Поглубже в разработку

Борисенко Глеб

ФТиАД2022

Какие увидел проблемы

- Деление кода по модулям
- Зачем нужны функции, зачем классы
- В принципе что такое классы
- Выделение логических сущностей
- И еще по мелочи до кучи

Что нужно учитывать

Код должен быть:

- Гибким (легко модифицируемым)
- Расширяемым
- Эффективным
- Простым
- Понятным

Все сочетать нельзя

- Удовлетворять всем качествам обычно невозможно.
- Поэтому хорошая разработка - это как ходить с шестом по канату.
- Нужно находить баланс между этими качествами.

Думайте о будущем

- Когда пишите код, думайте о будущем
- Вам или вашим коллегам с высокой вероятностью придется:
  - Чинить проблемы
  - Расширять функционал
  - Убирать функционал (стал по каким-то причинам легаси)
  - Уничтожать технический долг
  - Использовать ваш код как черный ящик
- Пишите код так, чтобы все это было удобно делать и через неделю, и через год.

Деление на модули

- В одном модуле находятся функции/классы, решающие определенную макроболь.
- Это делается для того, чтобы в случае возникновения такой же боли в будущем, мы могли модифицировать/расширять код, используя преимущественно этот модуль и не касаясь других
- Какие могут быть боли:
  - Интеграция с другим сервисом
  - Все, что касается поведения логической сущности (модель, например, работа с данными определенными)

Внутренняя метрика деления

- Есть что-то вроде эвристики, внутреннего ощущения метрики для деления
- Чем меньше мы затрагиваем при модификации/расширении модулей с учетом наименьшего набора строк в них, тем лучше
- То есть для нас есть две метрики:
  - Количество модулей
  - Количество строк в модулях
- И при изменении/расширении мы должны затрагивать как можно меньше модулей как можно меньшего объёма

Проблемы деления

- Приходится балансировать на двух стульях
- Запихнем в один большой модуль - будет много строк кода
- Запихнем в кучу маленьких модулей - будем затрагивать кучу модулей
- Это типа precision и recall

Иерархия модулей

- Код внутри одного модуля выполняет определенную логическую функцию, или имеет один логический смысл
- Модуль должен быть логически замкнутым (сам хз, как это объяснить, это на уровне ощущений)
- Поэтому мы должны уметь составить иерархию модулей
- То есть представить все в виде дерева. Обычно файловая структура вашего репозитория и является такой иерархией
- У вас есть корень репозитория, от которого идут веточки (папочки, модули) и/или листья (функции, классы). От веточек в свою очередь могут идти свои веточки и листья.

Влияние модуля на понимаемость

- Также модули нужны для улучшения понимаемости
- Я вижу модуль, и по его названию уже понимаю, что я там увиже, какую часть функционала он реализует
- Логика модуля должна соответствовать названию
- У вас не будет такого, что придя к окулисту, вам начали проверять кожу на пятке

Функции и классы

- Функция решает микроболь
- У нас нет проблемы хранения чего-то в оперативке постоянно, есть определенная задача, которая должна решиться и при необходимости вернуть нам какое-то значение, всё
- А Класс - это объединение функций и хранения в одном объекте
- Класс - это логическая сущность. У нее есть свои атрибуты и поведение
- Объект класса живет у нас некоторое времени, хранит в себе какую-то информацию, тогда как функция просто выполнилась и все.

ООП

- Абстракция данных
- Инкапсуляция
- Наследование
- Полиморфизм

- Советую глянуть: https://habr.com/ru/post/463125/
- Прям очень советую, выделить часик своего времени и прочитать хорошенечко

Модуль VS Класс

Модуль с функциями можно рассматривать как класс, но:

- У нас только один экземпляр (сам модуль)
- Все данные (атрибуты) будут определены при импорте модуля, а не в тот момент, когда мы в коде вызовем его впервые. А ведь наш модуль может зависеть от заранее определенных вещей.
- Эти данные доступны будут все время выполнения процесса питона. Они навсегда в оперативке. А объект класса мы можем удалить - и данные уйдут из оперативки.

Диаграмма классов

- Перед выполнением задачи в начале определяется структура логических сущностей
- Если их немного, то она может быть в голове
- Если много, то может быть нарисована схема (кастомная или UML-диаграмма)
- На ней отображается, какие у нас вообще есть сущности, как они взаимодействует
- По этой схеме реализуются классы
- Но в ней нет отображения работы функций!

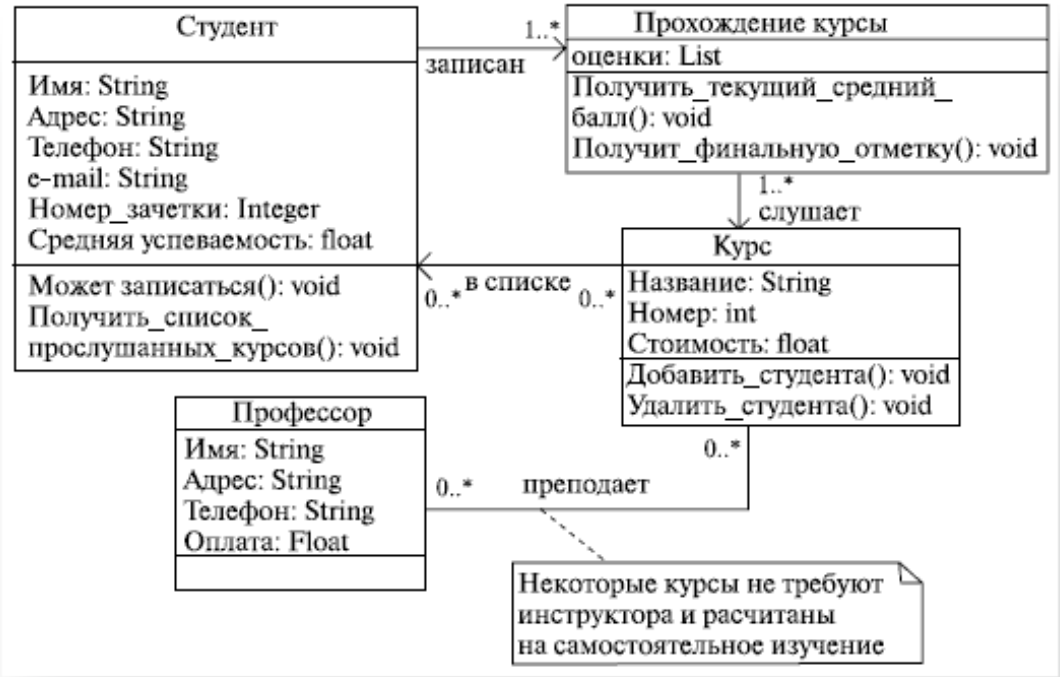
Какие есть связи (взаимодействия) между сущностями

- Зависимость
- Ассоциация
- Агрегация
- Композиция

Функции в классах

- Бывает, функция логически относится к классу, но не является конкретно реализацией поведения этой сущности
- Такую функцию можно всунуть в класс как @staticmethod
- Также бывает, что фунции в какой-то отдельный логический модуль не вынести (ну не выносятся они), тогда можно такие функции вынести в модуль utils.py конкретной директории
- Но utils.py должен быть как можно меньше!

Пример на диаграмму классов, выделение сущностей



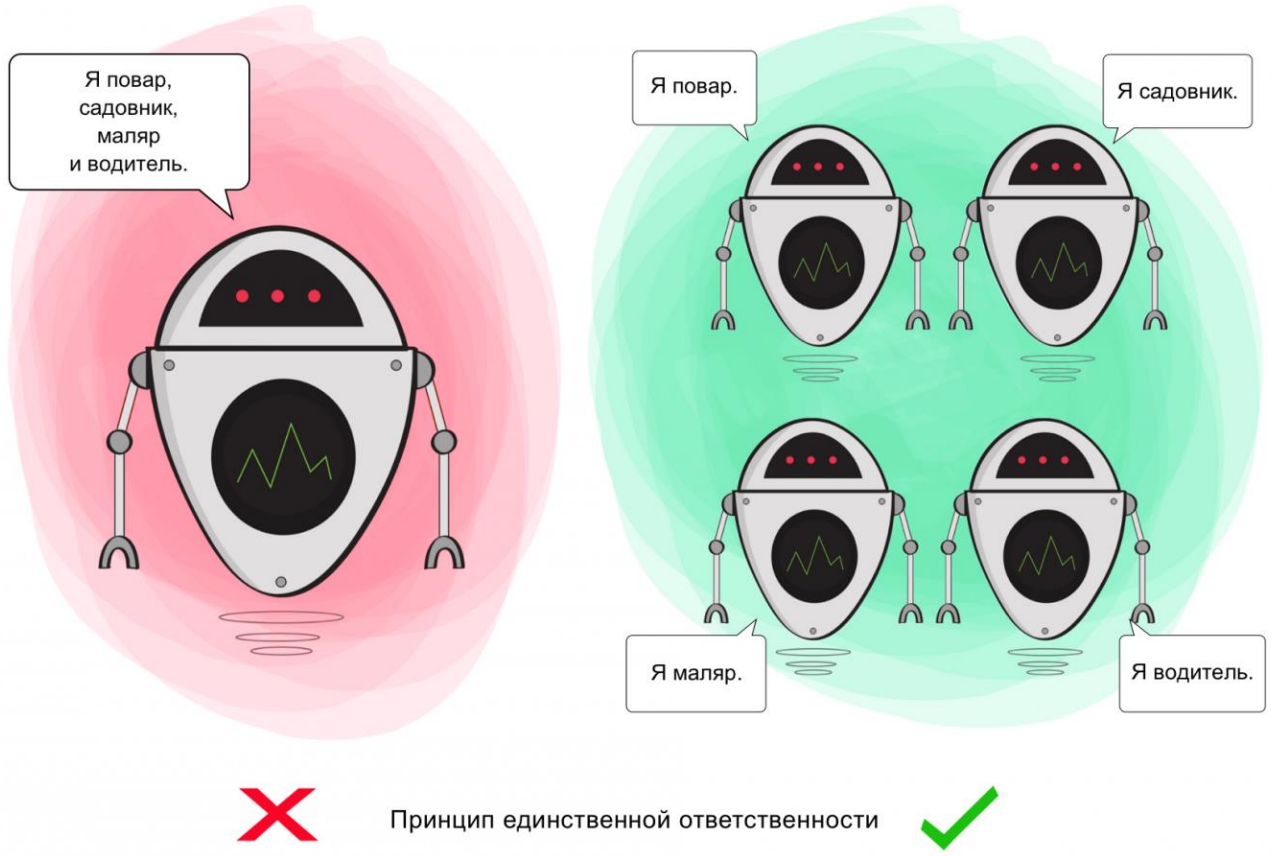
SOLID

- S – Single Responsibility (Принцип единственной ответственности)
- O — Open-Closed (Принцип открытости-закрытости)
- L — Liskov Substitution (Принцип подстановки Барбары Лисков)
- I — Interface Segregation (Принцип разделения интерфейсов)
- D — Dependency Inversion (Принцип инверсии зависимостей)

- https://habr.com/ru/company/productivity_inside/blog/505430/

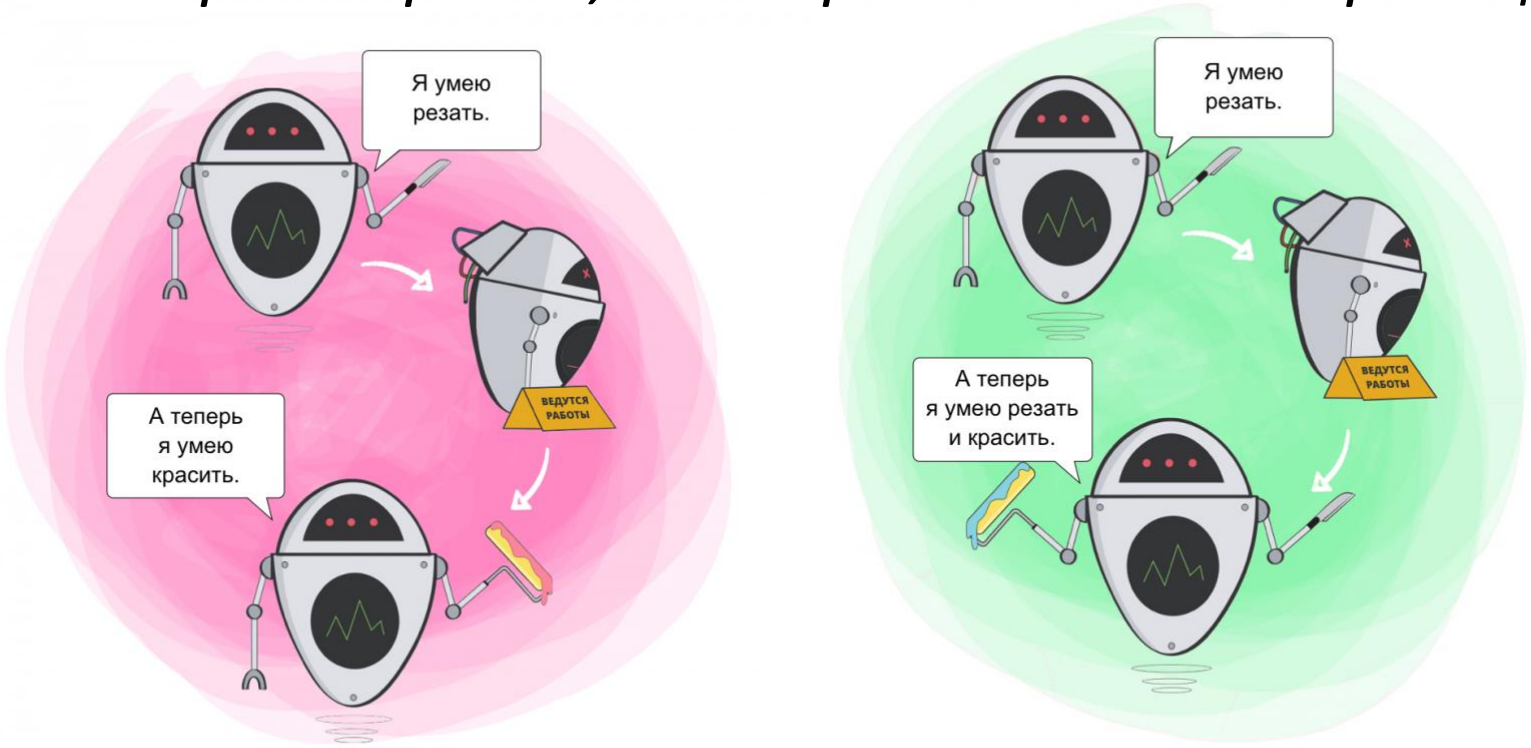
S

- *Каждый класс должен отвечать только за одну операцию*



O

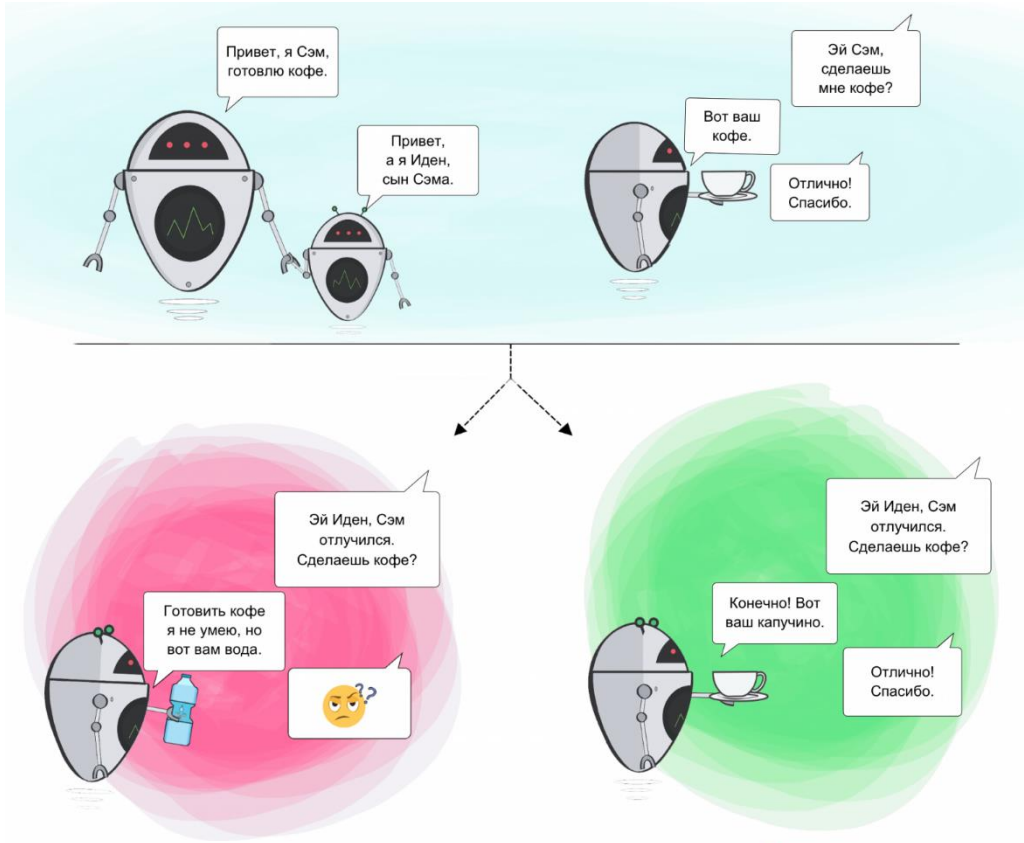
- *Классы должны быть открыты для расширения, но закрыты для модификации.*



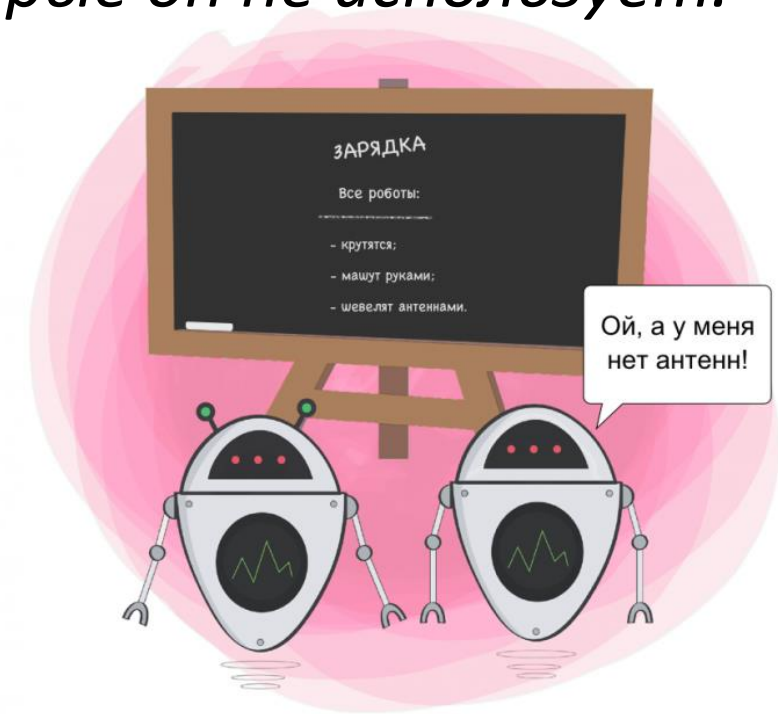
Принцип открытости/закрытости

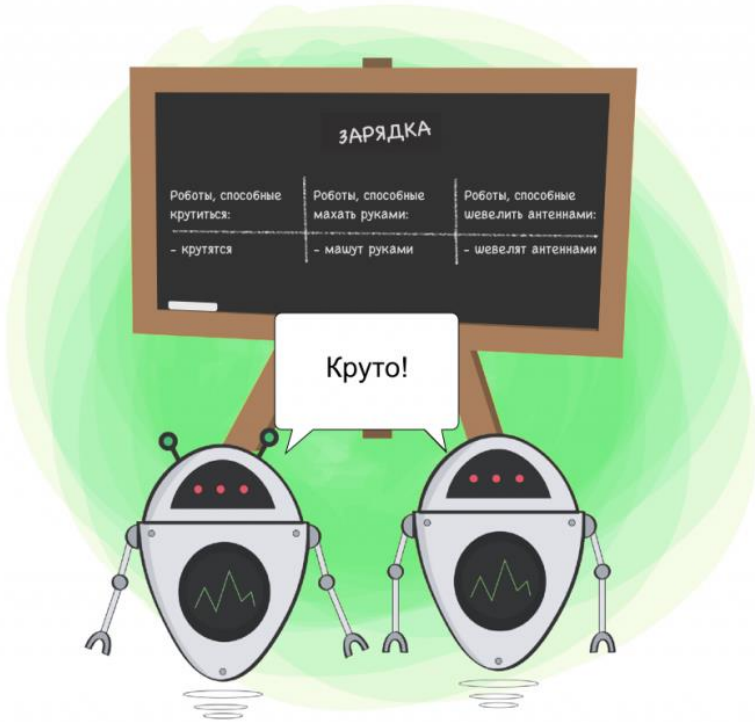
L

- *Если П является подтипом Т, то любые объекты типа Т, присутствующие в программе, могут заменяться объектами типа П без негативных последствий для функциональности программы.*



- *Не следует ставить клиент в зависимость от методов, которые он не использует.*

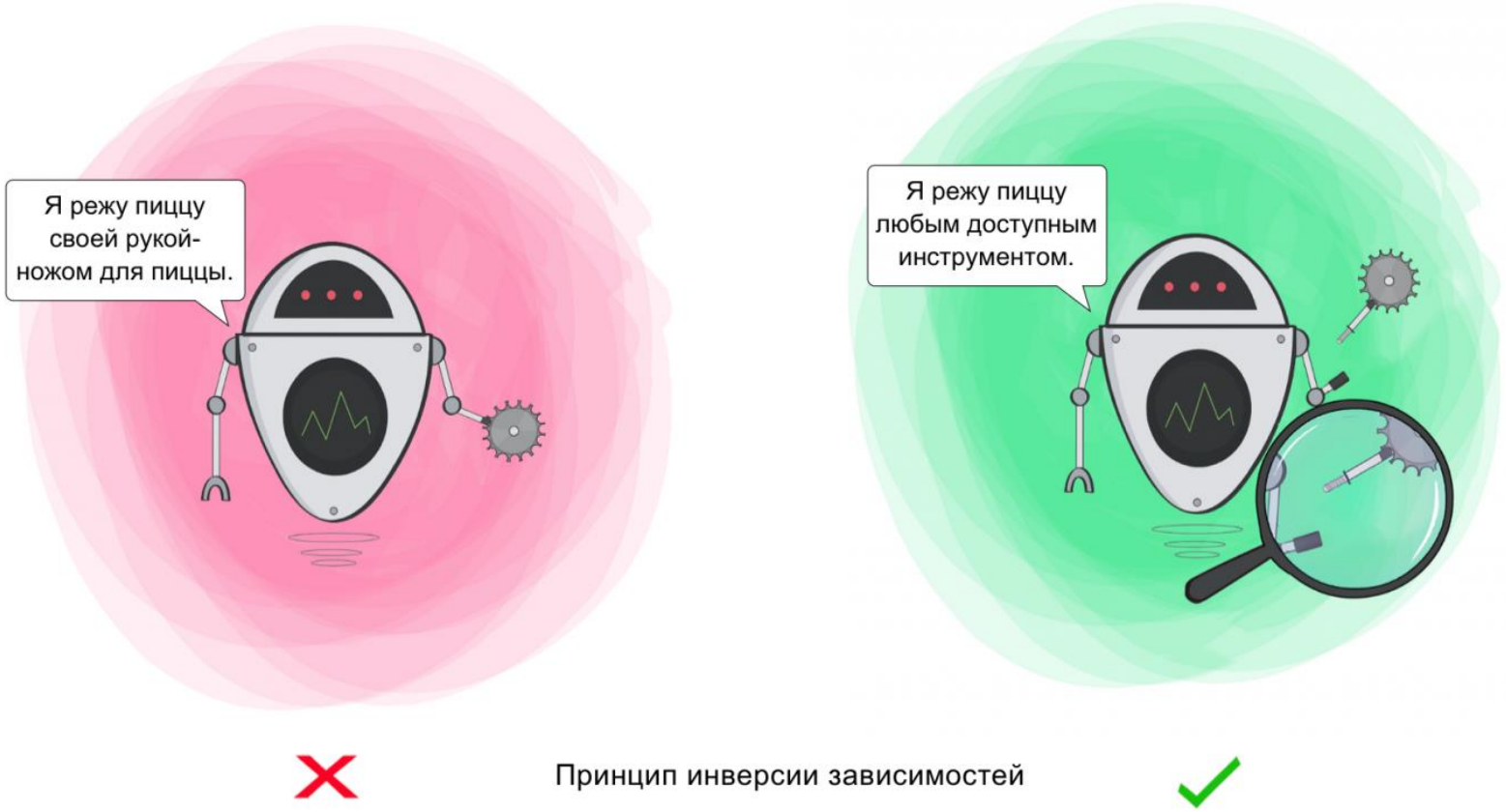




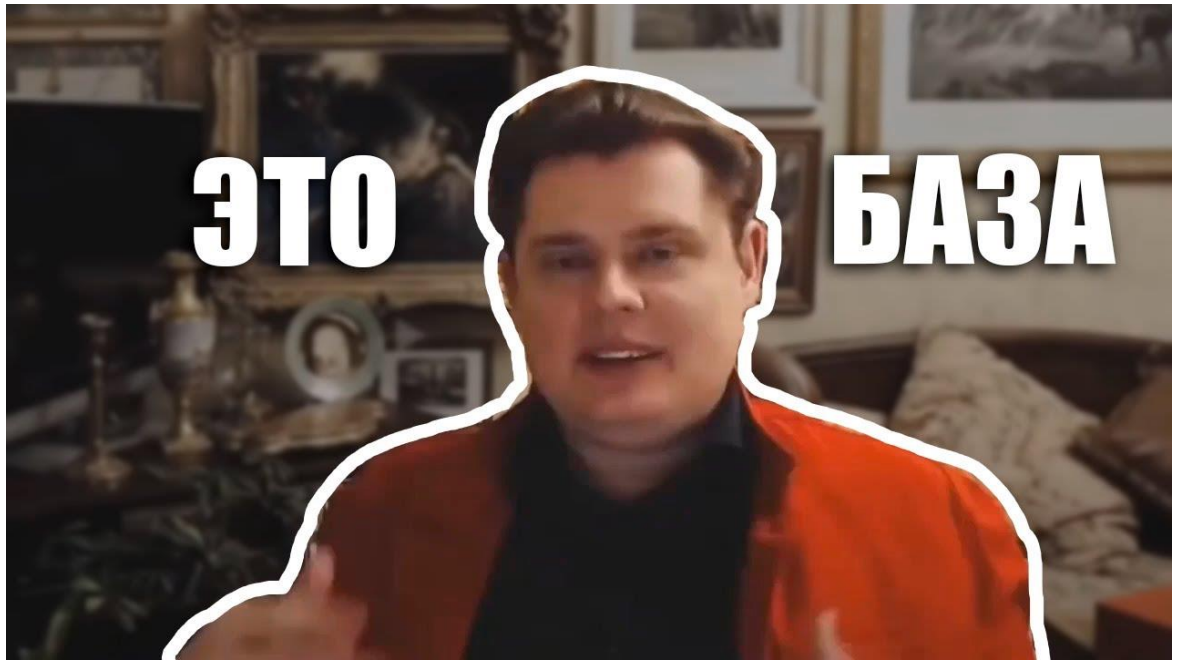
Принцип разделения интерфейсов

D

- *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.*



GRASP – 9 паттернов



Информационный эксперт (Information Expert)

- Ответственность должна быть назначена тому, кто владеет максимумом необходимой информации для исполнения — информационному эксперту.

Создатель (Creator)

- Проблема: Кто отвечает за создание объекта некоторого класса A?
- Решение: Назначить классу B обязанность создавать объекты класса A, если класс B:
  - содержит(contains) или агрегирует(aggregate) объекты A;
  - записывает(records) объекты A;
  - активно использует объекты A;
  - обладает данными для инициализации объектов A
- Можно сказать, что шаблон «*Creator*» - это интерпретация шаблона «*Information Expert*» (смотрите пункт № 1) в контексте создания объектов.

Контроллер (Controller)

- Отвечает за операции, запросы на которые приходят от пользователя, и может выполнять сценарии одного или нескольких вариантов использования (например, создание и удаление);
- Не выполняет работу самостоятельно, а делегирует компетентным исполнителям;
- Использование контроллеров позволяет отделить логику от представления, тем самым повышая возможность повторного использования кода.

Низкая связанность (Low Coupling) и Высокое зацепление (High Cohesion)

- Мера связности модулей определяется количеством информации которой располагает один модуль о природе другого.
- В свою очередь, мера зацепления модуля определяется степенью сфокусированности его обязанностей.

Полиморфизм

- Знаем уже из ооп и солид
- Шаблон полиморфизм позволяет обрабатывать альтернативные варианты поведения на основе типа.

Чистая выдумка (Pure Fabrication)

- Выделяем код взаимодействия не в их поведение (если они не подходят логически, не вписываются в предметную область классов), а в отдельный выдуманный класс
- Шаблон чистая выдумка позволяет ввести в программную среду дополнительный класс (не отражающий реальной сущности из предметной области) и наделить его требуемыми обязанностями.

Перенаправление

- Шаблон перенаправление реализует низкую связность между классами, путем назначения обязанностей по их взаимодействию дополнительному объекту — посреднику.
- Если честно, мне кажется, что это частный и наиболее частый случай чистой выдумки

Устойчивость к изменениям

- Сущность шаблона «устойчивый к изменениям» заключается в устранении точек неустойчивости, путем определения их в качестве интерфейсов и реализации для них различных вариантов поведения.
- Шаблон защищает элементы от изменения другими элементами (объектами или подсистемами) с помощью вынесения взаимодействия в фиксированный интерфейс, через который (и только через который) возможно взаимодействие между элементами

Бонусный шаблон (комментарий с хабра)

- Паттерны Головного Мозга.

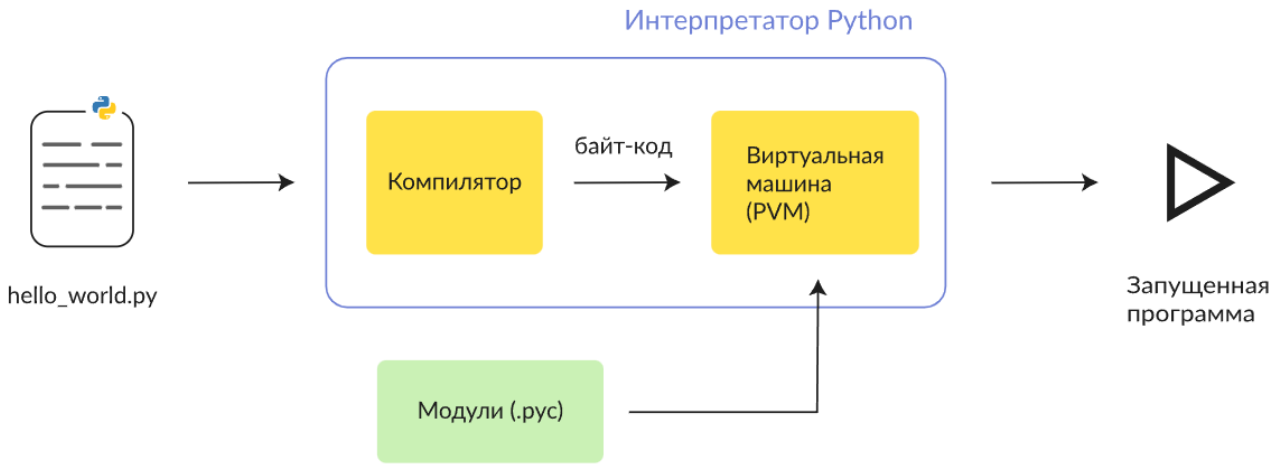
  Ситуация когда субъект воспринимает рисунок из стрелочек и прямоугольничков, как дарованное с небес откровение о мироустройстве. И пользуясь напильником и топором пытается впихнуть любую ситуацию в рамки одной или нескольких таких картинок, забывая об особенностях предметной области и уже существующей кодовой базы. Плодит тонны вспомогательных классов (слався Адаптер!) и прочего гудрона. Речь бессвязная («бла-бла-бла стратегия бла-бла-бла итератор бла-бла-бла фабрика»).

Диаграмма для домашки

- Рисуем!

Что такое питон

- Python – интерпретируемый язык
- Но точнее – это интерпретатор компилирующего типа



- PVM – по сути, просто большой цикл, который выполняет перебор инструкций в байт-коде и выполняет соответствующие им операции.

Скриптовость питона

- Питон – весьма высокоуровневый язык.
- На чистом питоне не получится написать высокоэффективную систему
- Но он очень хорошо подходит для скриптов и быстрого прототипирования
- Вы используете библиотеки, которые в свою очередь написаны эффективно и не на питоне (либы, в которых важна скорость, например numpy, pandas)
- Поэтому делать лучше все средствами пакета, как можно меньше описывая логику на чистом питоне.

Что такое объект (все – объект, а точнее ссылка на него), Garbage Collector

- В питоне абсолютно все – это на самом деле объекты класса PyObject.
- Все классы – наследуются от PyObject (и сами являются в свою очередь объектами PyObject)
- А если точнее – все не объекты, а ссылки на эти объекты
- При удалении объекта – удаляется не объект, а ссылка на него
- Когда ссылок становится 0, только в этом случае происходит удаление объекта из оперативной памяти Garbage Collector-ом
- GC также умеет очищать зацикленные ссылки, посредством поколений и простого алгоритма

Документирование

- Нужно документировать код (докстрингами описывать функции, ее параметры, return-ы, особенности)
- Нужно документировать API (для других людей ваш сервис черный ящик, с которым можно взаимодействовать только по API) сваггером
- Нужно документировать запуск кода (устновка среды, запуск приложения, самые важны особенности настройки) в README (для запуска проекта можно использовать Makefile)
- Выбавет полезно документировать код/проект более общим языком в вашей системе документирования типа Confluence

Гугление, сурсы

- Гуглите по ключевым словам, и желательно на английском
- Только с опытом приходит понимание, как правильно сформировать запрос к гуглу
- Например, `flask-restx swagger json list`, чтобы понять, как отобразить во сваггере в json-поле ввода список.
- Не чурайтесь лезть в source файлы инструмента. Это может дать достаточно много полезной инфы.