Override
    public void draw() { /* нарисовать */ }
}
```

Глядя только на класс, мы можем судить лишь о том, соблюдает или не соблюдает класс принцип SRP. Относительно принципа ISP мы такого сказать не можем, ибо нам необходимо знать контекст использования класса, как его используют классы Клиенты. Если классы Клиенты интересуются разными аспектами анализируемого класса, то стоит подумать о более тонкой сегрегации его интерфейсов.

Принцип инверсии зависимостей (DIP)

Представим себе некое достаточно крупное приложение с множеством классов. При естественном порядке построения сперва мы опишем некие базовые низкоуровневые сущности, затем перейдем к сущностям более высокого порядка, которые будут определены в терминах низкоуровневых сущностей. Этот путь кажется логичным, но приводит к «жесткому» дизайну. Одни компоненты приложения «впадают» в сильную зависимость от других, что не есть хорошо.

Формулировка:

*Модули верхних уровней не должны зависеть от модулей нижних уровней.
Оба типа модулей должны зависеть от абстракций.*

*Абстракции не должны зависеть от деталей.
Детали должны зависеть от абстракций.*

Пример. Торговое приложение B2B. Оформляем заказы.

```
// Некий просто товар
public class SimpleItem {
    public Money getPrice() {...};
}
```

```
// Заказ
public class Order {
    // добавить товар в заказ
    public void add(SimpleItem simpleItem) {
        total.add(simpleItem.getPrice());
    }
    // итог
    Money total;
}
```

В какой-то момент бизнес-правила меняются и теперь сущность товара по неким причинам представляет другой класс.

```
// Превосходный товар
public class PerfectItem {
    public Money getPrice() {...};
}
```

Проблема: мы не можем добавлять экземпляры PerfectItem в имеющийся класс Order, ибо последний сильно зависит от старого класса SimpleItem. Мы нарушили принцип инверсии зависимостей – заказ зависит от товара. Перепроектируем решение в соответствии с принципом. Выделим абстракцию в интерфейс и уже от абстракции (интерфейса) будут зависеть классы обоих уровней.

```
// Абстракция, интерфейс товара
public interface ItemInterface {
    public Money getPrice();
}
```

```
// Превосходный товар имплементирует абстрактный товар
public class PerfectItem implements ItemInterface {
    @Override
    public Money getPrice() {
        return ...
    }
}
```

```
// Заказ
public class Order {
    // добавить абстрактный товар в заказ, детали реализации товара нас здесь не интересуют
    public void add(ItemInterface item) {
        total.add(item.getPrice());
    }
    ....
}
```

Принцип инверсии зависимостей – это фундаментальный низкоуровневый механизм, лежащий в основе многих преимуществ, которые обещают объектно-ориентированные технологии. Его надлежащее применение необходимо для создания повторно используемых каркасов [5].

Принципы/шаблоны GRASP

Принципы/шаблоны GRASP были предложены Крейгом Ларманом в 1998 году в его книге [7]. Относительно двойственности названия сам автор отсылает к «отцам-основателям» [8]:

«Шаблон одного разработчика – это простейший строительный блок другого»

Принципы/шаблоны GRASP представляют с одной стороны другой относительно SOLID взгляд на проблемы проектирования программного обеспечения, в то же время их автор приходит к выводам аналогичным SOLID-принципам. Рассмотрим их более подробно.

Creator // Создатель

Сущности создаются, меняются, уничтожаются. Если с последними двумя действиями все понятно – у вас есть то, что требуется модифицировать или удалить, то с созданием все обстоит менее понятно. А именно кто (что) должен создать сущность? Да, в управляемых языках мы можем воспользоваться new, а дальше? Очевидно, вас не прельщает перспектива, например, того, что одна строка заказа окажется в двух разных заказах. Итак, мы пришли к необходимости назначения ответственности за создание сущностей класса некоторому вполне конкретному классу.

Согласно шаблону Creator, этот класс C должен удовлетворять одному из критериев:

- Класс C содержит или агрегирует объекты A.
- Класс C записывает экземпляры объектов A.
- Класс C активно использует объекты A.
- Класс B обладает данными инициализации, которые будут передаваться объектам A при их создании.

Например, согласно первому критерию, заказ (Order) является хорошим претендентом на роль Создателя объектов типа строки заказа (OrderItem).

Information Expert // Информационный эксперт

Как распределить ответственность между классами системы?

Ответственность должна быть назначена тому, кто владеет максимумом необходимой информации для исполнения – Информационному эксперту.

Например, заказ (Order) состоит из строк заказа (OrderItem), содержащих ссылку на товар (Item) и количество (поле qty). Согласно этому принципу, обязанность по вычислению общей стоимости заказа следует возложить на класс заказа (Order), так как он обладает всей полнотой информации о составе заказа – строки OrderItem. При вычислении суммы по всем строкам заказа объект класса Order вынужден обратиться к Информационному эксперту стоимости строки заказа – к объектам класса OrderItem, а те, в свою очередь, к объектам товаров Item, которые являются информационными экспертами относительно цены товара.

Low Coupling // Низкая связанность

Степень связанности – это мера, определяющая, насколько, как сильно один элемент связан с другими элементами, либо каким количеством данных о других элементах он обладает. Использование класса с высокой связанностью приводит к следующим проблемам:

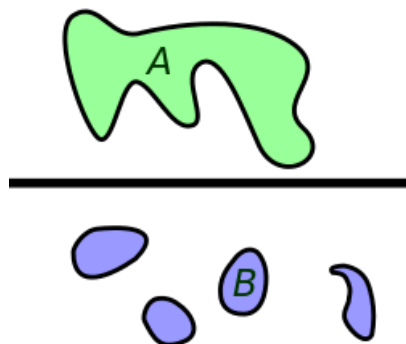
- Нестабильность связанных классов приводит к нестабильности в данном классе.
- Усложняется повторное использование класса с высокой связанностью.

Другой крайностью является нулевая связанность классов, что также нежелательно, т.к. система распадается на множество независимых классов со сложной реализацией. Система, возможно, и будет устойчива к изменениям, но это произойдет ценой повышения сложности каждого из составляющих ее компонентов.

Таким образом, необходимо найти некоторый баланс степени связанности, позволяющий выполнять функции системы классам совместно.

High Cohesion // Высокая связность (или Высокое зацепление)

Связность – это мера силы взаимосвязанности обязанностей класса. Обязанности класса с высокой степенью связности связаны между собой. Так и наоборот: класс со множеством разнообразных не связанных между собой обязанностей имеет слабую связность.



Множество А – связано, множество В – несвязно.

Источник https://ru.wikipedia.org/wiki/Связное_пространство

Применение классов с низкой связностью чревато следующими последствиями:

- Трудности понимания.
- Сложности при повторном использовании.
- Сложности поддержки.
- Низкая надежность.

Controller // Контроллер

Контролер – это объект, отвечающий на запросы пользователя (пользователей) системы, но не являющийся интерфейсным объектом. Контроллер имеет представление о системе в целом и делегирует обязанности по выполнению запросов конкретным исполнителям.

Типичный широко распространенный пример – шаблон MVC, в котором Контроллер, получая запросы пользователя, делегирует их тем или иным методам манипулирования моделью.

Polymorphism // Полиморфизм

При реализации похожих, но различных по особенностям реализации действий, следует применять полиморфные свойства наследования классов, а не условную логику проверки типа классов. Выделение общего интерфейса позволяет однотипно обрабатывать объекты с различной реализацией.

Protected variations // Устойчивость к изменениям

Шаблон Protected variations позволяет защитить одни объекты от изменений в других, связанных с ним, объектах. Это достигается путем обертки различных реализаций в один неизменный интерфейс, таким образом с одной стороны мы получаем стабильность интерфейса и свободу в его реализации.

Indirection // Посредник

Для уменьшения связанности между элементами системы используется промежуточный объект – Посредник. Примером реализации данного шаблона является шаблон Controller, рассмотренный выше. Controller «развязывает» объекты пользовательского интерфейса и модель. Другой пример использования шаблона Посредник – шаблон Адаптер. Объект Адаптер защищает класс Клиент от изменений во внешней системе.

Pure Fabrication // Чистая синтетика (Чистая выдумка)

Классы в модели предметной области отражают сущности реального мира, но зачастую построение системы, удовлетворяющей принципам Low Coupling/High Cohesion только на основе сущностей реального мира, невозможно – в таком случае проектировщик использует объекты Чистая выдумка, не имеющие прямого отображения в реальном мире, но позволяющие добиться соблюдения принципов Low Coupling/High Cohesion при реализации системы. Например, мы имеем объект заказ (Order), который надо каким-то образом сохранить, например, в БД. Включение логики сохранения заказа в БД в объект заказа неминуемо снизит его уровень связности, т.к. объект будет выполнять уже две мало связанные между собой обязанности. В тоже время (1) сохранение в БД потребуется для множества различных типов объектов и (2), в целом, это достаточно рутинная операция. Таким образом мы подошли к созданию некоторого объекта, отвечающего за сохранение объектов сущностей предметной области в БД. Этот объект и является «чистой выдумкой». Одной из широко известных реализацией этой задачи является ORM.

Принцип – не догма

Принципы, рассмотренные выше, называются «принципами», потому как представляют некоторые убеждения относительно того, как стоит или не стоит проектировать программное обеспечение.

Вы можете им следовать, получая преимущества, о которых заботились их создатели, но, возможно, в ваших конкретных условиях слепое следование какому-то принципу (или нескольким) не будет оправданно.

Как разработчик, вы можете (и это будет полезно) рассмотреть при решении вопросов проектирования несколько вариантов, каждый которых имеет свои плюсы и минусы. Какое именно решение возобладает, во многом зависит от конкретно ваших условий и ваших целей.

UML

UML (Unified Modeling Language – унифицированный язык моделирования) – язык графического описания для объектного моделирования в области разработки программного обеспечения, моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

Разработан на рубеже 1990-х–2000-х годов Гради Буч и Джеймсом Рамбо в недрах в компании Rational Software. Стимулом к созданию языка UML послужило желание стандартизировать (унифицировать) разрозненные схемы нотации и подходы к проектированию программного обеспечения.

UML позволяет описывать сущности графически на диаграммах, используя унифицированные соглашения об обозначении понятий (сущностей, классов, процессов) и взаимосвязей между ними.

Диаграмма классов

Диаграмма классов (Static Structure diagram) – это диаграмма, описывающая классы, их атрибуты, методы и взаимосвязи между ними [9].

Класс изображается в виде прямоугольника, содержащего три части:

Имя_класса	Имя класса пишем с большой буквы по центру в заголовке, имя Абстрактного класса пишем курсивом.
- атрибут	Атрибуты (поля, fields) класса пишем в средней части с маленькой буквы, выравниваем по левой грани.
+ метод	Методы (methods) пишем в нижней части с маленькой буквы, выравниваем по левой грани.

Видимость атрибутов (полей) класса обозначается знаком слева от имени атрибута.

Обозначение	Видимость
+	Публичный (public)
-	Приватный (private)
#	Защищенный (protected)
/	Производный (derived)
~	Пакет (package)

Взаимосвязи между классами отображаются с помощью стрелок:

Обозначение	Название	Описание
	Зависимость	Отношение между классами, при котором один зависит от другого, но не наоборот.
	Наследование	Подтип наследует данные и поведение базового типа (супертипа).
	Реализация	Сущность реализует (implement) поведение (но не данные), заданное абстрактным типом, чаще всего интерфейсом.
	Ассоциация	Показывает, что объекты одного класса связаны с объектами другого класса таким образом, что можно перемещаться от объектов одного класса к другому.
	Агрегация	Вид ассоциации при отношении между целым и его частями. Может включать более двух классов – контейнер и содержимое.
	Композиция	Более строгий вариант агрегации, имеет жесткую зависимость времени существования экземпляров класса контейнера от экземпляров содержащихся в нем классов. Если контейнер будет уничтожен, то все его содержимое будет также уничтожено.

Кратность (мощность) отношений, если применимо.

Обозначение	Описание
0..1	Ноль или один экземпляр.
1	Обязательно один экземпляр.
0..* или *	Ноль или более экземпляров.
1..*	Один или более экземпляров.

Пример диаграммы классов системы обработки заказов.

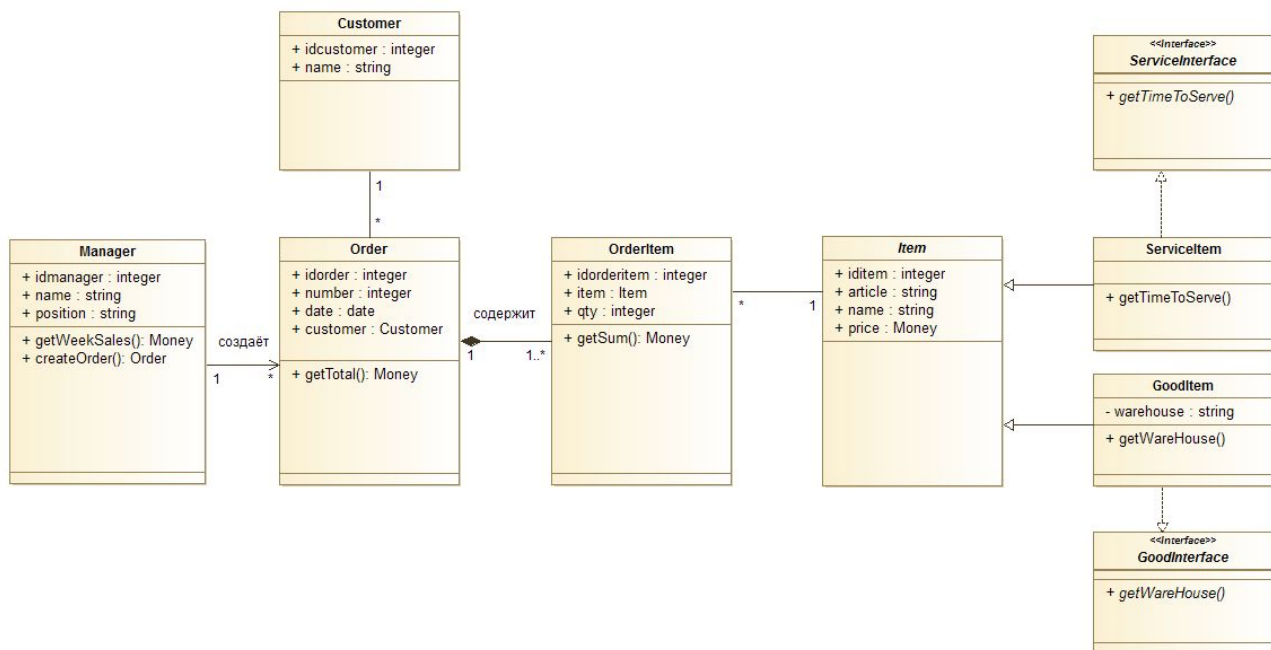
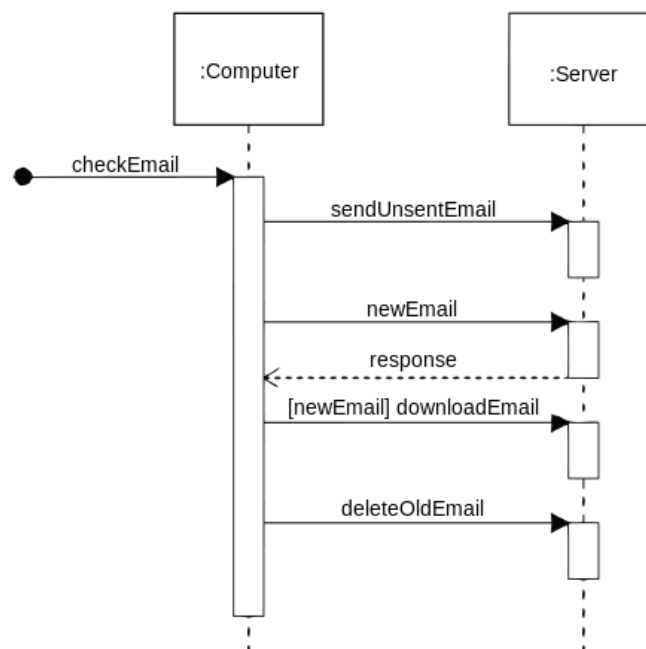


Диаграмма последовательностей




Диаграмма последовательности (Sequence Diagram) – диаграмма, на которой для некоторого набора объектов на единой временной оси показан жизненный цикл какого-либо объекта или взаимодействие объектов в рамках какого-либо определенного прецедента [10].



Источник: https://en.wikipedia.org/wiki/Sequence_diagram

Вертикальные пунктирные линии, «линии жизни» (lifeline), отображают течение времени объекта, имя класса которого указано в прямоугольнике в начале линии (вверху), время для всех объектов течет одинаково. Прямоугольники на «линии жизни» отображают деятельность объекта по выполнению определенной функции.

Горизонтальные стрелки отображают передачу сообщений между объектами.

Обозначение	Название	Описание
	Синхронное сообщение	При подаче синхронного сообщения отправитель теряет возможность выполнять действия до момента получения ответного сообщения. Синхронный вызов.
	Асинхронное сообщение	Если объект посылает асинхронное сообщение, то он продолжает выполнять свою работу, не ожидая ответного сообщения. Асинхронный вызов.
	Ответное сообщение	Ответ на синхронное сообщение.

Пример. Создание заказа, добавление позиции.

Практическое задание

1. Для итогового проекта нужно составить полную диаграмму классов. Какие классы будут присутствовать в системе? Какие атрибуты и методы они должны иметь? Какие из методов и атрибутов следует сделать публичными, а какие – приватными? Где можно применить наследование классов?
2. Выделить три основных, на ваш взгляд, процесса, происходящих в проектируемой системе. Изобразить для этих процессов диаграммы последовательностей.

Дополнительные материалы

1. «SOLID is OOP for Dummies» – <http://www.yegor256.com/2017/03/28/solid.html>
2. http://sergeyteplyakov.blogspot.ru/2013/10/articles.html#principles_and_patterns

Используемая литература

1. Типы в языках программирования, Б.Пирс, М.: «Лямбда пресс», «Добросвет», 2011.
2. Объектно-ориентированный анализ и проектирование с примерами приложений, Г.Буч, и др. М: Вильямс, 2008.
3. <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
4. Быстрая разработка программ, Принципы, Примеры, Практика, Р. Мартин и др. М: Вильямс, 2004.
5. Принципы, паттерны и методики гибкой разработки на языке C#, Р. Мартин, СПб.: Символ-Плюс, 2011.

6. Объектно-ориентированное конструирование программных систем, Бертран Мейер. М: Русская Редакция, 2005.
7. Применение UML 2.0 и шаблонов проектирования, К. Ларман,. М.: “Вильямс”, 2013.
8. Design Patterns, Gamma, E., Helm, R., Johnson, R., and Vlissides, MA.: Addison-Wesley, 1995.
9. https://en.wikipedia.org/wiki/Class_diagram
10. https://en.wikipedia.org/wiki/Sequence_diagram
11. <https://ru.wikipedia.org/wiki/Eiffel>
12. Programming stuff, блог Сергея Теплякова
http://sergeyteplyakov.blogspot.ru/2013/10/articles.html#principles_and_patterns
13. <https://habrahabr.ru/post/169487/>