Принципы проектирования

Принципы проектирования GRASP

General Responsibility Assignment Software Patterns
or
General Responsibility Assignment Software Principles

GRASP - шаблоны проектирования, используемые для решения общих задач по назначению обязанностей классам и объектам

Сформулированы Крейгом Ларманом в книге «Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development»

9 принципов

- Information expert
- Low coupling
- High cohesion
- Controller
- Creator

- Polymorphism
- Protected variation
- Pure fabrication
- Indirection

Information Expert (информационный эксперт)

Базовый принцип разделения ответственности

Ответственность на себя должен брать тот, кто обладает максимумом информации для исполнения.

Если класс – «эксперт» над какими-то данными, то методы для обработки этих данных должны быть в этом классе.

Information Expert (информационный эксперт)

```
public class Shop
    {
        private readonly List<Order> _orders = new List<Order>();

        public decimal GetTotalAmount()
        {
            return this._orders.Sum(x => x.Amount);
        }
    }
```

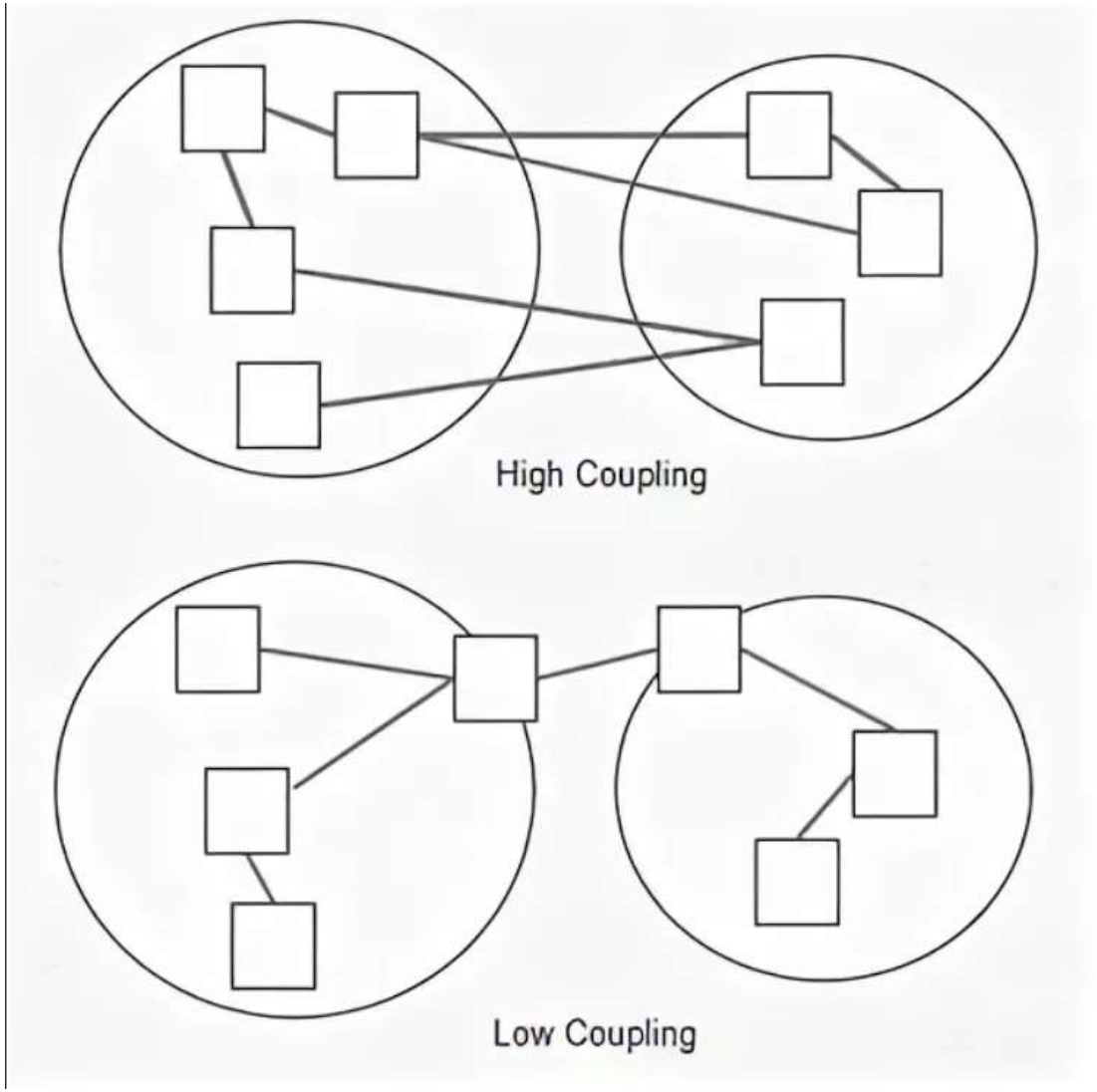
Low coupling (слабое зацепление)

- Степень зацепления — мера неотрывности элемента от других элементов.
- Слабое зацепление является оценочной моделью, которая диктует, как распределить обязанности, которые необходимо поддерживать.
- Слабое зацепление — распределение ответственностей и данных, обеспечивающее взаимную независимость классов. Класс со «слабым» зацеплением:
    - Имеет слабую зависимость от других классов;
    - Не зависит от внешних изменений (изменение в одном классе оказывает слабое влияние на другие классы);
    - Прост для повторного использования.

Low coupling (слабое зацепление)

- Степень зацепления — мера неотрывности элемента от других элементов.
- Слабое зацепление является оценочной моделью, которая диктует, как распределить обязанности, которые необходимо поддерживать.
- Слабое зацепление — распределение ответственностей и данных, обеспечивающее взаимную независимость классов. Класс со «слабым» зацеплением:
    - Имеет слабую зависимость от других классов;
    - Не зависит от внешних изменений (изменение в одном классе оказывает слабое влияние на другие классы);
    - Прост для повторного использования.

Low coupling (слабое зацепление)



Минусы сильной зацепленности модулей

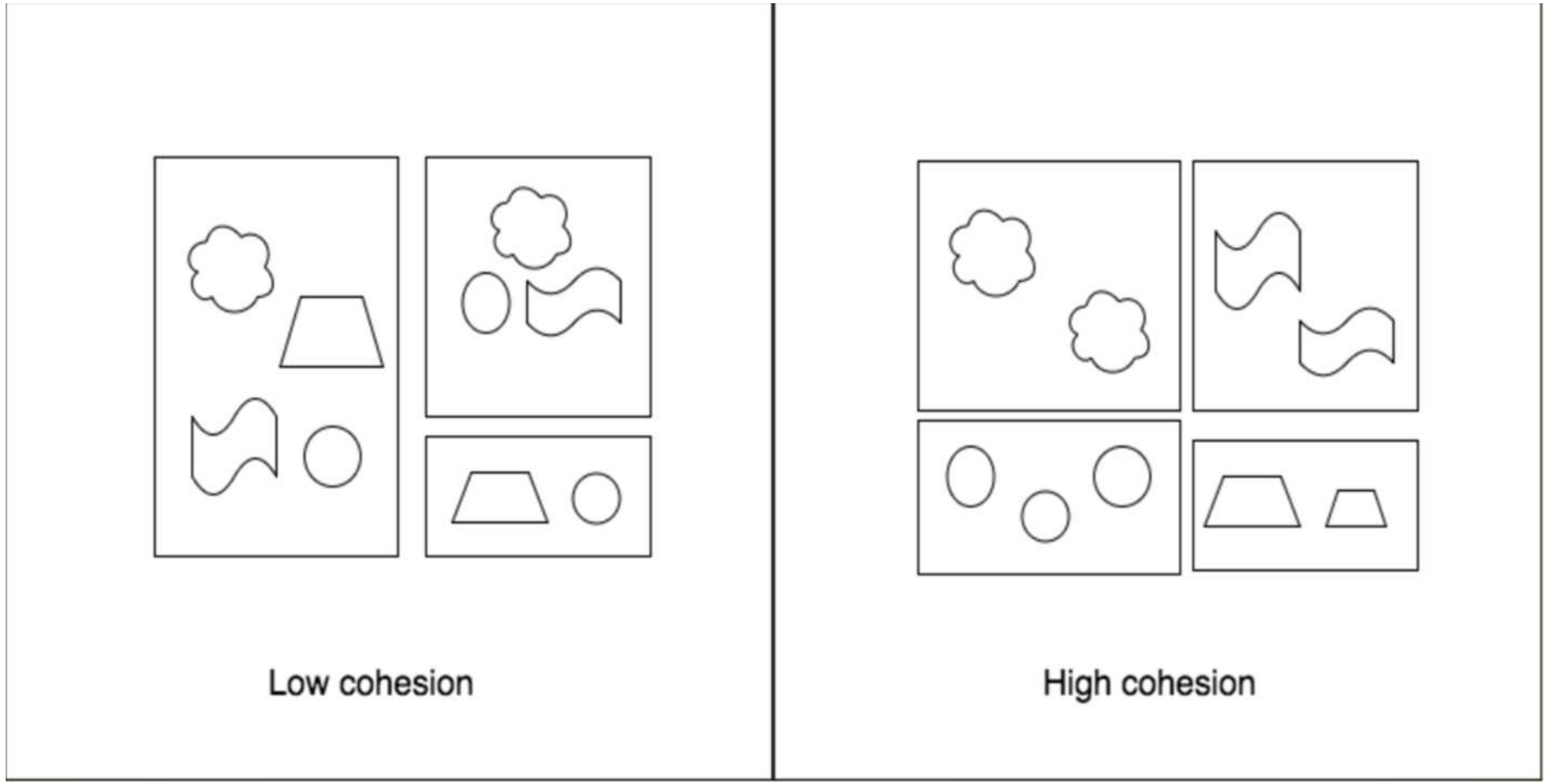
- Затрудняется понимание логики модулей
- Сложная модификация программ
- Проблемы с тестированием
- Невозможность переиспользования отдельного модуля

High cohesion (высокая связность)

Класс должен стараться выполнять как можно меньше не специфичных для него задач, и иметь вполне определенную область применения

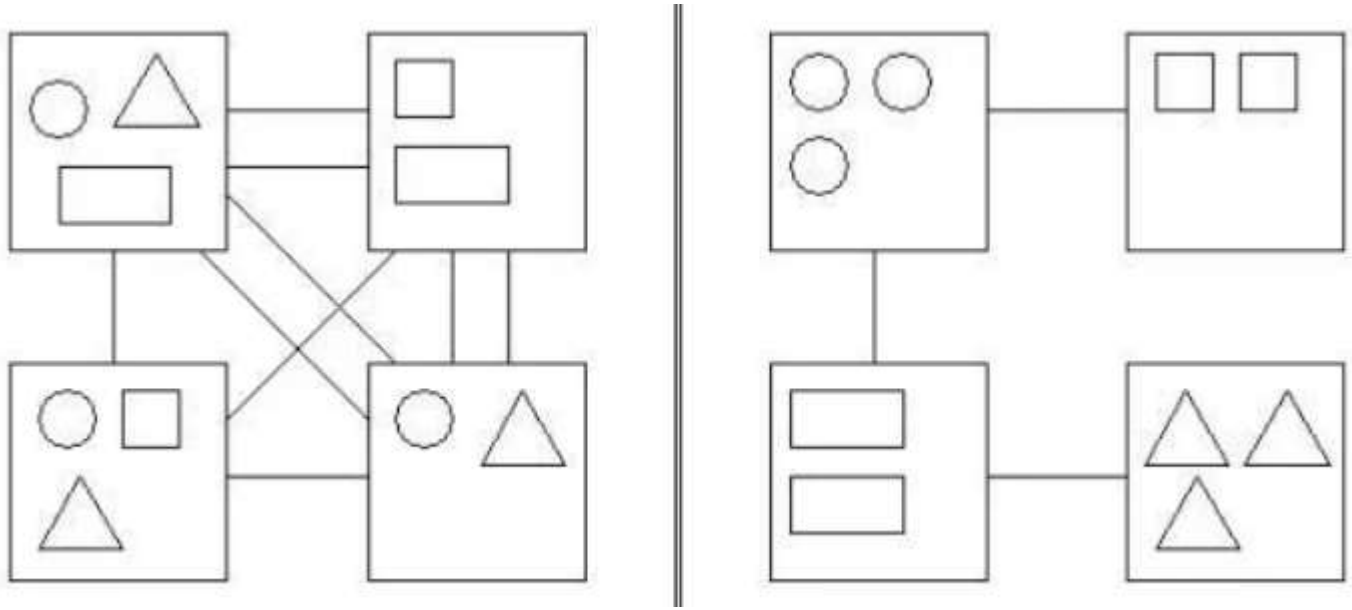
Если модуль несет много разноплановой ответственности, он сложнее в поддержке и переиспользовании, а также будет часто меняться

High cohesion (высокая связность)



Low coupling & High cohesion

Принципы дополняют и поддерживают друг друга.
Все остальные принципы призваны уменьшить зацепление (и увеличить связность).



High coupling & low cohesion

Low coupling & high cohesion

Controller (контроллер)

Controller — это объект, который отвечает за обработку системных событий, и при этом не относится к интерфейсу пользователя. Controller определяет методы для выполнения системных операций

Creator (создатель)

Создавать объекты должен тот класс, который (преимущественно) эти объекты использует (агрегирует либо вызывает методы).

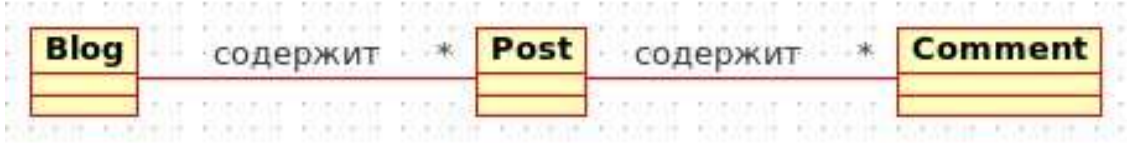
Сочетается с принципом «информационного эксперта» и с принципом низкой связности (знать об объектах должен только тот, кто их использует).

Не сочетается с концепцией Dependency Injection, при которой многие объекты создаются при старте приложения, а потом с помощью DI-механизма «внедряются» друг в друга

Creator (создатель)

объект А должен порождать объект Б, если:

- объект А содержит или агрегирует объекты Б (содержит в себе как свойство или коллекцию)
- объект А активно использует объекты Б (основной объем работы с объектом Б происходит посредством объекта А)
- объект А обладает данными инициализации объекта Б (каждый раз при создании объекта Б, данные берутся из объекта А)



Creator (создатель)

```
public class Shop
    {
        private readonly List<Order> _orders = new List<Order>();

        public void AddOrder(List<OrderProduct> orderProducts)
        {
            var order = new Order(orderProducts); // создатель
            _orders.Add(order);
        }
    }
```

Pure Fabrication (чистая выдумка)

Создание объекта, не отражающего никакую сущность доменной области, а нужного лишь для уменьшения связанности, усиления зацепления и легкости переиспользования.

К pure fabrication можно отнести все объекты сервисного слоя (например, объекты для общения с БД).

Pure Fabrication (чистая выдумка)

```
// Data Transfer Object

public class PersonDTO
{
    public string Name { get; set; }

    public int Age { get; set; }
}
```

Indirection (перенаправление)

Перенаправление реализует низкую связность между классами, путем назначения обязанностей по их взаимодействию дополнительному объекту — посреднику.

Яркий пример - Controller из триады MVC.

Polymorphism (полиморфизм)

В данном контексте шаблон Polymorphism решает проблему обработки альтернативных вариантов поведения на основе типа. Решать эту проблему стоит с использование полиморфных операций, а не с помощью проверки типа и условной логики.

Впоследствии можно легко расширять систему, добавляя новые вариации.

Protected Variations (устойчивость к изменениям)

Проблема модификации системы наиболее актуальна в условиях динамически изменяющихся требований. Зачастую, удается выделить т.н. точки неустойчивости системы, которые наиболее часто будут подвержены изменению/модификации. Тогда определим эти точки в качестве интерфейсов.

Принципы проектирования SOLID

S — single responsibility
O — open-closed
L — Liskov substitution
I — interface segregation
D — dependency inversion

Собраны Робертом Мартиным

SRP: The Single Responsibility Principle
Принцип единственной ответственности

Каждый модуль (класс) должен быть ответственен за единственную задачу. Он может иметь несколько методов, но они должны использоваться лишь для решения общей задачи.

Пример: модуль, формирующий отчеты, не должен отвечать за хранение и извлечение данных для отчетов.

Принцип коррелирует с принципом Information Expert в GRASP.

OCP: The Open Closed Principle
Принцип открыт/закрыт

Система должна быть открыта для расширения, но закрыта для модификации.

Имеется в виду использование полиморфизма, то есть расширение через новую имплементацию интерфейса, либо через расширение интерфейса.

Коррелирует с принципами Polymorphism и Protected Variations.

LSP: The Liskov Substitution Principle
Принцип подстановки Лисков

Если для каждого объекта o1 типа S существует объект o2 типа T, такой,
что для любой программы P, определенной в терминах T,
поведение P не изменяется при замене o2 на o1, то S является подтипом T.

$$(\forall o_1 \in S)$$
$$(\exists o_2 \in T)$$
$$\forall P \in T$$
$$P(o_1) = P(o_2)$$

LSP: The Liskov Substitution Principle
Принцип подстановки Лисков

Производный класс должен быть взаимозаменяем с базовым классом.

Для этого должны выполняться условия:

1. Предварительные условия не могут быть усилены в подтипе.
2. Постусловия не могут быть ослаблены в подтипе.
3. Инварианты супертипа должны быть сохранены в подтипе.

Клиент не должен знать, использует ли он базовый класс или какой-то из его подтипов (поведение должно быть ожидаемым).

LSP: Квадрат-прямоугольник

```
class Rectangle
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }

    public int GetArea() => Width * Height;
}
```

```
void ProcessRectangle(Rectangle rect)
{
    rect.Width = 20;
    rect.Height = 10;
    var area = rect.GetArea();
    ...
}
```

```
class Square : Rectangle
{
    public override int Width
    {
        get => base.Width;
        set
        {
            base.Width = value;
            base.Height = value;
        }
    }

    public override int Height
    {
        get => base.Height;

        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
}
```

ISP: The Interface Segregation Principle
Принцип разделения интерфейсов

Много небольших интерфейсов для разных клиентов лучше, чем один большой общий интерфейс.

При нарушении этого принципа клиент, использующий некоторый интерфейс со всеми его методами, зависит от методов, которыми не пользуется, и поэтому оказывается восприимчив к изменениям в этих методах. В итоге мы приходим к жесткой зависимости между различными частями интерфейса, которые могут быть не связаны при его реализации.

ISP: The Interface Segregation Principle
Принцип разделения интерфейсов

Плохо

```
interface IDevice
{
    void Print(...);
    void Fax(...);
    void Scan(...);
}
```

Хорошо

```
interface IPrinter
{
  void Print(...);
}

interface IFax
{
  void Fax(...);
}

interface IScanner
{
  void Scan(...);
}
```

DIP: The Dependency Inversion Principle
Принцип инверсии зависимостей

Зависимости строятся на абстракциях (интерфейсах).

Модули более высоких уровней (бизнес-логики) не должны зависеть от модулей более низкого уровня (ввода/вывода).

Детали должны зависеть от абстракций, а не наоборот.

Не путать с внедрением зависимостей!

Выводы

Семь раз отмерь - один раз отрежь.
Сначала проектируем - потом пишем код.