Первый слайд  
SOLID – данный акроним или сокращение широко используется в ООП. SOLID по факту – это пять разных под принципов:

* Принцип единственной ответственности – Single Responsibility
* Открытости и закрытости – Open – Closed
* Принцип подстановки Барбары Лисков – Liskov Substitution
* Разделение интерфейсов – Interface segregation
* Инверсия зависимости – Dependency Inversion

Каждый из перечисленных принципов это рекомендация, совет или best practices по организации структуры проекта. У каждого есть своя область и граница применения. Давайте разберем каждый из принципов.

Второй слайд

Принцип единственной ответственности -Single Responsibility Principle.

Суть принципа звучит так – Каждый модуль (компонент) должен иметь одну и только одну причину изменения. То есть основная идея – это организовать свой код таким образом, чтобы в случае изменения задействовалась наименьшее количество модулей (класс/структура/метод). Ключевым понятием применительно к данному принципу является Cohesion или связанность и согласованность.  
Допустим, нам надо определить класс отчета, по которому мы можем перемещаться по страницам и который можно выводить на печать. На первый взгляд мы могли бы определить следующий класс:

Третий слайд:  
Весь функционал компонента должен быть целостным, обладать высокой связностью (high cohesion).  
Ключевым понятием применительно к данному принципу является cohesion или связность/согласованность. Это понятие описывает, насколько близко связаны компоненты. Чем больше связность между компонентами, тем больше программа соответствует принципу единой ответственности

Например, первые три метода класса относятся к навигации по отчету и представляют одно единое функциональное целое, обладают высокой связностью. 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? 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. Решением было бы вынести каждую обязанность в отдельный компонент (в данном случае в отдельный класс):  
Речь идет именно об обязанности компонента, в качестве которого может выступать не только тип (например, класс), но и метод или свойство. И вполне возможно, что в одном каком-то методе сгруппировано несколько обязанностей.  
Четвертый слайд – Шестой слайд:  
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.

Однако одного умения готовить картофельное пюре для повара вряд ли достаточно. Хотелось бы, чтобы повар мог приготовить еще что-то. И в этом случае мы подходим к необходимости изменения функционала класса, а именно метода MakeDinner. Но в соответствии с рассматриваемым нами принципом классы должны быть открыты для расширения, но закрыты для изменения. То есть, нам надо сделать класс Cook отрытым для расширения, но при этом не изменять.  
  
Для решения этой задачи мы можем воспользоваться паттерном Стратегия. В первую очередь нам надо вынести из класса и инкапсулировать всю ту часть, которая представляет изменяющееся поведение. В нашем случае это метод MakeDinner. Однако это не всегда бывает просто сделать. Возможно, в классе много методов, но на начальном этапе сложно определить, какие из них будут изменять свое поведение и как изменять. В этом случае, конечно, надо анализировать возможные способы изменения и уже на основании анализа делать выводы. То есть, все, что подается изменению, выносится из класса и инкапсулируется во вне - во внешних сущностях.

Теперь приготовление еды абстрагировано в интерфейсе IMeal, а конкретные способы приготовления определены в реализациях этого интерфейса. А класс Cook делегирует приготовление еды методу Make объекта IMeal.

Теперь класс Cook закрыт от изменений, зато мы можем легко расширить его функциональность, определив дополнительные реализации интерфейса IMeal.  
Это можно было реализовать другим распространенным способом применения принципа открытости/закрытости паттерном Шаблонный метод.

Седьмой слайд:

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

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

Восьмой слайд:

В этом случае подкласс MicroAccount добавляет дополнительное предусловие, то есть усиливает его, что недопустимо. Поэтому в реальной задаче мы можем столкнуться с проблемой – при создании instance MicroAccount c вызовом метода SetCapital(200) – нам выдаст ошибку

С точки зрения класса Account метод вполне является работоспособным. Однако при передаче в него объекта MicroAccount мы столкнемся с ошибкой. В итоге пинцип Лисков будет нарушен.

Девятый слайд:

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. Рассмотрим пример нарушения принципа Лисков при ослаблении постусловия: В качестве постусловия в классе Account используется начисление бонусов в 100 единиц к финальной сумме, если начальная сумма от 1000 и более. В классе MicroAccount это условие не используется. Исходя из логики класса Account, в методе CalculateInterest мы ожидаем получить в качестве результата числа 1200. Однако логика класса MicroAccount показывает другой результат. В итоге мы приходим к нарушению принципа Лисков, хотя формально мы просто применили стандартные принципы ООП - полиморфизм и наследование.

Десятый слайд:  
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:  
  
Поле age выступает инвариантом. И поскольку его установка возможна только через конструктор или свойство, то в любом случае выполнение предусловия и в конструкторе, и в свойстве гарантирует, что возраст не будет меньше 0. И данное условие сохранит свою истинность на протяжении всей жизни объекта User.

Одиннадцатый слайд:

Теперь рассмотрим, как здесь может быть нарушен принцип Лисков. Пусть у нас будут следующие два класса:

Двенадцатый слайд:

С точки зрения класса Account поле не может быть меньше 100, и в обоих случаях, где идет присвоение - в конструкторе и свойстве это гарантируется. А вот производный класс MicroAccount, переопределяя свойство Capital, этого уже не гарантирует. Поэтому инвариант класса Account нарушается.

Во всех трех вышеперечисленных случаях проблема решается в общем случае с помощью абстрагирования и выделения общего функционала, который уже наследуют классы Account и MicroAccount. То есть не один из них наследуется от другого, а оба они наследуются от одного общего класса.

Таким образом, принцип подстановки Лисков заставляет задуматься над правильностью построения иерархий классов и применения полиморфизма, позволяя уйти от ложных иерархий наследования и делая всю систему классом более стройной и непротиворечивой.

Тринадцатый слайд:

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.

Техники для выявления нарушения этого принципа:

* Слишком большие интерфейсы
* Компоненты в интерфейсах слабо согласованы (перекликается с принципом единой ответственности)
* Методы без реализации (перекликается с принципом Лисков

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.

Рассмотрим на примере. Допустим у нас есть интерфейс отправки сообщения:

Интерфейс определяет все основное, что нужно для отправки сообщения: само сообщение, его тему, адрес отправителя и получателя и, конечно, сам метод отправки. И пусть есть класс EmailMessage, который реализует этот интерфейс:

Надо отметить, что класс EmailMessage выглядит целостно, вполне удовлетворяя принципу единственной ответственности. То есть с точки зрения связности (cohesion) здесь проблем нет.

Теперь определим класс, который бы отправлял данные по смс:

Здесь мы уже сталкиваемся с небольшой проблемой: свойство Subject, которое определяет тему сообщения, при отправке смс не указывается, поэтому оно в данном классе не нужно. Таким образом, в классе SmsMessage появляется избыточная функциональность, от которой класс SmsMessage начинает зависеть.

Это не очень хорошо, но посмотрим дальше. Допустим, нам надо создать класс для отправки голосовых сообщений.

Класс голосовой почты также имеет отправителя и получателя, только само сообщение передается в виде звука, что на уровне C# можно выразить в виде массива байтов. И в этом случае было бы неплохо, если бы интерфейс IMessage включал бы в себя дополнительные свойства и методы для этого, например:  
  
И здесь опять же мы сталкиваемся с ненужными свойствами. Плюс нам надо добавить новое свойство в предыдущие классы SmsMessage и EmailMessage, причем этим классам свойство Voice в принципе то не нужно. В итоге здесь мы сталкиваемся с явным нарушением принципа разделения интерфейсов.

Для решения возникшей проблемы нам надо выделить из классов группы связанных методов и свойств и определить для каждой группы свой интерфейс:  
  
Теперь классы больше не содержат неиспользуемые методы. Чтобы избежать дублирование кода, применяется наследование интерфейсов. В итоге структра классов получается проще, чище и яснее.  
  
Семнадцатый слайд:

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.

Класс Book, представляющий книгу, использует для печати класс ConsolePrinter. При подобном определении класс Book зависит от класса ConsolePrinter. Более того мы жестко определили, что печать книгу можно только на консоли с помощью класса ConsolePrinter. Другие же варианты, например, вывод на принтер, вывод в файл или с использованием каких-то элементов графического интерфейса - все это в данном случае исключено. Абстракция печати книги не отделена от деталей класса ConsolePrinter. Все это является нарушением принципа инверсии зависимостей.