Исключения, SOLID, TDD

Базовые принципы объектно-ориентированного проектирования

- В настоящее время сформировались базовые принципы ОО проектирования, позволяющие
  - избавиться от признаков плохого дизайна;
  - создать наилучший дизайн для данного набора функций.

- Набор принципов **SOLID** это аббревиатура из начальных букв пяти основных принципов объектно-ориентированного проектирования.

SOLID

Принципы SOLID

1. **S**ingle-Responsibility Principle (SRP) – принцип единственной обязанности;
2. **O**pen/Closed Principle (OCP) – принцип открытости/закрытости;
3. **L**iskov Substitution Principle (LSP) – принцип подстановки Лисков;
4. **I**nterface Segregation Principle (ISP) – принцип разделения интерфейсов.
5. **D**ependency-Inversion Principle (DIP) – принцип инверсии зависимости

- Базовые принципы были выработаны ценой больших усилий за десятилетия развития технологии программного обеспечения.

- Это совместный результат размышления и работы большого числа разработчиков и исследователей.

1. Принцип единственной обязанности (Single-Responsibility Principle, SRP)

- Описание принципа:

У класса должна быть только одна причина для изменения!

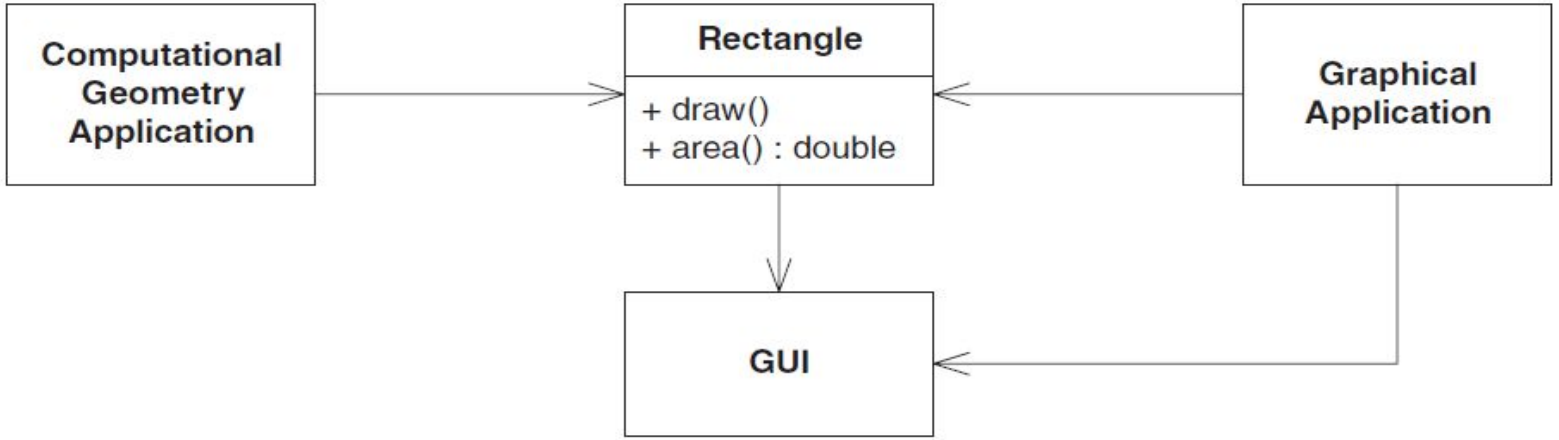
- Любое изменение требований проявляется в изменении распределения обязанностей между классами.

Пояснение принципа

- Каждый класс имеет свои обязанности в программе.
- Если у класса есть несколько обязанностей, то у него появляется несколько причин для изменения.
- Изменение одной обязанности может привести к тому, что класс перестанет справляться с другими.
- Такого рода связанность – причина хрупкого дизайна, который неожиданным образом разрушается при изменении.

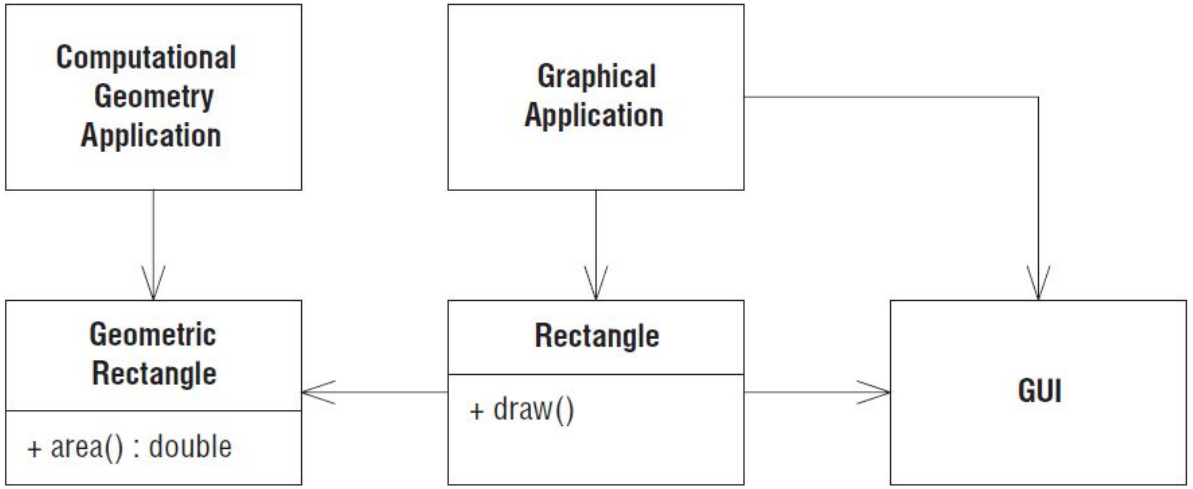
Пример

- Классом Rectangle пользуются два разных приложения.
- Одно приложение связано с вычислительной геометрией.
  - класс Rectangle применяется для вычислений с геометрическими фигурами; на экране оно ничего не рисует.
- Другое приложение связано с графикой (может только частично касаться и вычислительной геометрии),
  - выводит прямоугольник на экран.
- Такой дизайн нарушает принцип SRP.



Более правильный подход к проектированию класса Rectangle

- Необходимо распределить обязанности между двумя разными классами:
  - в класс GeometricRectangle описывается *вычислительная часть* (метод area());
  - в классе Rectangle остается *рисование* (метод draw()).
- В этом случае – изменения в алгоритме рисования прямоугольников не будут повлиять на приложение ComputationalGeometryApplication.



Определение обязанности

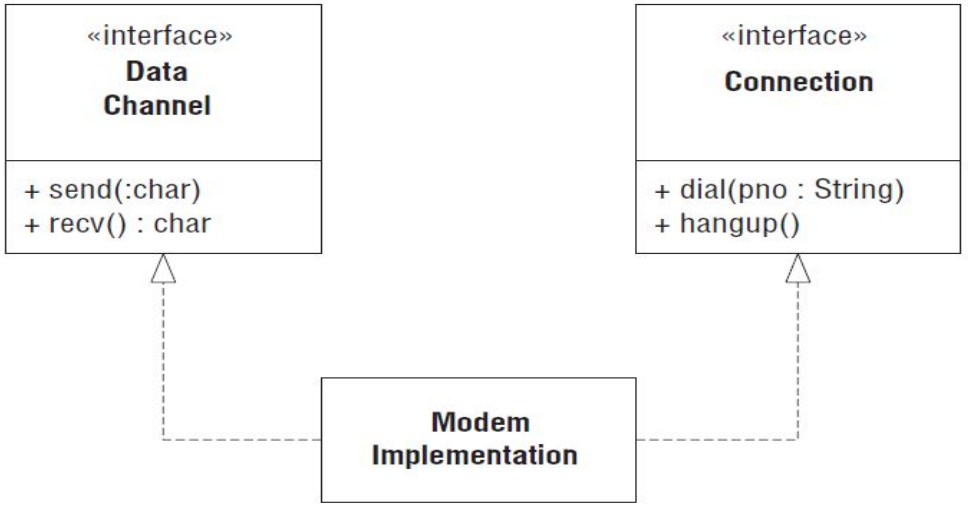
- Если можно найти несколько причин для изменения класса, то у такого класса более одной обязанности.
  - иногда увидеть это трудно.
- Разработчики привыкли воспринимать обязанности группами.

- Например, интерфейс модема:

```
public interface Modem
{
  public void Dial(string pno);  // набор номера
  public void Hangup();          // заканчивать работу
  public void Send(char c);      // отправлять данные
  public char Recv();            // получать данные
}
```

Следует ли разделять группы обязанностей?

- Все зависит от того, как именно ожидается изменение приложения.
- Например: предполагается что будут меняться сигнатуры методов *управления соединением* (Dial() и Hangup())
  - но классы, которые используют методы Send() и Recv() также придется повторно компилировать и развертывать.
- Для того, чтобы это не делать следует интерфейс разделить, как показано ниже:
  - это защищает приложение-клиент от связанности двух обязанностей.



- Если обязанности не меняются по отдельности, то и разделять их нет необходимости.
  - разделение в этом случае попахивает ненужной сложностью.
- Отсюда вытекает следствие.

  Проблема изменения возникает только в том случае, если происходят соответствующие ей изменения.
- Не нужно применять принцип SRP (как и другие принципы) если для того нет причин.

Заключение

- Принцип единственной обязанности – один из самых простых, но при этом его трудно применять правильно.
- Часто объединение обязанностей кажется разработчикам совершенно естественным.
- Их выявление и разделение является одной из задач проектирования ПО.

2. Принцип открытости/закрытости (Open/Closed Principle – OCP)

- Любая система на протяжении своего жизненного цикла претерпевает изменения.
- Об этом следует помнить, разрабатывая систему, которая предположительно переживет первую версию
- Описание принципа:

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

- Применение данного принципа позволяет
  - создавать системы, которые будет сохранять стабильность при изменении требований;
  - будет существовать дольше первой версии.

Пояснение принципа

- Принцип OCP рекомендует проектировать систему так, чтобы в будущем аналогичные изменения можно было реализовать
  - путем добавления нового кода,
  - а не изменением уже работающего.
- Это кажется невозможным, но существуют относительно простые и эффективные способы *приблизиться к такому результату*.

Описание принципа OCP

- Модули, соответствующие принципу OCP, имеют две основных характеристики:

1. ***открыты для расширения***.
   - поведение модуля может быть расширено
     - если требования к приложению изменяются,
     - то можно добавить в модуль новое поведение, отвечающее изменившимся требованиям.
   - т. е. можно изменить, что делает этот модуль.
2. ***закрыты для модификации***.
   - расширение функциональности модуля не приводит к изменению в исходном или двоичном коде модуля.
   - двоичное исполняемое представление модуля (DLL или EXE-файл) остается неизменным.

- Эти характеристики кажутся противоречивыми
  - обычно расширение поведения модуля предполагает изменение его исходного кода.
- Поведение модуля, который нельзя изменить, принято считать ***фиксированным***.
- Можно ли изменить поведение модуля, не трогая его исходного кода?
- Как можно изменить состав функций модуля, не изменяя сам модуль?

Интерфейсы - абстракции

- Принцип OCP можно реализовать с помощью *интерфейсов (абстрактных классов).*
- Интерфейсы фиксированы, но на их основе можно создать неограниченное множество различных поведений
  - ***поведения*** – это ***классы производные от абстракций***.
  - они могут манипулировать абстракциями.
- Интерфейсы (абстрактные классы)
  - могут быть ***закрыты*** для модификации – является фиксированными;
  - но их поведение можно расширять, создавая новые производные классы.

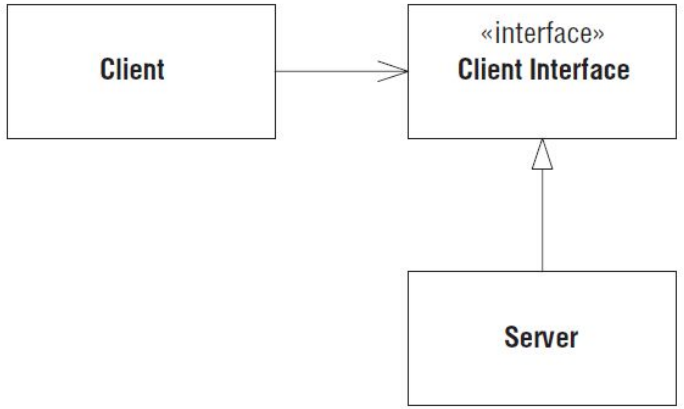
Пример нарушения принципа OCP

- Простой дизайн, в котором класс Client *использует* класс Server.

- Классы Client и Server являются конкретными классами (не абстрактными).
- Такое проектирование (дизайн) нарушает принцип OCP:
  - Если потребуется, чтобы объект класса Client использовал другой серверный объект, то класс Client придется изменить – указать в нем имя нового серверного класса.

Исправление примера (согласование с принципом OCP)

- Изменим дизайн: класс Client будет использовать не класс Server, а интерфейс ClientInterface.



- Интерфейс ClientInterface – это м.б. абстрактный класс, который содержит только абстрактные методы.
  - называется ClientInterface, а не ServerInterface, т.к. *абстрактные классы более тесно ассоциированы со своими клиентами, чем с реализующими их конкретными классами.*
- Класс Client использует такой интерфейс (абстракцию).
- Объекты класса Client будут использовать объекты производного класса Server
  - Здесь используется шаблон проектирования Стратегия (Strategy).

Соблюдение принципа OCP

- Если потребуется, чтобы объекты Client использовали другие серверные классы, то нужно создать новый класс, производный от ClientInterface.
- **Сам класс Client при этом не изменится!**

- У класса Client есть некоторые методы, которые используют методы абстрактного интерфейса ClientInterface.
- Подтипы ClientInterface могут реализовывать этот интерфейс, как сочтут нужным.
- В результате: поведение, описанное в классе Client, можно расширять и модифицировать путем создания новых подтипов ClientInterface.
  - классов реализующих интерфейс ClientInterface

Способы реализации принципа OCP

- Принцип OCP может быть реализован с помощью двух шаблонов проектирования:
  - шаблон «Стратегия»: базовый класс является полностью открытым интерфейсом;
  - шаблон «Шаблонный метод»: базовый класс является одновременно открытым и закрытым.
- Они позволяют добиться четкого отделения общей функциональности от деталей ее реализации.

- Если потребуется расширить поведение метода DrawAllShapes (например: рисовать еще один вид фигур), то достаточно будет добавить новый класс, производный от Shape.

- Сам метод DrawAllShapes изменять не придется.

- Поэтому DrawAllShapes удовлетворяет принципу OCP.

- Его поведение можно расширить без модификации исходного кода.

Например: добавление класса Triangle

- *Это вообще* не скажется ни на одном из приведенных выше модулей.
  - Какие-то части системы все же придется изменить для включения класса Triangle, но весь представленный в листинге код останется неприкосновенным.
- В реальном приложении в классе Shape было бы гораздо больше методов.
- И все равно добавление новой фигуры не вызывает сложностей, потому что нужно лишь создать новый производный класс и реализовать все его методы.
- Не требуется проверять все приложение, выискивая места, требующие изменений.
- Это решение не хрупкое.

Вывод

- Если программа удовлетворяет принципу OCP, то *для ее модификации нужно написать новый код, а не изменить существующий.*
- При этом не возникнет каскада изменений, характерных для программ, которые не следуют этому принципу.
- Единственно необходимые изменения:
  - добавление нового модуля
  - поправки в Main, позволяющие создавать объекты нового типа.

Изменение требований

- Рассмотрим, что произойдет с методом DrawAllShapes, если заказчик потребует *все круги рисовались раньше всех квадратов.*
- Метод DrawAllShapes не закрыт от такого рода изменения.
- Чтобы реализовать новые требования нужно
  - на первом проходе из списка выбирались все объекты Circle;
  - на втором проходе – все объекты Square.

Предвидение и «естественная» структура

- Если бы такие изменения предполагались, то можно было бы придумать абстракцию, защищающую от них.
- Введенные ранее абстракции – скорее помеха, а не подмога для реализации такого изменения.
  - в самом деле, что может быть естественнее базового класса Shape с производными от него Square и Circle?
- Почему эта естественная, хорошо соотносящаяся с реальным миром модель не оптимальна?
- Однако такая модель *не является естественной в системе,* где упорядоченность связана с типом фигуры.

Печальный вывод:

- Каким бы «закрытым» ни был модуль, всегда найдется такое изменение, от которого он не закрыт.
- *Не существует моделей, естественных во всех контекстах!*

Предвидение изменений

- Поскольку от всего закрыться нельзя, то нужно мыслить стратегически.
- Иными словами, проектировщик должен решить, от каких изменений закрыть дизайн:
  - определить, какие изменения наиболее вероятны, а затем сконструировать абстракции, защищающие от них.
- ***Для этого требуется способность к предвидению, которая приходит только с опытом.***

Опытный проектировщик

- достаточно хорошо знает пользователей и предметную область, чтобы оценить вероятность тех или иных изменений.
- призывает на помощь ООП, чтобы защититься от наиболее вероятных изменений.
- Это непростая задача!
- Необходимо строить обоснованные гипотезы о том, с каким изменениями приложение может столкнуться в будущем.
  - Если проектировщик угадывает верно, он вправе торжествовать ☺️
  - Если нет, то возникает проблема ☹️

- Догадки не всегда бывают правильными!
- Следование принципу OCP обходится дорого.
- На создание подходящих абстракций уходят время и силы разработчиков.
- Абстракции увеличивают сложность дизайна программы.
- Существует предел количеству абстракций, которые могут позволить себе разработчики.
- Хотелось бы ограничить применение OCP только вероятными изменениями.
- Но как узнать, какие изменения вероятны?

Определение вероятных изменений

- Вероятные изменения можно определить
  - с помощью исследования;
  - задавая правильные вопросы;
  - призывая на помощь свой опыт и здравый смысл.

- После всего этого *ничего не предпринимается, пока изменение не произойдет!*

Подход

- Правильный подход к разработке ПО.

  «***Обманул меня раз – позор тебе,***

  обманул другой – позор мне».
- Чтобы не перегружать программу ненужной сложностью, можно *один раз* позволить себе быть обманутым.
  - Это означает, что первоначально код пишется без учета возможных изменений.
- Если же изменение происходит, то создаются абстракции, которые в будущем защитят от *такого рода* изменений.

Выводы

- Если программа удовлетворяет принципу OCP, то *для ее модификации нужно написать новый код, а не изменить существующий.*
- При этом не возникнет каскада изменений, характерных для программ, которые не следуют этому принципу.
- Единственно необходимые изменения:
  - добавление нового модуля
  - поправки в Main, позволяющие создавать объекты нового типа.

- Однако, не стоит бездумно применять абстракции вообще ко всем частям приложения.
- ***Нужно применять абстракции только к тем фрагментам программы, которые часто изменяются***.
- *Отказ от преждевременного абстрагирования столь же важен, как и само абстрагирование.*

Заключение

- Во многом принцип открытости/закрытости является ***основой основ объектно-ориентированного проектирования***.
- Следование этому принципу позволяет получить от ООП максимум обещанного:
  - гибкость;
  - возможность повторного использования;
  - удобство сопровождения.
- Чтобы удовлетворить данному принципу, недостаточно просто использовать какой-нибудь ОО язык программирования.

4. Принцип инверсии зависимости (DIP)

Описание принципа:

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

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

Причина использования слова *«инверсия»*

- При структурном анализе и проектировании, принято создавать программные конструкции, в которых
  - модули верхнего уровня зависят от модулей нижнего уровня;
  - стратегия зависит от деталей ☹
- Цель таких методологий: определить иерархию подпрограмм, описывающую, как модули верхнего уровня обращаются к модулям нижнего уровня.
- В правильно спроектированной ОО-программе структура зависимостей должна быть «инвертированной» (т.е. обратной) по отношению к той, что возникает в результате применения

Недостаток зависимости модулей верхнего уровня от модулей нижнего уровня

- В модулях верхнего уровня содержатся важные стратегические решения и бизнес-модели приложения.
  - они отличают одно приложение от другого.
- Если они зависят от модулей нижнего уровня, то
  - изменение модулей нижних уровней напрямую отразиться на модулях верхнего уровня
  - их изменения становятся причиной их изменения модулей верхнего уровня.
- Такое положение – недопустимо!

Приоритетность модулей верхнего уровня

- Модули верхнего уровня (которые определяют стратегию, содержат высокоуровневые бизнес-правила)
  - ***должны влиять на модули нижнего уровня***, а не наоборот;
  - должны быть ***приоритетнее*** модулей, определяющих детали реализации, и независимы от них.
- Модули верхнего уровня вообще никак не должны зависеть от модулей более низкого уровня!

- ***Желательно повторно использовать именно модули верхнего уровня***, которые определяют стратегию.
  - опыт повторного использования низкоуровневых модулей уже накоплен в виде библиотек подпрограмм.
- Если модули верхнего уровня зависят от модулей нижнего уровня, то их трудно повторно использовать в различных контекстах.
- Если модули верхнего уровня не зависят от модулей нижнего уровня, то повторное использование модулей верхнего уровня существенно упрощается.
- ***Этот принцип лежит в основе***

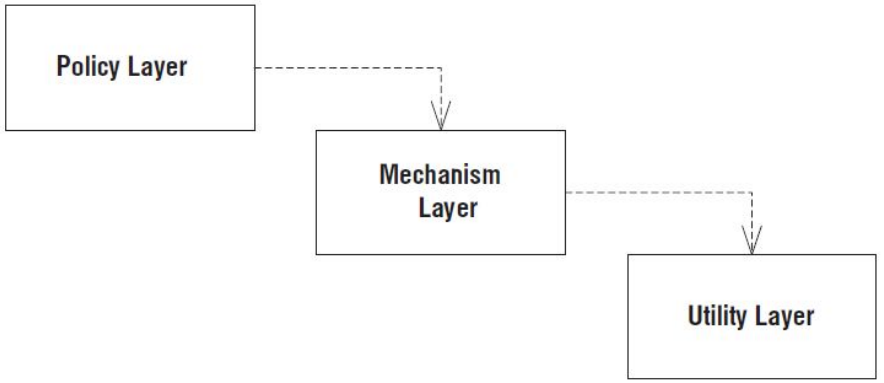
Разбиение программной системы на слои (архитектурный стиль)

- В любой хорошо структурированной объектно-ориентированной архитектуре можно выделить ясно очерченные слои.

- В каждом слое имеется набор тесно связанных сервисов (методов классов) с помощью четко определенных и контролируемых интерфейсов.

Пример: наивная схема разбиения на слои

- Высокоуровневый слой Policy использует слой более низкого уровня Mechanism



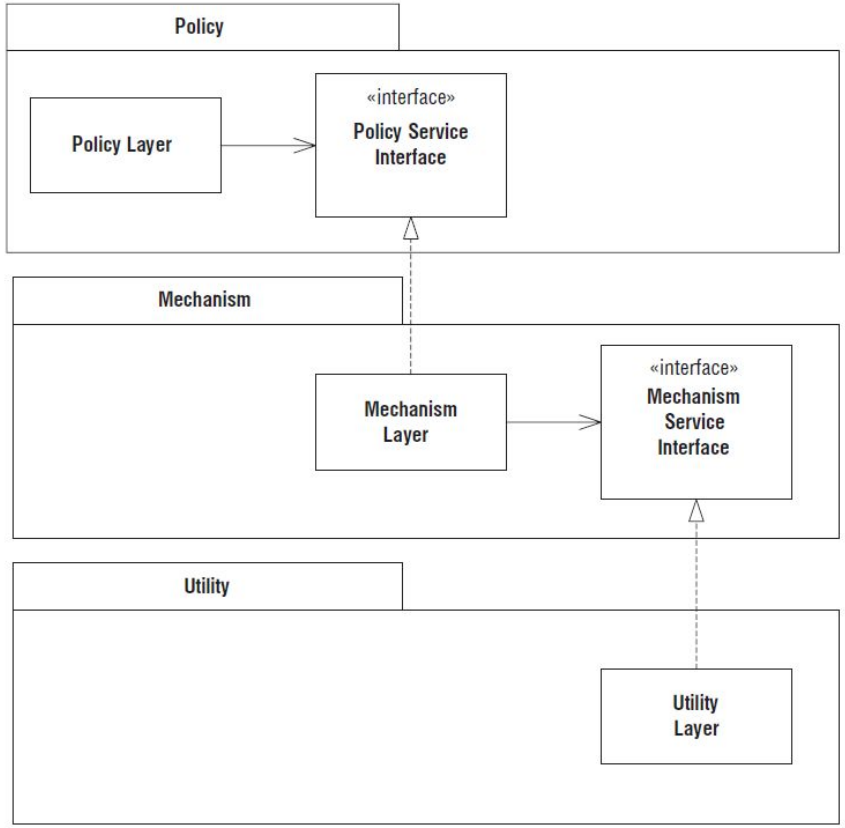
- Слой Mechanism, в свою очередь, пользуется слоем Utility, содержащим детали реализации.
- Такая структура может показаться естественной, но есть один большой недостаток:

  слой Policy зависит от изменений во всех слоях на пути к Utility.

  - такая зависимость ***транзитивна***.

Идея инвертированных слоев

- Слой более высокого уровня объявляет ***абстрактный интерфейс*** служб, в которых он нуждается.
- Слои нижних уровней должны ***реализовывать*** эти интерфейсы.
- В этом случае:
  - Верхние слои не зависят от нижних.
  - Нижние слои зависят от абстрактного интерфейса служб, *объявленного на более высоком уровне.*



Инверсия владения

- В таком подходе инвертируются не только зависимости, но и владение интерфейсами.
  - обычно служебные библиотеки являются владельцами своих интерфейсов.
- ***В соответствии с принципом DIP именно клиентские классы (пользователи) владеют абстрактными интерфейсами – т.е. описывают их.***
  - серверные классы (предоставляющие услуги) наследуют им.
- Модули нижнего уровня предоставляют реализацию интерфейсов, объявленных на более высоком уровне.

Понятие владения

- Под ***владением*** в принципе DIP понимается следующее:

  интерфейсы публикуются владеющими ими клиентами, а не реализующими их серверами.
- Интерфейс находится в том же пакете (библиотеке), что и клиент.
- В результате серверная библиотека или пакет по необходимости зависит от клиентской.

Достоинство инверсии владения

- Благодаря инверсии владения слой PolicyLayer невосприимчив к любым изменениям в слоях MechanismLayer и UtilityLayer.
- Слой PolicyLayer можно повторно использовать с любым нижним слоем, который согласован с интерфейсом PolicyServiceInterface.
- Т.о., в результате инверсии зависимостей, создается структура, которая является:
  - более гибкой,
  - более прочной,
  - более подвижной.

Зависимость от абстракций

- Упрощенная интерпретация принципа DIP:

  «**Зависеть надо от абстракций**».
- В программе не должно быть зависимостей от конкретных классов.
- Все связи должны вести на интерфейс (или абстрактный класс)
  - все связи должны вести на интерфейс (или абстрактный класс);
  - не должно быть переменных, в которых хранятся ссылки на конкретные классы;
  - не должно быть классов, производных от конкретных классов;
  - не должно быть методов, переопределяющих метод, реализованный в одном из базовых классов.

Редко изменяющиеся классы

- Нет явных причин соблюдать это данное эвристическое правило для конкретных, но редко изменяющихся классов.
- Если класс не будет часто изменяться и не предполагается создавать аналогичные ему производные классы, то зависимость от такого класса не принесет особого вреда.
- Например, в большинстве систем класс, описывающий строку, конкретный (в C# это класс string).
  - класс string изменяется редко, поэтому в прямой зависимости от него нет никакой беды.

- Однако конкретные классы, являющиеся частью прикладной программы (которые программисты пишет с*ами*) *в большинстве случаев, изменчивы.*
- Именно от *таких конкретных классов* и не желательно зависеть напрямую.
- Изменчивость собственных классов можно изолировать, скрывая их за абстрактным интерфейсом.
  - однако это решение неполное.

Заключение

- В традиционном структурном программировании структура зависимостей: стратегия зависит от деталей.
  - Это плохо, т.к. стратегия становится восприимчивой к изменению деталей.
- В ОО программировании структура зависимостей инвертируется,
  - детали, и стратегии зависят от абстракции, а интерфейсами служб часто владеют клиенты.

- Инверсия зависимостей – отличительный признак ОО проектирования и неважно, на каком языке написана программа.
  - если зависимости инвертированы, значит, мы имеем ОО дизайн;
  - в противном случае дизайн процедурный.

- Принцип инверсии зависимостей – это
  - фундаментальный низкоуровневый механизм, лежащий в основе многих преимуществ, которые обещают ОО технологии.
  - необходим для создания повторно используемых каркасов (фреймворков).
- Крайне важен для конструирования кода, устойчивого к изменениям.
- Поскольку абстракции и детали изолированы друг от друга, такой код гораздо легче сопровождать.

3. Принцип подстановки Лисков (Liskov Substitution Principle, LSP)

- Описание принципа:

Должна быть всегда возможность вместо базового типа подставлять любой его подтип.

- Механизмы, лежащие в основе принципа открытости/закрытости:
  - абстрагирование;
  - полиморфизм.

- В статически типизированных языках (например, C#) одним из главных механизмов поддержки абстрагирования и полиморфизма – является ***наследование***.
- Именно наследование позволяет нам создавать производные классы, реализующие абстрактные методы, объявленные в базовых классах.

Формулировка Барбары Лисков (в 1988 г.

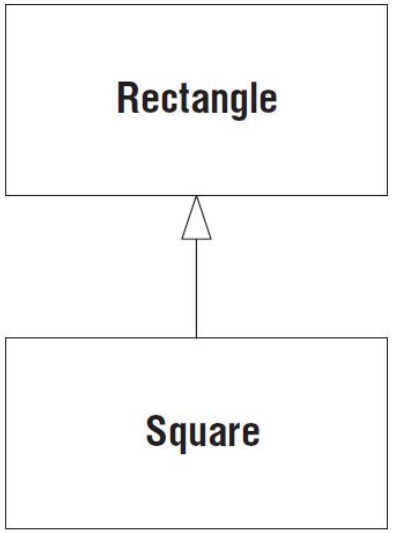
- Свойство подстановки:
  - если для каждого объекта o1 типа S существует объект o2 типа T, такой, что
  - для любой программы P, определенной в терминах T, поведение P не изменяется при замене o1 на o2,
  - то S является подтипом T.

Пример нарушения принципа LSP

- Функция f(B b) принимает в качестве аргумента ссылку на объект базового класса B
- Класс D производный от B.
- ***Предположим***: при передаче функции f под видом объекта класса B объекта некоторого класса D она ведет себя неправильно
- В этом случае класс D нарушает принцип LSP!

Требования немного изменились

- Программа должна работать не только прямоугольниками, но и *квадратами.*
- Часто говорят, что наследование – это отношение *ЯВЛЯЕТСЯ (IS-A).*
  - Если новый вид объекта ЯВЛЯЕТСЯ частным случаем старого вида, то класс нового объекта должен быть производным от класса старого объекта.
- ***Вроде бы*** квадрат во всех отношениях *является прямоугольником.*
- ***Логичный вывод***: класс Square



- Такой вывод может привести к тонким, но весьма существенным проблемам
  - их невозможно предвидеть, пока не столкнешься с ними в программе.
- Первый признак, наводящий на мысль, что тут не все в порядке:
  - классу Square не нужны оба поля height и width.
  - однако же он наследует их от Rectangle.
    - это расточительство;
    - часто таким расточительством можно пренебречь;
    - предположим, что эффективное использование памяти нас не очень волнует.

Правильность не является внутренне присущим свойством

- Важное следствие принципа подстановки Лисков:

  невозможно установить правильность классов модели, если рассматривать их изолированно.

- Правильность модели можно выразить только в терминах ее клиентов.

Вывод

- ***Обдумывая вопрос о том, подходит ли конкретный дизайн, нельзя рассматривать решение в изоляции.***
- Необходимо смотреть на него через призму разумных предположений со стороны пользователей данного дизайна.
- Часто такие предположения выражаются в виде утверждений в автономных тестах, написанных для базового класса
  - Это еще один довод в пользу разработки через тестирование.

Возникающие вопросы

- Что же произошло?
- Почему логичная (на первый взгляд) модель, состоящая из классов Square и Rectangle, оказалась плохой?
- Разве квадрат – это не прямоугольник?
- Разве отношение ЯВЛЯЕТСЯ не имеет места?

Вывод

- С точки зрения автора функции g – созданная модель не верна!
- Квадрат может быть прямоугольником.
- Но с точки зрения функции g объект Square точно *не* является объектом Rectangle.
  - Т.к. *поведение* объекта Square несовместимо с предположениями о поведении объекта Rectangle.
- С точки зрения такого поведения Square не является Rectangle,
  - именно *поведение* и интересует любую программу.

Эвристическое правило

- Существуют простое эвристическое правило, способные подсказать, когда имеет место нарушение принципа LSP.
- ***Производный класс, умеющий делать меньше, чем базовый, обычно нельзя подставить вместо базового, и потому он нарушает LSP.***

Заключение

- Принцип открытости/закрытости (OCP) лежит в основе многих требований ОО проектирования.
- Если этот принцип соблюден, то приложение более надежно, лучше поддается сопровождению и пригодно для повторного использования.
- Принцип подстановки Лисков – один из основных инструментов реализации принципа OCP.

- Возможность подстановки подтипов позволяет без модификации расширять модуль, выраженный в терминах базового типа.
  - разработчики вправе рассчитывать на это по умолчанию.
- Поэтому контракт базового типа должен быть хорошо и ясно понятен
  - если не явно навязан, из кода.

- Термин ЯВЛЯЕТСЯ – слишком общий (широкий), чтобы служить определением подтипа.
- Правильное определение подтипа – ***заместим***
  - *может ли объект одного класса заместить объект другого класса.*
- ***Заместимость*** определяется явным или неявным контрактом.

5. Принцип разделения интерфейсов (ISP)

- Принцип разделения интерфейсов (ISP):

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

- Если клиент вынужденно зависит от методов, которыми не пользуется, то он оказывается восприимчив к изменениям в этих методах.

- В результате возникает непреднамеренная связанность между всеми клиентами.

«Массивные» интерфейсы

- Класс имеет «массивный» (fat) интерфейс, если функции этого интерфейса недостаточно сцепленные.
- «Массивный» интерфейс класса можно разбить на группы методов.
- Каждая группа предназначена для обслуживания разнотипных клиентов.
- Одним клиентам нужна одна группа методов, другим – другая.

- Могут быть объекты, нуждающиеся в несцепленных интерфейсах, однако клиентам необязательно знать, что это единый класс.
- Клиенты должны лишь знать об абстрактных интерфейсах, обладающих свойством сцепленности.

Смысл принципа разделения интерфейсов

- ***Клиенты не должны вынужденно зависеть от методов, которыми не пользуются.***
- Если клиент вынужденно зависит от методов, которыми не пользуется, то он оказывается восприимчив к изменениям в этих методах.
- В результате возникает непреднамеренная связанность между всеми клиентами.

Вывод

- Если клиент зависит от класса, содержащего методы, которыми этот клиент не пользуется,
  - но пользуются другие клиенты,
- то данный клиент становится зависим от всех изменений, вносимых в класс в связи с потребностями этих «других клиентов».
- Хотелось бы по возможности избегать таких связей и потому нужно стремиться разделять интерфейсы.

Заключение

- Жирные классы приводят к неочевидным и вредным связям между их клиентами.
- Если одному клиенту требуется изменить жирный класс, то оказываются затронуты и все остальные классы.
- Поэтому клиенты должны зависеть только от методов, которые вызывают.

- Для зависимости только от вызываемых методов нужно разбивать интерфейс жирного класса на несколько интерфейсов, специально предназначенных для клиентов.
  - В каждом таком интерфейсе объявляются только методы, которые вызывает конкретный клиент или группа клиентов.
- После этого жирный класс может унаследовать всем специальным для клиентов интерфейсам и реализовать их.
  - разрывает зависимость клиента от методов, к которым он не обращается,
  - делает клиентов независимыми друг от друга.

Обработка исключений

В программах периодически возможны сценарии, которые приводят к ошибкам.

Например, пользователь вводит текст там, где предполагается ввести число.

Или стечение обстоятельств приводит к делению на ноль.

Сценарий в таком случае закончит работу с ошибкой.

Часто нужно уметь предупредить такой случай, отловив исключение.

```
>>>100 / 0

Traceback (most recent call last):
  File "", line 1, in
    100 / 0
ZeroDivisionError: division by zero
```

Синтаксис отлова исключений такой (общий вид):

```
try:
   блок 1        # интерпретатор пытается выполнить блок1
except (name1,name2):
   блок 2        # выполняется, если в блоке try возникло исключение name1 или
name2
except name3:
   блок 3        # выполняется, если в блоке try возникло исключение name3
except:
   блок 4        # выполняется для всех остальных возникших исключений
else:
   блок 5        # выполняется, если в блоке try не возникло исключения
finally:
   блок 6        # выполнится всегда
```

Обязательно должны быть только try и except

```
try:
    k = 1 / 0
except:
    print ‘Деление на 0 ☹’
```


Отлавливаем любое исключение

```
try:
    k = 1 / 0
except ZeroDivisionError:
    print ‘Деление на 0 ☹’
```


Отлавливаем конкретное исключение (конкретную ошибку)

```
a = input ()

try:
        b = a**3
except:
        print 'Seems like a is not number'
        b = 'try again'
finally:
        print b
```

```
E:\python>test3.py
5
125

E:\python>test3.py
'sd'
Seems like a is not number
try again
```

Обратите внимание, что, как и везде, блоки выделяются отступами.

В этом

```
E:\python>test3.py
dssfd
Traceback (most recent call last):
  File "E:\python\test3.py", line 1, in <module>
    a = input ()
  File "<string>", line 1, in <module>
NameError: name 'dssfd' is not defined
```

```
try:
        a = input ()
except:
        print 'Invalid input'
        a = 0

try:
        b = a**3
except:
        print 'Seems like a is not number'
        b = 'try again'
finally:
        print b
```

```
E:\python>test3.py
zdf
Invalid input
0
```

Разумеется, исключения при вводе данных лучше отслеживать в цикле по
принципу – если введено не то, дать пользователю ввести ещё раз:

```
f = 0
print 'Enter number:'

while f == 0:
    try:
        a = input ()
    except:
        print 'Invalid input. Try again'
        continue

    try:
        print a + 5
    except:
        print 'Invalid input (not a number). Try again'
        continue

    f = 1
```

```
E:\python>test3.py
Enter number:
sd
Invalid input. Try again
q
Invalid input. Try again
'sfs'
Invalid input (not a number). Try again
4
9
```

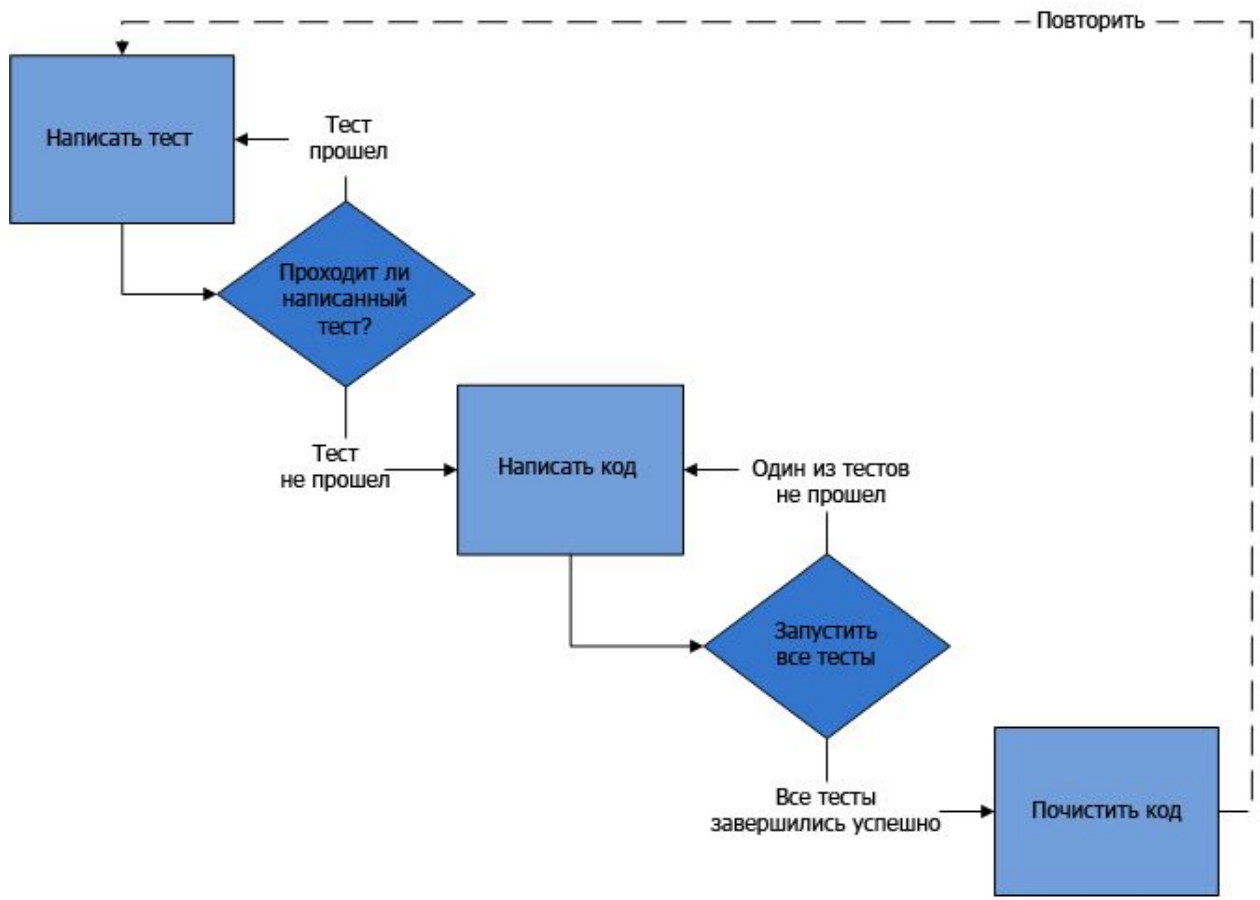
Создание и использование своего исключения

```
class B(Exception):
    pass


class C(B):
    pass


class D(C):
    pass


for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Обычно в скриптах «для себя» исключениями пользуются очень редко.

Но если предполагается передача (или продажа) программы кому-то, нужно обеспечить невозможность вылета программы с ошибкой, то есть предусмотреть
везде обработку исключений.

Разработка через тестирование



Модуль unittest

Для автоматизации тестов, unittest поддерживает некоторые важные концепции:

- **Испытательный стенд** (test fixture) - выполняется подготовка, необходимая для выполнения тестов и все необходимые действия для очистки после выполнения тестов. Это может включать, например, создание временных баз данных или запуск серверного процесса.
- **Тестовый случай** (test case) - минимальный блок тестирования. Он проверяет ответы для разных наборов данных. Модуль unittest предоставляет базовый класс TestCase, который можно использовать для создания новых тестовых случаев.
- **Набор тестов** (test suite) - несколько тестовых случаев, наборов тестов или и того и другого. Он используется для объединения тестов, которые должны быть выполнены вместе.
- **Исполнитель тестов** (test runner) - компонент, который управляет выполнением тестов и предоставляет пользователю результат. Исполнитель может использовать графический или текстовый интерфейс или возвращать специальное значение, которое сообщает о результатах выполнения тестов.

Пример использования unittest

```
import unittest

class TestStringMethods(unittest.TestCase):

  def test_upper(self):
      self.assertEqual('foo'.upper(), 'FOO')

  def test_isupper(self):
      self.assertTrue('FOO'.isupper())
      self.assertFalse('Foo'.isupper())

  def test_split(self):
      s = 'hello world'
      self.assertEqual(s.split(), ['hello', 'world'])
      # Проверим, что s.split не работает, если разделитель - не строка
      with self.assertRaises(TypeError):
          s.split(2)

if __name__ == '__main__':
    unittest.main()
```

Пояснения к примеру

- Тестовый случай создаётся путём наследования от unittest.TestCase.
- Тесты определяются с помощью методов, имя которых начинается на test.
- Суть каждого теста - вызов assert для проверки ожидаемого результата;
- Методы setUp() и tearDown() позволяют определять инструкции, выполняемые перед и после каждого теста, соответственно.

Использование unittest из командной строки

```
python -m unittest test_module1 test_module2

python -m unittest test_module.TestClass

python -m unittest test_module.TestClass.test_method
```

Можно также указывать путь к файлу:

```
python -m unittest tests/test_something.py
```

С помощью флага -v можно получить более детальный отчёт:

```
python -m unittest -v test_module
```

Для нашего примера подробный отчёт будет таким:

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Обнаружение тестов

- unittest поддерживает простое обнаружение тестов. Для совместимости с обнаружением тестов, все файлы тестов должны быть модулями или пакетами, импортируемыми из директории верхнего уровня проекта.

- Обнаружение тестов реализовано в TestLoader.discover(), но может быть использовано из командной строки:

```
cd project_directory
python -m unittest discover
```

Организация тестов

- Базовые блоки тестирования это тестовые случаи - простые случаи, которые должны быть проверены на корректность.
- Тестовый случай создаётся путём наследования от unittest.TestCase.
- Тестирующий код должен быть самостоятельным, то есть никак не зависеть от других тестов.
- Простейший подкласс TestCase может просто реализовывать тестовый метод (метод, начинающийся с test).
- Тестов может быть много, и часть кода настройки может повторяться. К счастью, мы можем определить код настройки путём реализации метода setUp(), который будет запускаться перед каждым тестом

Организация тестов (продолжение)

Можно разместить все тесты в том же файле, что и сама программа (таком как widgets.py), но размещение тестов в отдельном файле (таком как test_widget.py) имеет много преимуществ:

- Модуль с тестом может быть запущен автономно из командной строки.
- Тестовый код может быть легко отделён от программы.
- Меньше искушения изменить тесты для соответствия коду программы без видимой причины.
- Тестовый код должен изменяться гораздо реже, чем программа.
- Протестированный код может быть легче переработан.
- Тесты для модулей на C должны быть в отдельных модулях, так почему же не быть последовательным?
- Если стратегия тестирования изменяется, нет необходимости изменения кода программы.

Пропуск тестов и ожидаемые ошибки

unittest поддерживает пропуск отдельных тестов, а также классов тестов. Вдобавок, поддерживается пометка теста как "не работает, но так и надо".

Пропуск теста осуществляется использованием декоратора skip() или одного из его условных вариантов.

Пропуск тестов (пример)

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass
```

Пропуск тестов (декораторы)

Декораторы, пропускающие тесты или говорящие об ожидаемых ошибках:

@unittest.skip(reason) - пропустить тест. reason описывает причину пропуска.

@unittest.skipIf(condition, reason) - пропустить тест, если condition истинно.

@unittest.skipUnless(condition, reason) - пропустить тест, если condition ложно.

@unittest.expectedFailure - пометить тест как ожидаемая ошибка.

Для пропущенных тестов не запускаются setUp() и tearDown(). Для пропущенных классов не запускаются setUpClass() и tearDownClass(). Для пропущенных модулей не запускаются setUpModule() и tearDownModule().

Подтесты

Когда некоторые тесты имеют лишь незначительные отличия, например некоторые параметры, unittest позволяет различать их внутри одного тестового метода, используя менеджер контекста subTest().

class NumbersTest(unittest.TestCase):

```
def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

Assert

assertEqual(a, b) — a == b

assertNotEqual(a, b) — a != b

assertTrue(x) — bool(x) is True

assertFalse(x) — bool(x) is False

assertIs(a, b) — a is b

assertIsNot(a, b) — a is not b

assertIsNone(x) — x is None

assertIsNotNone(x) — x is not None

assertIn(a, b) — a in b

assertNotIn(a, b) — a not in b

assertIsInstance(a, b) — isinstance(a, b)

assertNotIsInstance(a, b) — not isinstance(a, b)

assertRaises(exc, fun, *args, **kwds) — fun(*args, **kwds) порождает исключение exc

assertRaisesRegex(exc, r, fun, *args, **kwds) — fun(*args, **kwds) порождает исключение exc и сообщение соответствует регулярному выражению r

assertWarns(warn, fun, *args, **kwds) — fun(*args, **kwds) порождает предупреждение

assertWarnsRegex(warn, r, fun, *args, **kwds) — fun(*args, **kwds) порождает предупреждение и сообщение соответствует регулярному выражению r

assertAlmostEqual(a, b) — round(a-b, 7) == 0

assertNotAlmostEqual(a, b) — round(a-b, 7) != 0

assertGreater(a, b) — a > b

assertGreaterEqual(a, b) — a >= b

assertLess(a, b) — a < b

assertLessEqual(a, b) — a <= b

assertRegex(s, r) — r.search(s)

assertNotRegex(s, r) — not r.search(s)

assertCountEqual(a, b) — a и b содержат те же элементы в одинаковых количествах, но порядок не важен

Лабораторная работа

Используя подход TDD, напишите метод, который получает на вход строку, и возвращает строку с большой буквы, и оканчивающуюся точкой. (добавлять точку нужно только, если ее не было изначально)