*Урок 8. Антипаттерны*

Оглавление

[Введение 2](#_Toc91606988)

[Антипаттерны разработки 3](#_Toc91606989)

[В коде 3](#_Toc91606990)

[Магические числа (Magic numbers) 3](#_Toc91606991)

[Спагетти-код (Spaghetti Code) 4](#_Toc91606992)

[Лазанья-код (Lasagna Code) 4](#_Toc91606993)

[Слепая вера (Blind Faith) 4](#_Toc91606994)

[Шифрованный код (Cryptic code) 4](#_Toc91606995)

[Жёсткое кодирование (Hard code) 5](#_Toc91606996)

[Мягкое кодирование (Soft code) 5](#_Toc91606997)

[Поток лавы (Lava flow) 5](#_Toc91606998)

[Специализированные антипаттерны в Python 5](#_Toc91606999)

[Возврат из функции переменных разных типов 7](#_Toc91607000)

[Создание словарей более сложным способом 7](#_Toc91607001)

[Обработка ошибок и исключительных ситуаций через if 7](#_Toc91607002)

[Отдельное присваивание значений вместо распаковки 8](#_Toc91607003)

[Использование map и фильтров вместо списковых включений 8](#_Toc91607004)

[В ООП 8](#_Toc91607005)

[Боязнь размещать логику в объектах предметной области 8](#_Toc91607006)

[Божественный объект (God Object) 8](#_Toc91607007)

[Полтергейст (Poltergeist) 9](#_Toc91607008)

[Сплошное одиночество (Singletonitis) 9](#_Toc91607009)

[Приватизация (Privatization) 9](#_Toc91607010)

[Методологические антипаттерны 9](#_Toc91607011)

[Программирование методом копирования – вставки (Copy – Paste programming) 9](#_Toc91607012)

[Золотой молоток (Golden Hammer) 9](#_Toc91607013)

[Фактор невероятности (Improbability factor) 10](#_Toc91607014)

[Преждевременная оптимизация (Premature optimization) 10](#_Toc91607015)

[Изобретение велосипеда (Reinventing the wheel) 10](#_Toc91607016)

[Изобретение квадратного колеса (Reinventing the square wheel) 10](#_Toc91607017)

[Архитектурные антипаттерны 10](#_Toc91607018)

[Инверсия абстракции (Abstract Inversion) 10](#_Toc91607019)

[Большой комок грязи (Big ball of mud) 10](#_Toc91607020)

[Затычка на ввод (Input kludge) 11](#_Toc91607021)

[Волшебная кнопка (Magic pushbutton) 11](#_Toc91607022)

[Членовредительство (Mutilation) 12](#_Toc91607023)

[Дымоход предприятия (Stovepipe Enterprise) 12](#_Toc91607024)

[Дымоход системы (Stovepipe System) 13](#_Toc91607025)

[Путаница (Jumble) 13](#_Toc91607026)

# Введение

Сегодня у нас на повестке дня антипаттерны. Задание в этом уроке будет несложное. А в рамках заключительного мы поговорим о микросервисах и задания там не будет вообще. Можно будет выполнить и сдать долги.

Мы в течение всего курса изучали паттерны, т.е. шаблоны. Но всегда ли хорошо использовать шаблоны? Ведь они устаревают, плюс можно применить шаблон не в той ситуации и т.д.

Изучив паттерны, мы можем пытаться целенаправленно применить их к своей задаче. В нашем случае мы в какой-то степени делали это, но больше для отработки навыков. В реальной жизни подгонять задачу под паттерн – не самая хорошая идея.

Применять шаблон ради только применения – тоже неправильно.

Т.е. мы должны отталкиваться от задачи, а не от паттерна и анализировать подходит нам некоторый паттерн или нет и в случае положительного ответа применять его.

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

**Антипаттерны** – это паттерны наоборот или «вредные советы». Чем они хороши? Иногда знание их и умение их разглядеть в проекте могут быть даже более полезными, чем умение применять паттерны.

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.

Нас интересуют прежде всего антипаттерны блоков 1 и 2.

# Антипаттерны разработки

Они подразделяются на:

**В коде.** Т.е. как не надо писать код.

**В ООП.** Имеется в виду про классы и методы, неправильно организовано взаимодействие их друг с другом.

**Методологические.** Это каким образом мы разрабатываем программу, неверные практики поведения разработчика.

## В коде

### Магические числа (Magic numbers)

**Листинг 1. Урок 8. Коды к уроку/magic\_numbers.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, т.к. визуально непонятно что означает каждый из параметров.

В этом случаем самым простым решением будет сделать параметры константами.

А иначе при модернизации задачи придется делать правки во всех функциях, где есть указание на такие параметры.

### 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 не поставили валидатор. Мы доверяем пользователю, что он введет правильные данные. Это ошибка.

### Шифрованный код (Cryptic code)

**Листинг 2. Урок 8. Коды к уроку/cryptic\_code.py**

|  |
| --- |
| a = 10 b = 90 n10 = 100 abc = 5 python = 0 pthn = 0  *# 1. поленились придумать названия # 2. не смогли придумать название # 3. не знаем, что это антипаттерн  # c = a + b # a = 100 # # # def numbers(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. Урок 8. Коды к уроку/hard\_code.py**

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

Программа становится негибкая

### Мягкое кодирование (Soft code)

Встречается у любителей все сделать супергибко.

Оптимальный вариант – когда мы меняем программу, но незначительно.

### Поток лавы (Lava flow)

**Листинг 4. Урок 8. Коды к уроку/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, не учитывая 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, хардкод и т.д.

И попробуйте понять, почему у вас встречаются найденные антипаттерны.

## Специализированные антипаттерны в Python

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

**Листинг 5. Урок 8. Коды к уроку/python\_antipatterns.py**

|  |
| --- |
| **from** os **import** unlink **from** os.path **import** exists   **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** exists(**"some\_file.txt"**):  unlink(**"some\_file.txt"**)  *# Good* **try**:  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  a, b, c = 1, 2, 3  data = **'leo:25'** name = data.split(**':'**)[0]  name, \_ = data.split(**':'**)   nums = [10, 20, 30]  *# Bad:* nums\_proceed = map(**lambda** x: x \* 2, nums) print(list(nums\_proceed))  *# 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 функции переменных разных типов

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

**class** SomeException(Exception):  
 **pass**

**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** exists(**"some\_file.txt"**):  
 unlink(**"some\_file.txt"**)  
  
*# Good***try**:  
 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.

## 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 функция, нарушающие принцип SRP. Например, 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)

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

### Сплошное одиночество (Singletonitis)

Синглтоны не по назначению, т.е. везде синглтоны.

### Приватизация (Privatization)

**Листинг 6. Урок 8. Коды к уроку/privatization.py**

|  |
| --- |
| **class** A:   **def** \_\_init\_\_(self):  *# 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 использовать protected.

## Методологические антипаттерны

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 какой-то готовый код и пытаемся подогнать его к своей задаче.

Готовым кодом пользоваться можно, но вначале выполнить его рефакторинг, оптимизацию.

### 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».

### Фактор невероятности (Improbability factor)

Отрицание возможности возникновения ошибки. Не учитываем возможность случайности. Из этого возникает паттерн «Слепая вера».

### Преждевременная оптимизация (Premature optimization)

Из него «вытекает» Soft Code. Мы что-то начали писать и сразу занялись оптимизацией.

Как избежать? Использовать «правило двух»: мы что-то сделали, потом сделали еще раз и заметили дубль ошибки или какого-то проблемного момента. 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 существующие рабочие фреймворки.

# Архитектурные антипаттерны

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.

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 возможность указания неверных данных.

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, мы реализуем что-то не в бизнес-логике приложения, а в интерфейсе.

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.

## 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.

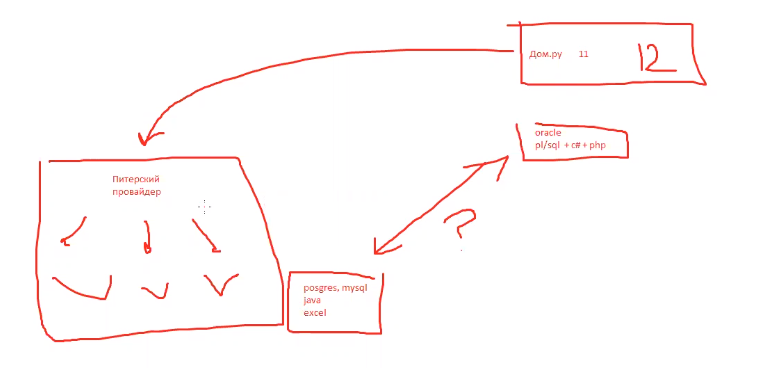
Решение: переписать класс без жёсткой привязки к конкретной задаче (по возможности).

## 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.

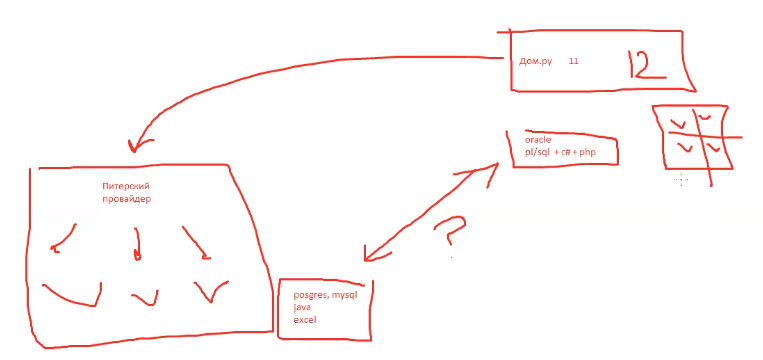
Решение: создание эталонной модели стандартов, создание общей операционной среды.



Если попытаться адаптировать эти системы друг к другу, будет дымоход предприятия.

## Дымоход системы (Stovepipe System)

Антипаттерн аналогичен Stovepipe Enterprise, но рассматривается несогласованность в рамках одной системы, а не набора систем.



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

## Путаница (Jumble)

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

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

