# Введение

*СЛАЙД 1-2*

Сегодня у нас на повестке дня антипаттерны. Задание в этом уроке будет несложное. А в следующий раз мы поговорим о микросервисах и задания там не будет вообще. Можно будет доделать долги.

*СЛАЙД 3-4*

Всегда ли хорошо использовать шаблоны? Наверно нет. Они устаревают, можно применить шаблон не в той ситуации и т.д.

*СЛАЙД 5*

Изучив шаблоны, мы можем пытаться насильно применить их к своей задаче. В нашем случае мы делали это, но в большей степени для отработки навыков. В реальной жизни подгонять результат под паттерн – не очень хорошо.

Применять шаблон ради только применения – тоже не очень хорошо.

Лучше делать так: у нас есть типовая задача, под которую мы ищем шаблон (паттерн) и успешно применяем на практике.

*СЛАЙД 6*

Т.е. мы отталкиваемся от задачи и думаем подходит нам некоторый паттерн или нет и применяем.

Мы можем не знать паттерны, но это не означает, что такой человек – плохой разработчик. Но знать их и уметь применить – это хорошо. Паттерны модернизируют наш процесс мышления.

*СЛАЙД 7*

Антипаттерны – это паттерны наоборот или «вредные советы». Чем они хороши? Иногда могут быть даже более полезны, чем паттерны.

*СЛАЙД 8*

Антипаттерны условно можно разделить на три группы. Они относятся к разным областям процесса разработки.

**Антипаттерны разработки (development antipatterns).** Самые простые и говорят о том, как мы пишем код.

**Архитектурные антипаттерны (architectural antipatterns).** Их труднее обнаружить. И это скорее сделает опытный разработчик-архитектор. Они в архитектуре – в программе в целом и ее частях.

**Организационные антипаттерны (managerial antipatterns).** Это немного про другое – про организацию командной разработки.

*СЛАЙД 9*

Нас интересуют прежде всего антипаттерны блоков 1 и 2.

*СЛАЙД 10*

Начнем с антипаттернов разработки:

**В коде.** Т.е. что мы написали в коде неверно.

**В ООП.** Имеется в виду про классы и методы, как они друг с другом взаимодействуют.

**Методологические.** Это каким образом мы разрабатываем программу.

*СЛАЙД 11-12*

Теперь подробнее про каждый блок. Начнем с антипаттернов в коде.

*СЛАЙД 13*

**Магические числа (Magic numbers).**

**Листинг 1. magic\_number.py**

|  |
| --- |
| **def** draw\_sprite(arg1, arg2, arg3):  **pass** draw\_sprite(53, 320, 240)  SCREEN\_WIDTH = 640 SCREEN\_HEIGHT = 480 SCREEN\_X\_CENTER = SCREEN\_WIDTH / 2 SCREEN\_Y\_CENTER = SCREEN\_HEIGHT / 2 SPRITE\_CROSSHAIR = 53  draw\_sprite(SPRITE\_CROSSHAIR, SCREEN\_X\_CENTER, SCREEN\_Y\_CENTER) |

В этом примере когда мы передаем в функцию параметры таким образом: draw\_sprite(53, 320, 240), то можно говорить о магических числах.

В этом случаем самым простым решением будет сделать эти параметры константами.

А иначе при модернизации задачи придется делать правки во всех функциях, где есть указание таких параметров.

**Спагетти-код (Spaghetti Code).**

Код похож на миску лапши. У нас есть некоторый запутанный код, который непонятно каким образом работает. Запутанный код, в котором мы не можем разобраться. Возникает вопрос, почему так сложно и как сделать проще.

**Лазанья-код (Lasagna Code).**

Лазанья – это многослойный пирог. В коде слишком много слоев. Чрезмерное расслоение программы. Если у нас каждый слой независимый, то проблем нет. Но если слои связаны, то может возникнуть необходимость каскадного изменения слоев. Т.е. меняется один слой, а за ним – все остальные. Аналогичная ситуация наблюдается в случае чрезмерного наследования. В результате с таким кодом никто не захочет работать дальше.

**Слепая вера (Blind Faith).**

Простой антипаттерн. Его суть – недостаточная проверка входных данных. Например, в функции или классе, или где-то не поставили валидатор. Мы доверяем пользователю, что он введет правильные данные. Это ошибка.

*СЛАЙД 14*

*Шифрованный код (Cryptic code).*

**Листинг 2. cryptic\_code.py**

|  |
| --- |
| a = 10 b = 90 n10 = 100 abc = 5 python = 0 pthn = 0  *# 1. поленились придумать названия # 2. не смогли придумать название  # c = a + b # a = 100 # # # def a(b, c): # return b + c* |

Это тот случай, когда переменные называются неправильно. Это и есть шифрокод. Работает примерно, как и магические числа. Почему так происходит? Мы поленились придумать названия или не смогли придумать названия.

**Жёсткое кодирование (Hard code).**

Этот антипаттерн знают все. Конкретные данные, которые мы «забили» в программу.

**Листинг 3. hard\_code.py**

|  |
| --- |
| url = **'http://ya.ru'** path = **'C://...'** |

Программа становится негибкая. Часто так получаются костыли, когда у нас нет времени.

**Мягкое кодирование (Soft code).**

Встречается у любителей все сделать супергибко.

Оптимальный вариант – когда мы меняем программу, но незначительно!

**Поток лавы (Lava flow)**

**Листинг 4. lava\_flow.py**

|  |
| --- |
| *# Одна часть кода написана с учетом ошибки в другой части кода* **def** cities():  result = [**'gelendzik'**, **'piter'**, **'tula'**, 1, **'perm'**, **'samara'**]  **return** result   **def** double():  city\_list = cities()  **return** [item \* 2 **for** item **in** city\_list]   **print**(double()) |

Это когда ошибки в одних частях кода несут в себе ошибки в других частях. Т.е. одна часть кода написана с учетом ошибки в другой части кода. В этом примере первый разработчик допустил ошибку в функции, второй – написал свою с учетом этой ошибки, третий – с учетом ошибки второго и т.д.

Теперь порассуждаем, какие из этих антипаттернов встречаются чаще всего! Проверьте у себя. Обычно у какого-то разработчика некоторый антипаттерн встречается чаще всего. Например, часто это магические числа, шифрокод, хард код.

И попробуйте понять, почему у вас встречаются найденные антипаттерны.

Давайте теперь рассмотрим чисто питоновские антипаттерны, но их немного, потому как Питон – очень продвинутый язык.

**Листинг 4. python\_antipatterns.py**

|  |
| --- |
| **import** os   **class** SomeException(Exception):  **pass** *# Bad* **def** some\_func(arg):  **if not** arg:  **return** None  **return** arg   res = some\_func(None)  **if** res **is not** None:  *# go on* **pass** *# Good* **def** some\_func(arg):  **if not** arg:  **raise** SomeException(**'no arg!'**)  **return** arg   **try**:  res = some\_func(None)  *# go on* **except** SomeException:  *# handle it* **pass** nums = [1, 2, 3]  *# Bad* nums\_squares = dict([(elem, elem \* 2) **for** elem **in** nums])  *# Good* nums\_squares2 = {elem: elem \* 2 **for** elem **in** nums}  *# Bad* **if** os.path.exists(**"some\_file.txt"**):  os.unlink(**"some\_file.txt"**)  *# Good* **try**:  os.unlink(**"some\_file.txt"**) **except** OSError:  **pass** *# Определение исключительной ситуации # Это исключение?   # Bad* a = 15 b = 85  \_tmp = a  a = b + 2 b = \_tmp - 4  *# Good* c = 15 d = 85  c, d = d + 2, c - 4  data = **'leo:25'** name = data.split(**':'**)[0]  name, \_ = data.split(**':'**)   nums = [10, 20, 30]  *# Bad:* nums\_proceed = map(**lambda** x: x \* 2, nums)  *# Good* nums\_proceed2 = [x \* 2 **for** x **in** nums]  *# Bad* nums\_filtered = filter(**lambda** x: x < 10, nums)  *# Good* nums\_filtered2 = [x **for** x **in** nums **if** x < 10] |

Возврат из ф-ции переменных разных типов:

*# Bad***def** some\_func(arg):  
 **if not** arg:  
 **return** None  
 **return** arg

А как правильно:

**class** SomeException(Exception):  
 **pass***# Good***def** some\_func(arg):  
 **if not** arg:  
 **raise** SomeException(**'no arg!'**)  
 **return** arg

Создание словарей более сложным способом:

*# Bad*nums\_squares = dict([(elem, elem \* 2) **for** elem **in** nums])  
  
*# Good*nums\_squares2 = {elem: elem \* 2 **for** elem **in** nums}

Обработка ошибок и исключительных ситуаций через if

*# Bad***if** os.path.exists(**"some\_file.txt"**):  
 os.unlink(**"some\_file.txt"**)  
  
*# Good***try**:  
 os.unlink(**"some\_file.txt"**)  
**except** OSError:  
 **pass**

Как узнать, где if, где try?

Нужно задать вопрос.

*# Определение исключительной ситуации  
# Это исключение?*

Отдельное присваивание значений вместо распаковки

*# Bad*a = 15  
b = 85  
  
\_tmp = a  
  
a = b + 2  
b = \_tmp - 4  
  
*# Good*c = 15  
d = 85  
  
c, d = d + 2, c - 4  
  
data = **'leo:25'**

**# плохо**name = data.split(**':'**)[0]  
**# хорошо**  
name, \_ = data.split(**':'**)

Использование map и фильтров вместо списковых включений

*# Bad:*nums\_proceed = map(**lambda** x: x \* 2, nums)  
  
*# Good*nums\_proceed2 = [x \* 2 **for** x **in** nums]

*# Bad*nums\_filtered = filter(**lambda** x: x < 10, nums)  
  
*# Good*nums\_filtered2 = [x **for** x **in** nums **if** x < 10]

Это не жесткий антипаттерн, просто списковое включение быстрее.

*СЛАЙД 15-16*

Переходим к антипаттернам в ООП

**Боязнь размещать логику в объектах предметной области**

Самый простой пример – из Django.

Должно быть – толстые модели, тонкие контроллеры, тупые шаблоны.

**Божественный объект (God Object)**

Это какой-то класс или ф-ция, нарушающие SRP. Какой-то большой класс, отвечающий буквально за все! Встречается, когда мы не знаем, как разделить методы класса.

И второй случай, когда. Например, у нас был нормальный изолированный класс, например, order. Но по каким-то причинам растет вместе с нашей системой и становится огромным.

**Полтергейст (Poltergeist)**

*Полтергейст* — недолговечный объект, используемый для инициализации или для вызова методов в другом, более постоянном классе. Типичная причина этого антипаттерна — плохой дизайн.

Объект, который, то есть, то нет. И мы не можем на 100% всегда воспроизвести ошибку.

Например, у нас есть поток, в котором создается некоторый объект, который взял и удалился сборщиком мусора. А может и не удалиться. Этот поток запускается в другом потоке, и мы не знаем, когда сработает этот сборщик мусора. Часто встречается в потоках и мультипроцессах.

Решение: применить очереди сообщений.

**Сплошное одиночество (Singletonitis)**

Появился, когда всем полюбился синглтон. Синглтоны не по назначению, т.е. везде синглтоны.

**Приватизация (Privatization).**

**Листинг 5. privatization.py**

|  |
| --- |
| **class** A:   **def** \_\_init\_\_(self):  # модификаторы доступа  *# private, protected, public* self.\_\_a = 0  self.\_a = 0  self.a = 0 |

Это когда мы используем двойное подчеркивание.

В чем здесь проблема? Приватные атрибуты не доступны в наследниках класса. Лучше исп-ть *protected*

*СЛАЙД 17-18*

Переходим к методологическим антипаттернам. Т.е. что мы делаем не так в процессе разработки.

**Программирование методом копирования — вставки (Copy — Paste programming)**

Т.е. когда мы берем старую задачу и пытаемся, думаем, как ее подогнать к новой задаче.

Желательно сначала подумать о задаче, а если и копируем, то делать рефакторинг.

Есть мнение, что большинство кода – со стак оверфлоу, но это не относится к этому антипаттерну.

**Золотой молоток (Golden Hammer)**

Штука, которая решает все задачи. Уверенность в том, что найденное решение универсально.

Может относится, например, к фреймворку, языку – Django может все, Python может все. НАМ же в идеале нужно идти от задачи к коду, а не наоборот.

**Фактор невероятности (Improbability factor)**

Предположение невозможности того, что сработает известная ошибка. Не учитываем возможность случайности. Из этого возникает паттерн «Слепая вера».

*СЛАЙД 19*

**Преждевременная оптимизация (Premature optimization)**

Из него «вытекает» Soft Code. Мы что начали писать и начали сразу оптимизировать.

Как избежать? Использовать правило двух: мы что-то сделали, потом сделали еще раз и заметили дубль. И когда это повторяется в третий раз, выполняем рефакторинг.

**Изобретение велосипеда (Reinventing the wheel)**

Например, изобретаем свой фреймворк. Это хорошо только в учебных целях.

**Изобретение квадратного колеса (Reinventing the square wheel)**

Написали свой фреймворк, но получилось хуже, чем было.

*СЛАЙД 20-21*

Переходим к архитектурным антипаттернам.

**Инверсия абстракции (Abstract Inversion)**

Сокрытие части функциональности от внешнего использования в надежде на то, что никто не будет её использовать.

Это вариант паттерна Приватизация.

Такое явление считается антипаттерном, только если скрывается действительно нужная пользователям функциональность, что заставляет их реализовывать её самостоятельно в классах-наследниках. Антипаттерн — ошибка проектирования и недостаточного анализа требований к классу.

Решение: проанализировать требования к классу и наделить все необходимые для работы методы соответствующими префиксами ('\_','\_\_').

**Большой комок грязи (Big ball of mud)**

Система с нераспознаваемой структурой.

Еще один вариант *Спагетти-кода*, на этот раз архитектурный. Фактически это отсутствие архитектуры как таковой. Система без структуры тяжело поддаётся рефакторингу, плохо поддерживается и провоцирует попадание в ошибочные состояния, т. к. их количество сложно отследить.

Решение: структурировать код, применить по необходимости известные шаблоны проектирования.

**Затычка на ввод (Input kludge)**

Забывчивость в спецификации и выполнение поддержки возможного неверного ввода.

В сленге разработчиков есть понятие «защита от дурака» — защита от ввода некорректных или неожиданных данных. Она позволяет избежать случайных ошибок при вводе некорректных данных и становится важным компонентом информационной безопасности, блокирующим попытки взлома или внедрения вредоносного кода.

Суть антипаттерна состоит не в отсутствии такой защиты, а в недостаточной её проработанности. Это приводит к потенциальной «дыре» в безопасности (особенно актуально для веб-приложений) и возможности внесения некорректных данных.

Решение: всегда составлять строгую спецификацию на ввод данных извне, валидировать и обезвреживать введённые данные.

**Волшебная кнопка (Magic pushbutton)**

Похоже на нарушение принципа – толстая модель, тонкий контроллер. Или, например, не в бизнес-логике работаем, а в интерфейсе.

Выполнение результатов действий пользователя в виде неподходящего (недостаточно абстрактного) интерфейса.

Как правило, неподходящее место — это обработчик события взаимодействия с каким-либо интерактивным элементом приложения, например, с кнопкой. Отсюда и появилось название антипаттерна.

Ошибка состоит в том, что бизнес-логика приложения помещается в место, которое не должно её содержать. Если говорить об архитектуре MVC, можно представить, что логика находится не в *Модели*, а в *Представлении*, что нарушает весь смысл использования архитектурного шаблона MVC.

Решение: бизнес-логика размещается в соответствующем разделе. Взаимодействие пользователя с приложением должно быть тонким и прозрачным и не должно содержать бизнес-логики. Вместо этого должны вызываться соответствующие сервисы.

**Членовредительство (Mutilation)**

Излишнее «затачивание» объекта под определённую очень узкую задачу таким образом, что он не способен будет работать с никакими иными, пусть и очень схожими задачами.

Принцип DRY провозглашает идею создания общеиспользуемых решений. Однако чрезмерная сфокусированность классов на своей задаче может навредить этому принципу. Если существует класс похожих задач в проекте, следует предусмотреть такое решение, которое было бы общим для них.

Решение: переписать класс без жёсткой привязки к конкретной задаче (по возможности).

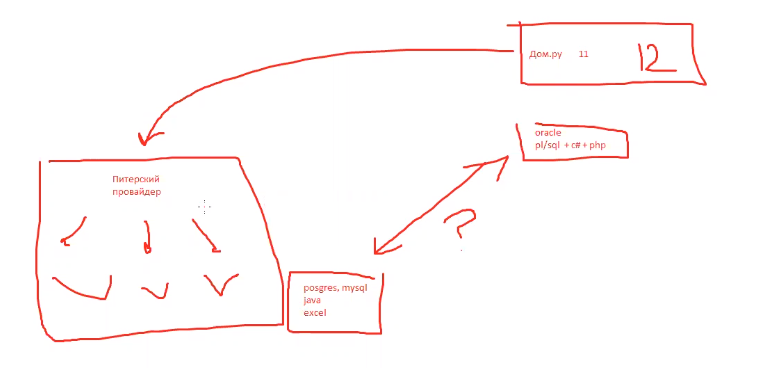
*СЛАЙД 22*

**Дымоход предприятия (Stovepipe Enterprise)**

Несогласованность архитектур корпоративных приложений, которая ухудшает или сводит на нет адаптивность, повторное использование и функциональную совместимость. Этот антипаттерн сравнивают с дымоходом печи, который необходимо постоянно ремонтировать. Причём при ремонте используются любые подручные средства. В результате получается что-то очень специфическое.

Ситуация может возникнуть, когда происходит объединение компаний, в каждой из которых использовалась своя архитектура.

Решение: создание эталонной модели стандартов, создание общей операционной среды, системных профилей, координирующих использование продуктов и стандартов.

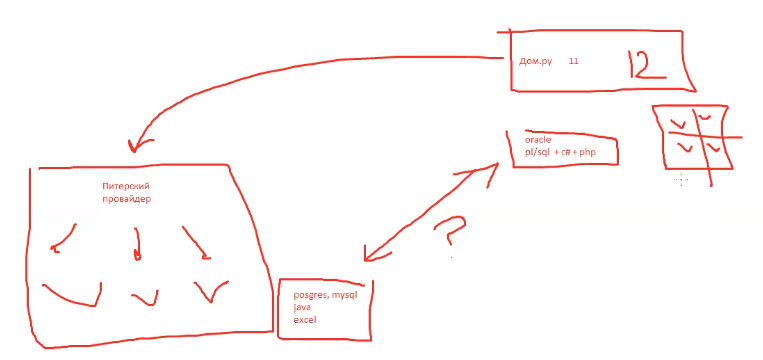


Если подружить эти системы, будет дымоход предприятия.

**Дымоход системы (Stovepipe System)**

Антипаттерн аналогичен Stovepipe Enterprise, но рассматривается несогласованность в рамках одной системы, а не набора систем. Как правило, это устаревшая (legacy) система, не удовлетворяющая новым потребностям бизнеса. Одна из проблем — отсутствие подсистемных абстракций (в Stovepipe Enterprise это межсистемные абстракции). Другая — взаимосвязь подсистем по типу «точка-точка». Реализация системы хрупка из-за большого количества неявных зависимостей от конфигурации, особенностей установки и её состояния. При попытке расширения появляются новые связи «точка-точка», и ситуация усугубляется.

В чем идея?



То же самое, что и дымоход предприятия, но в рамках отделов.

**Путаница (Jumble)**

Смешивание горизонтальных и вертикальных элементов дизайна, приводящее к нестабильности архитектуры.

Вертикальные элементы дизайна должны зависеть от конкретной программной реализации. Горизонтальные элементы дизайна — общие. Если этот порядок нарушается, получаем *Путаницу*. Ухудшаются возможности повторного использования и надёжность. Из-за вертикальных элементов появляются программные зависимости, что ухудшает расширяемость.

Решение: идентифицировать элементы горизонтального дизайна и перенести их на отдельный уровень архитектуры. Если они — абстракции конкретных реализаций подсистем, то можно добавить вертикальные элементы как расширения для дополнительного функционала. Рекомендуется добавить динамические элементы (метаданные) в архитектуру и обеспечить их взаимодействие со статическими элементам (горизонтальными и вертикальными). При правильном балансе между ними мы получим хорошо структурированный, расширяемый продукт. Иногда такой подход называют Horizontal-Vertical-Metadata (HVM) pattern.

Например, в модуле работы с заказами начинаем выполнять операции с байтами. Это и есть путаница.

