SOLID

* Это принципы проектирования классов.

Основные идеи:

1. Увеличение уровня абстракции: интерфейсы, а не реализация.
2. Усиления связей внутри класса
3. Ослабление связей между классами
4. Хорошо спроектированный класс потребуется меньше менять в будущем

1.Single Responsibility Principle

2. Open/Close Principle

3/ Liskov Substituion Principle

4.Interface Segregation Principle

5.Dependency Inversion Principle

**Single Responsibility Principle** (**SRP) - Принцип единственной обязанности**

На каждый класс должна быть возложена одна единственная обязанность.

Класс должен иметь один повод для изменения. Этот принцип является реализацией условий «Сильное внутреннее сцепление «high cohesion».

Например, если класс задает направление движение машины, то этот класс не должен выполнять какие-либо другие задачи. Таким образом, данный принцип помогает разбирать общую конструкцию на независимые модули и уменьшать межмодульную связь.

Сложность в применении данного принципа заключается в том, что прежде всего нужно научиться правильно чувствовать границы его использования.

**OCP – принцип открытости. Закрытости**

Программные сущности должны быть открыты для расширения, но закрыты для изменений.

Требует переходить от реализации к абстракции.

Суть: Не следует менять классы каждый раз

Если понадобиться добавить новую функциональность к классу, то существующий класс не модифицируем, а создаем наследника класса с новыми возможностями. То есть у нас должна быть возможность расширять класс без изменения самого класса.

**Liskov Substituion Principle** **LSP – Принцип замещения Барбары Лисков**

Как правильно наследоваться и при этом абстрагироваться.

Следует корректно реализовать публичный контракт интерфейса не только физически, но и логически.

Принцип запрещает опускаться обратно от абстракции к реализации.

Пусть у нас есть класс Прямоугольник с методами, задающий ширину, высоту и рассчитывающий площадь. Теперь мы захотели создать класс Квадрат. Квадрат – тот же самый треугольник, но с одинаковыми сторонами Класс Квадрат наследуется от прямоугольника и переопределяет его методы: подставляем значения все работает. Но если мы начнем использовать класс прямоугольника в качестве интерфейса, а работать будем с классом Квадрат, мы изменяем оба параметра. Чтобы решить эту проблему, создается общий интерфейс для обоих классов и вместо наследования используется интерфейс.

**Interface Segregation Principle IPS – принцип разделения интерфейсов**

Много специализированных интерфейсов лучше, чем один универсальный.

Допустим, есть класс auto с методами комплектации для всех автомобилей. Если мы наследуемся от интерфейса, то все методы реализованы в нем должны быть описаны в классе – наслелника. В результате чего классы могут получить ненужные методы. Для решения этой проблемы мы разделяем интерфейсы.

**Dependency Inversion Principle DIP – принцип инверсий зависимостей**

Зависимости внутри системы строиться на основе абстракции. Модули верхнего уровня не зависят от модулей нижнего уровня. Оба типа модулей должны зависеть об абстракции.

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракции.

Мы на объект просто ссылаемся. А значит зависимость от него слабая, как того и требует условие – low coupling.

Допустим, у нас есть низкоуровневый класс HTTPService с логикой запроса и высокоуровневый класс HTTP, в конструктор которого мы передаем низкоуровневый модуль. После чего вызываем его методы и нарушаем принцип инверсии зависимости: высокоуровневый модуль зависит от низкоуровневого. Для решения проблемы мы создаем отдельный интерфейс и передаем его в высокоуровневый интерфейс. Теперь наш класс не зависит от низкоуровневого модуля.

SRP – делай модули меньше, реализация концепции high cohesion

OCP – делай модули расширяемыми

LSP – наследуй правильно

ISP – дроби интерфейсы

DIP – используй интерфейсы

Одиночка – это порождающий паттерн проектирования, который гарантирует, что класс имеет только один экземпляр и предоставляет глобальную точку доступа к нему.

Одиночка решает сразу две проблемы (нарушая принцип единой обязанности класса)

1. Гарантирует наличие единственного экземпляра класса. Чаще всего полезно для доступа к какому-либо общему ресурсу, например, бд.

Представим, что мы создали объект, а через некоторое время пытаетесь создать еще один. В этом случае хотелось бы получить старый объект вместо создания нового.

Такое поведение невозможно реализовать с помощью обычного конструктора, поскольку конструктор класса всегда возвращает новый объект.

1. Предоставляет глобальную точку доступа. Это не просто глобальная переменная, через которую можно добраться до определенного объекта, Глобальные переменные не защищены от записи, по этому любой код может подменить их значения без ведома.

Решение

Все реализации Одиночки сводятся к тому, чтобы скрыть конструктор и создать публичный статический метод, который и будет контролировать жизненный цикл объекта – одиночки.

Еслси у вас есть доступ к классу одиночки, значит, будет и доступ к этому статическому методу. С такой точки кода вы бы его вызывали, он всегда будет отдавать один и тот же объект.

Фасад – это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фрейворку.

Проблема

Коду приходится работать с большим количеством объектов определенной сложной библиотеки или фреймворка. Должны самостоятельно инициализировать эти объэкты, следить за правильным порядком зависимостей.

В результате бизнес-логика классов тесно переплетается с деталями реалиации посторонних классов. Такой код достаточно сложно понимать и поддерживать.

Решение

Фасад полезен в том случае, если вы используете какую-нибудь сложную библиотеку с множеством движущихся частей, из которых вам нужна только часть.

Например, программа, заливающая в социальные сети видео с котятами, может использовать профессиональную библиотеку для сжатия видео, но все, что нужно клиентскому коду этой программы, это простой метод encode(filename, format). Создав класс с таким методом, вы реализуете свой первый фасад.

1. **Фасад** дает быстрый доступ к определенной функциональности подсистемы. Он знает, каким классам нужно переадресовать запрос, и какие данные для этого нужны.
2. **Дополнительный фасад** можно ввести, чтобы не загромождать единый фасад разнородной функциональностью. Он может использоваться как клиентом, так и другими фасадами.
3. **Сложная подсистема** имеет множество разных классов. Для того чтобы заставить всех их что-то делать, нужно знать подробности устройства подсистемы, порядок инициализации объектов и другие детали.

Классы подсистемы не знают о существовании фасада и работают друг с другом напрямую.

1. **Клиент** использует фасад вместо непосредственной работы с объектами сложной подсистемы.

Стратегия – это поведенческий паттерн проектирования, определяющий семейство схожих алгоритмов и размещающий каждый из них в собственном классе. После этого алгоритмы можно заменять друг на друга прямо при выполнении программы.

Проблема

Решили написать программу навигатор. Она должна показывать карту, которая позволяет легко ориентироваться в незжнакомых местах.

Функции: прокладка маршрутов из точки а в точку б.

Первая версия прокладывала только автомобильную версию, мы захотели разширить добавили еще пешком, позже еще захотели расширить и добавили общественный транспорт, с дальнейшем с велодорожками и так далее.

При добавлении новых возможностей класс расширяется каждый раз в двое, мало того, что становится трудно ориентироваться так и с каждым новым алгоритмо багов ставало больше, которые требует времени на исправление.

Решение

Паттерн стратегия предлагает определить семейство похожих алгоритмов, которые часто изменяются или расширяются, и вынести их в собственные классы, которые называются стратегиями.

Вместо того. Чтобы начальный класс сам выполянл алгоритмы, он будет играть роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнением работы. Чтобы изменить алгоритм, достаточно подставить в контекст другой объект.

Важно чтобы все стратегии имели один интерфейс. Используя этот интерфейс, контекст будет независим от конкретных классов стратегий.

1. **Контекст** сохраняет ссылку на объект конкретной стратегии, работая с ним через общий интерфейс стратегий.
2. **Стратегия** определяет интерфейс, общий для всех вариантов алгоритма. Контекст использует этот интерфейс для вызова алгоритма.

Для контекста неважно, какая именно вариация алгоритма будет выбрана, поскольку они имеют одинаковый интерфейс.

1. **Конкретные стратегии** реализуют разные вариации алгоритма.
2. При выполнении программы контекст получает вызовы от клиента и делегирует их объекту конкретной стратегии.
3. Клиент должен создать объект конкретной стратегии и передать его конструктору контекста. Кроме того, клиент должен иметь возможность заменить стратегию на лету, используя сеттер поля стратегии. Благодаря этому контекст не будет знать о том, какая именно стратегия сейчас выбрана.