Термин "SOLID" представляет собой акроним для набора практик проектирования программного кода и построения гибкой и адаптивной программы. Данный термин был введен 15 лет назад известным американским специалистом в области программирования Робертом Мартином (Robert Martin), более известным как "дядюшка Боб" или Uncle Bob (Bob - сокращение от имени Robert).

Сам акроним образован по первым буквам названий SOLID-принципов:

* **S**ingle Responsibility Principle (Принцип единственной обязанности)
* **O**pen/Closed Principle (Принцип открытости/закрытости)
* **L**iskov Substitution Principle (Принцип подстановки Лисков)
* **I**nterface Segregation Principle (Принцип разделения интерфейсов)
* **D**ependency Inversion Principle (Принцип инверсии зависимостей)

Принципы SOLID - это не паттерны, их нельзя назвать какими-то определенными догмами, которые надо обязательно применять при разработке, однако их использование позволит улучшить код программы, упростить возможные его изменения и поддержку.

**Принцип единственной обязанности** (Single Responsibility Principle) можно сформулировать так:

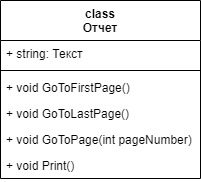
**У класса должна быть только одна причина для изменения**

Под обязанностью здесь понимается набор функций, которые выполняют единую задачу. Суть этого принципа заключается в том, что класс должен выполнять одну единственную задачу. Весь функционал класса должен быть целостным, обладать высокой связностью (high cohesion).

Конкретное применение принципа зависит от контекста. В данном случае важно понимать, как изменяется класс. Если класс выполняет несколько различных функций, и они изменяются по отдельности, то это как раз тот случай, когда можно применить принцип единственной обязанности. То есть иными словами, у класса несколько причин для изменения.

Но если же все функции класса, как правило, изменяются вместе и составляют одно функциональное целое, решают одну задачу, то нет смысла применять данный принцип.

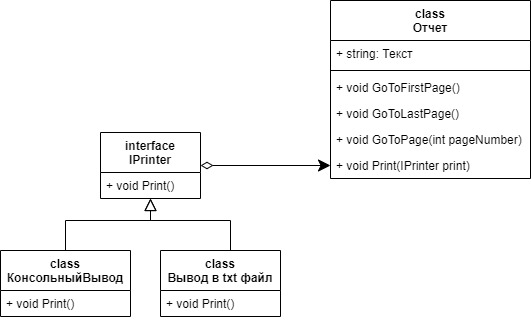
Рассмотрим принцип на примере.



Первые три метода относятся к навигации по отчету и представляют одно единое функциональное целое. От них отличается метод Print, который производит печать. Что если нам понадобится печатать отчет на консоль или передать его на принтер для физической печати на бумаге? Или вывести в файл? Сохранить в формате html, txt, rtf и т.д.? Очевидно, что мы можем для этого поменять нужным образом метод Print(). Однако это вряд ли затронет остальные методы, которые относятся к навигации страницы.

Также верно и обратное - изменение методов постраничной навигации вряд ли повлияет на возможность вывода текста отчета на принтер или на консоль. Таким образом, у нас здесь прослеживаются две причины для изменения, значит, класс Report обладает двумя обязанностями, и от одной из них этот класс надо освободить.

В этом случае мы могли бы вынести функционал печати в отдельный класс, а потом применить агрегацию:



Теперь объект Report получает ссылку на объект IPrinter, который используется для печати, и через метод Print выводится содержимое отчета. Побочным положительным действием является то, что теперь функционал печати инкапсулируется в одном месте, и мы сможем использовать его повторно для объектов других классов, а не только Report.

**Принцип открытости/закрытости (Open/Closed Principle)** можно сформулировать так:

**Сущности программы должны быть открыты для расширения, но закрыты для изменения.**

Суть этого принципа состоит в том, что система должна быть построена таким образом, что все ее последующие изменения должны быть реализованы с помощью добавления нового кода, а не изменения уже существующего.

Теперь приготовление еды абстрагировано в интерфейсе IPrinter, а конкретные способы приготовления определены в реализациях этого интерфейса. А класс Отчет делегирует приготовление еды методу Print объекта IPrinter. Теперь класс Report закрыт от изменений, зато мы можем легко расширить его функциональность, определив дополнительные реализации интерфейса IPrinter.

**Принцип подстановки Лисков** (Liskov Substitution Principle) представляет собой некоторое руководство по созданию иерархий наследования. Изначальное определение данного принципа, которое было дано Барбарой Лисков в 1988 году, выглядело следующим образом:

*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.*

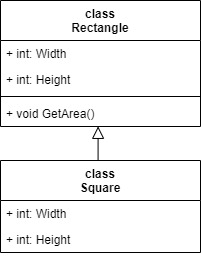
То есть иными словами класс S может считаться подклассом T, если замена объектов T на объекты S не приведет к изменению работы программы.

В общем случае данный принцип можно сформулировать так:

**Должна быть возможность вместо базового типа подставить любой его подтип**.

Фактически принцип подстановки Лисков помогает четче сформулировать иерархию классов, определить функционал для базовых и производных классов и избежать возможных проблем при применении полиморфизма.

Проблему, с который связан принцип Лисков, наглядно можно продемонстрировать на примере двух классов Прямоугольника и Квадрата. Пусть они будут выглядеть следующим образом:



Как правило, квадрат представляют как частный случай прямоугольника - те же прямые углы, четыре стороны, только ширина обязательно равна высоте. Поэтому в классе Квадрат у одного свойства устанавливаются сразу и ширина, и высота.

На первый взгляд вроде все правильно, классы предельно простые, всего два свойства, и, казалось бы, сложно где-то ошибиться. Однако представим ситуацию:

class Program

{

    static void Main(string[] args)

    {

        Rectangle rect = new Square();

        TestRectangleArea(rect);

        Console.Read();

    }

    public static void TestRectangleArea(Rectangle rect)

    {

        rect.Height = 5;

        rect.Width = 10;

        if (rect.GetArea() != 50)

            throw new Exception("Некорректная площадь!");

    }

}

С точки зрения прямоугольника метод TestRectangleArea выглядит нормально, но не с точки зрения квадрата. Мы ожидаем, что переданный в метод TestRectangleArea объект будет вести себя как стандартный прямоугольник. Однако квадрат, будучи в иерархии наследования прямоугольником, все же ведет себя не как прямоугольник. В итоге программа вывалится в ошибку.

Иногда для выхода из подобных ситуаций прибегают к специальному хаку, который заключается в проверке объекта на соответствие типам.

Но такая проверка не отменяет того факта, что с архитектурой классов что-то не так. Более того такие решения только больше подчеркивают проблему несовершенства архитектуры. И проблема заключается в том, что производный класс Square не ведет себя как базовый класс Rectangle, и поэтому его не следует наследовать от данного базового класса. В этом и есть практический смысл принципа Лисков. Производный класс, который может делать меньше, чем базовый, обычно нельзя подставить вместо базового, и поэтому он нарушает принцип подстановки Лисков.

Существует несколько типов правил, которые должны быть соблюдены для выполнения принципа подстановки Лисков. Прежде всего это правила контракта.

Контракт представляет собой некоторый интерфейс базового класса, некоторые соглашения по его использованию, которым должен следовать класс-наследник. Контракт задает ряд ограничений или правил, и производный класс должен выполнять эти правила:

• Предусловия (Preconditions) не могут быть усилены в подклассе. Другими словами подклассы не должны создавать больше предусловий, чем это определено в базовом классе, для выполнения некоторого поведения

Предусловия представляют набор условий, необходимых для безошибочного выполнения метода. Например, в лассе Rectangle можно поставить условие, что вводимое значение не должно быть меньше 0. Тогда в классе Square это условие также должно быть. Причем объектом предусловий могут быть только общедоступные свойства или поля класса или параметры метода, как в данном случае. Приватное поле не может быть объектом для предусловия, так как оно не может быть установлено из вызывающего кода.

* Постусловия (Postconditions) не могут быть ослаблены в подклассе. То есть подклассы должны выполнять все постусловия, которые определены в базовом классе.

Постусловия проверяют состояние возвращаемого объекта на выходе из функции. Например, если функция будет проверять результат, не меньше ли он нуля?

* Инварианты (Invariants) — все условия базового класса - также должны быть сохранены и в подклассе

Инварианты - это некоторые условия, которые остаются истинными на протяжении всей жизни объекта. Как правило, инварианты передают внутреннее состояние объекта. Например, проверка при инициализации поля на корректные данные, высота равна ширине в квадрате.

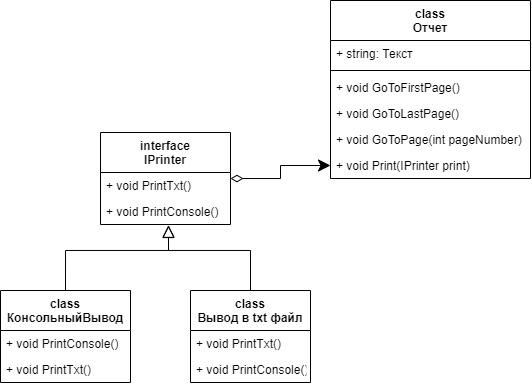
**Принцип разделения интерфейсов** (Interface Segregation Principle) относится к тем случаям, когда классы имеют "жирный интерфейс", то есть слишком раздутый интерфейс, не все методы и свойства которого используются и могут быть востребованы. Таким образом, интерфейс получатся слишком избыточен или "жирным".

Принцип разделения интерфейсов можно сформулировать так:

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

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.

В этом случае интерфейс класса разделяется на отдельные части, которые составляют раздельные интерфейсы. Затем эти интерфейсы независимо друг от друга могут применяться и изменяться. В итоге применение принципа разделения интерфейсов делает систему слабосвязанной, и тем самым ее легче модифицировать и обновлять.



Пример как не правильно.

**Принцип инверсии зависимостей** (Dependency Inversion Principle) служит для создания слабосвязанных сущностей, которые легко тестировать, модифицировать и обновлять. Этот принцип можно сформулировать следующим образом:

**Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций.**

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

Все тоже самое, как и в первом примере с отчетом главное, что в классе используется интерфейс.