GeekBrains

Архитектура и шаблоны проектирования на Python

Антипаттерны

На этом уроке

Антипаттерны в коде. Антипаттерны в ООП. Антипаттерны в архитектуре. Методологические антипаттерны.

Оглавление

Основные проблемы применения шаблонов

Паттерны — это инструмент. То, как он будет работать (или «не работать») в конкретной задаче, целиком зависит от разработчика. Бездумное применение паттернов приводит к обратной проблеме— антипаттерну.

Паттерны проектирования — это попытка стандартизировать лучшие практики, найденные при анализе практических решений. В теории это может показаться полезным, однако на практике часто приводит к ненужному дублированию кода. Почти всегда более эффективно использовать хорошо продуманную реализацию, а не бездумно копировать паттерн проектирования.

У каждого паттерна есть свои плюсы и минусы. Некоторые паттерны, например Singleton, зачастую рассматриваются как антипаттерн. В иных случаях, например, при использовании Bridge или Visitor, отход от проверенного решения скорее всего породит массу проблем. Поэтому следует тщательно взвешивать пользу и вред применения того или иного паттерна, вносимого в конкретное решение.

Антипаттерны

Антипаттерн — широко распространённый подход к решению часто встречающихся задач, который обычно неэффективен и скорее всего является контрпродуктивным. По мнению «Банды четырёх», должно присутствовать как минимум два ключевых элемента, чтобы формально отличить фактический антипаттерн от простой дурной привычки, плохой практики или плохой идеи:

- Это обычно используемый процесс, структура или схема действий, которые первоначально представлялись подходящими и эффективными для решения задачи, но дают более тяжёлые последствия, чем другие проверенные методы.
- Существует ещё одно решение, которое документировано, повторяемо и доказало свою эффективность.

Антипаттерны в коде

Magic Number // Магические числа

Магическое число — целочисленная константа (реже — другой литерал), используемая для однозначной идентификации ресурса или данных. Такое число само по себе не несёт никакого смысла и может вызвать недоумение, встретившись в коде программы без соответствующего контекста или комментария. Попытка изменить его на другое, даже близкое по значению, может привести к непредсказуемым последствиям. По этой причине подобные числа иронично названы магическими.

Например, такой фрагмент будет плохим:

```
draw_sprite(53, 320, 240)
```

Не автору этого фрагмента трудно сказать, что такое 53, 320 или 240.

Правильным будет указать смысл констант — как минимум в их названии.

```
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)
```

Магические числа — это потенциальный источник ошибок в программе: если одно и то же магическое число используется в программе более одного раза, его изменение потребует правок каждого вхождения вместо одной правки значения именованной константы.

Spaghetti Code // Спагетти-код

Спагетти-код — плохо спроектированный или плохо поддерживаемый исходный код со сложной и запутанной структурой управления, с чрезмерным использованием исключений, потоков или других «неструктурированных» ветвящихся конструкций. Он назван так, потому что программный код концептуально похож на миску спагетти.

Спагетти-код может быть вызван несколькими факторами, в том числе непрерывной модификацией несколькими людьми с разными навыками и стилями программирования в течение длительного жизненного цикла.

Решение: рефакторинг исходного кода, разделение на слои и т. п.

Lasagna Code // Лазанья-код

Лазанья-код — это структура программы с несколькими чётко определенными и разделяемыми слоями, где каждый уровень кода обращается к службам в нижележащих слоях через хорошо определённые интерфейсы. Типичный пример *Лазанья-кода* — интерфейс между различными подсистемами, например, между кодом веб-приложения, бизнес-логикой и реляционной базой данных.

Однако с ростом числа слоёв:

1) Замедляется реакция системы.

2) Усложняется «вертикальная» модификация кода. Например, добавление нового поля в базу данных потянет за собой изменения во всех надлежащих слоях системы. Если их количество превосходит некоторый разумный предел для системы, вы получили *Лазанья-код*.

Решение: переработка архитектуры приложения с уменьшением количества слоёв.

Blind faith // Слепая вера

Недостаточная проверка корректности входных данных, исправления ошибки или результатов работы кода. Очень часто программист думает, что его код всегда будет в идеальных условиях, никогда не выдаст ошибки и не получит неверных входных данных или ещё чего-либо непредвиденного.

«Никому нельзя доверять, даже себе. Мне — можно! Но только до обеда». Но и не следует доводить это недоверие до паранойи, то есть приходить к антипаттерну ненужной сложности. Просто следует помнить про проверку входных данных и возможные проблемы чужого кода, который используете вы.

Cryptic Code // Шифрокод

Использование аббревиатур вместо имён, раскрывающих семантику. Лучшие практики написания кода советуют применять говорящие названия: само имя сущности говорит разработчику, какого рода данные она содержит. Аббревиатуры, как правило, не такие, поэтому их использование может затруднить понимание для других разработчиков. Не бойтесь длинных названий, все современные IDE имеют функцию подсказки и помогут вам набрать имя объекта или класса по нескольким буквам и контексту. Это куда быстрее и точнее, чем вспоминать, что значит, например, baos вместо byte_array_output_stream. Не секрет, что человек воспринимает слова не последовательно по буквам, а читает целиком слово как иероглиф (вы легко пропустите незамеченной перестановку соседних букв в одном слове). Поэтому **понимание** идентификатора byte_array_output_stream происходит **быстрее,** чем более короткой формы baos.

Решение: использовать имена, раскрывающие смысл сущности.

Hard Code // Жёсткое кодирование

Использование антипаттерна *Жёсткое кодирование* приводит к тому, что исходный код программы меняется каждый раз при изменении входных данных или желаемого формата выходных, тогда как пользователю было бы удобнее настраивать параметры работы программы, не трогая её код. Довольно часто встречается жёсткое кодирование путей и имён файлов, например, ‘C:\Windows’. Однако есть, например, системная переменная %SystemRoot%, и возможны другие решения этой проблемы.

Решение: сведения об окружении системы желательно хранить в конфигурационных файлах или использовать переменные среды. В крайнем случае — использовать пользовательский ввод, что не всегда возможно. В случае с Python мы можем использовать модули **sys** и **os**.

Soft Code // Мягкое кодирование

Soft code — это другая крайность, патологическая боязнь жёсткого кодирования, которая приводит к тому, что настраивается всё что угодно. При этом конфигурирование системы само по себе превращается в программирование.

Уравновешенный подход между этими двумя крайностями зависит от характеристик системы. Программы с закрытым исходным кодом должны быть максимально конфигурируемыми, поскольку конечный пользователь не имеет доступа к исходному коду. Внутрикорпоративное программное обеспечение может быть менее конфигурируемым.

Решение: рассмотрите ценность дополнительной гибкости, которую вы представляете для конечного пользователя, и сравните её с повышенной сложностью конфигурирования.

Lava flow // Поток лавы

Поток лавы — проблема, при которой программа, написанная в ситуации недостатка исходных условий, выводится в продакшен и продолжает развиваться.

Эксплуатация незавершённой системы приводит к необходимости поддерживать обратную совместимость (поскольку теперь от неё зависят многие другие компоненты, в том числе внешние по отношению к системе).

Зачастую выявленные ошибки не могут быть устранены, потому что другой, возможно, сторонний, код написан со знанием того, что «это ошибка» и использованием этой ошибки как нормального поведения. Сторонний код принимает эту особенность и приводит своими силами поведение системы к непротиворечивому состоянию, или (что чаще) — принимает «неверные» правила игры и делает «три пишем, пять в уме». Так ошибка логически остается в головах всех разработчиков, касающихся этой проблемы, и её уже нельзя исправить, ибо исправление приведёт к необходимости переписывать значительно большее количество кода, написанного поверх этой злополучной ошибки.

Изменения в команде разработчиков, работающих над проектом, часто усугубляют проявление этого антипаттерна. Вместо того, чтобы приводить код в порядок (заниматься рефакторингом и перепроектированием), последующие разработчики обходят ошибки, увеличивая сложность и запутанность системы. Поток лавы приводит к эрозии архитектуры или полной её деградации, значительному увеличению технического долга.

Решение: короткие циклы разработки, управление конфигурированием, постепенный рефакторинг «мёртвого» кода.

Специфические для Python антипаттерны в коде

По материалам можно выделить некоторые типичные для Python-разработчиков ошибки.

Возврат из функции переменных разных типов:

```
// 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
```

Создание словарей более сложным способом:

```
nums = [1, 2, 3]

// Bad
nums_squares = dict([(elem, elem * 2) for elem in nums])

// Good
nums_squares = {elem: elem * 2 for elem in nums}
```

Запрос разрешения вместо «прощения» — в Python принята концепция EAFP (easier to ask for forgiveness than permission):

```
import os

// Bad
if os.path.exists("some_file.txt"):
   os.unlink("some_file.txt")

// Good
try:
   os.unlink("some_file.txt")
except OSError:
   pass
```

Отдельное присвоение значений переменным вместо распаковки:

```
// Bad
a = 15
b = 85

_tmp = a
```

```
a = b + 2
b = a - 4

// Good

a = 15
b = 85

a, b = b + 2, a - 4
```

Использование map / filter там, где можно обойтись без них:

```
nums = [10, 20, 30]

// Bad
nums_proceed = map(lambda x: x * 2, nums)

// Good
nums_proceed = [x * 2 for x in nums]


// Bad
nums_filtered = filter(lambda x: x < 10, nums)

// Good
nums_filtered = [x for x in nums if x < 10]
```

Антипаттерны ООП

Anemic Domain Model // Боязнь размещать логику в объектах предметной области

Анемичная модель предметной области — это использование модели предметной области программного обеспечения, в которой объекты предметной области практически не содержат кода бизнес-логики (проверки, вычисления, бизнес-правила и т. п.).

Решение: можно избежать *Анемичной модели области*, пытаясь распределить ответственность по тем же классам, которые содержат данные.

God object (The Blob) // Божественный объект

Божественный объект — это объект, который слишком много знает или делает.

Общей методикой программирования является разделение большой проблемы на несколько меньших задач и создание решений для каждого из них. Как только будут решены меньшие проблемы, будет решена большая проблема в целом. Поэтому каждый объект для небольшой проблемы должен знать только о себе. Аналогично существует только один набор проблем, который должен решить объект, и это его собственные проблемы.

Программа, которая использует *Божественный объект*, не следует этому подходу. Большая часть общей функциональности такой программы кодируется в единый «всезнающий» объект, который

поддерживает большую часть информации обо всей программе и предоставляет большинство методов для управления этими данными. Поскольку этот объект содержит так много данных и требует очень много методов, его роль в программе становится богоподобной (всезнающей, всеохватывающей).

Вместо того, чтобы объекты внутри программы напрямую связывались между собой, они полагаются на единственный *Божественный объект*. Поскольку он тесно связан с большей частью другого кода, поддержка такой программы становится более сложной, чем при равномерно распределённой ответственности между объектами. Изменения, внесённые в объект в пользу одной части (класса) программы, могут иметь неожиданные последствия для других частей (классов) программы.

Решение: распределить ответственность между различными классами программы, более строго использовать принцип единственной ответственности.

Poltergeist // Полтергейст

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

Иногда *Полтергейсты класса* создаются, потому что разработчик ожидал потребности в более сложной архитектуре. Например, *Полтергейст* возникает, если один и тот же класс действует и как инициатор, и как клиент в паттерне Command, в то время как разработчик исходил из предположения о разнесении этой функциональности между разными классами.

Решение: удалить *Полтергейст класса*, перенеся его функциональность в вызываемый или вызывающий класс, возможно, используя наследование.

Singletonitis // Сплошное одиночество

Неуместное использование паттерна *Одиночка* приводит к росту влияния отрицательных качеств этого паттерна в системе. Он усложняет написание модульных тестов, много *Одиночек* делают код нетестируемым.

Решение: свести к минимуму использование классов *Одиночка* или ограничить их использование малым количеством классов приложения (корневых классов).

Privatization // Приватизация

Чрезмерное закрытие методов (в Python при помощи префикса `'__'`) затрудняет расширение в классах-потомках.

Решение: функционально важные методы в классе лучше объявлять как защищённые (префикс `'_'`), чтобы иметь возможность их переопределения в потомках.

Методологические антипаттерны

Copy — Paste // Программирование методом копирования — вставки

Копирование существующего кода вместо создания общих решений.

Пожалуй, один из самых старых антипаттернов. Он говорит о том, что разработчики не думали о будущем проекта (возможно, он создавался в спешке), либо об их неопытности.

Нарушение принципа DRY и, как следствие, плохая переносимость кода и невозможность его повторного использования — эта опасность справедлива, если копирование-вставка кода происходит внутри одного проекта. Если работу такого кода нужно изменить, разработчику придётся искать все места, куда этот код был скопирован, и редактировать его на месте.

Как следствие, такой код может оказаться «забытым» для модификации. К примеру, в двух из трёх мест разработчик изменил функциональность скопированного кода, а в одном — забыл. Это неизбежно создаст проблему.

Копирование кода — естественный путь распространения ошибок. Код, содержащий ошибку, при копировании принесёт её с собой.

Увеличение количества кода без увеличения его полезности: следствие — нарушение принципа KISS.

Решение: вместо копирования кода внутри проекта необходимо создавать общие решения, которые будут переиспользоваться различными компонентами. Например, вынести общую функциональность в отдельный класс *Сервис* и вызывать его методы, где это необходимо. Копирования кода из чужого проекта лучше не допускать, т. к. велик риск перенести в свой проект потенциальные проблемы или невнимательно произвести его «адаптацию», что приведёт к новым ошибкам.

Golden hammer // Золотой молоток

Уверенность в том, что найденное решение универсально.

Как правило, выражается в том, что разработчик всюду старается применить своё любимое или хорошо освоенное решение, не задумываясь о более эффективных альтернативах.

Решение: для каждой конкретной задачи необходимо проводить поиск наиболее эффективного решения, которое не обязательно будет совпадать с каким-то шаблоном или общеприменимой практикой.

Improbability factor // Фактор невероятности

Предположение невозможности того, что сработает известная ошибка.

Этот антипаттерн — родственник уже описанного антипаттерна *Слепая вера*, с той лишь разницей, что разработчик в данном случае осознаёт возможность ошибки, но тем не менее игнорирует её обработку и предотвращение.

Решение — такое же, как у *Слепой веры*: обязательно проверять все возвращаемые результаты на ошибки, даже если уверены в обратном.

Premature optimization // Преждевременная оптимизация

Оптимизация на основе недостаточной информации.

Как ни странно, чтобы выполнять оптимизацию кода, нужно иметь работающий код. Преждевременная оптимизация — это, как правило, попытка сделать эффективным код, который ещё не написан, а только проектируется. Для оптимизации важно владеть полной информацией о работе конкретного куска кода, чтобы правильно оценить все его слабые стороны. Поэтому занятия «оптимизацией в уме» чаще всего приводят к появлению странных и необоснованных решений, которые просто-напросто не работают, ведь разработчик, прежде чем проверить работоспособность кода, начал его оптимизировать.

Английский ученый и специалист в области Computer Science Энтони Хоар (автор алгоритма «быстрой» сортировки) произнес фразу, ставшую культовой среди разработчиков:

«Преждевременная оптимизация — корень всех зол».

Решение: процесс оптимизации должен происходить поэтапно и только на основе анализа уже написанного и работающего кода, а не на этапе проектирования.

Reinventing the wheel // Изобретение велосипеда

Создание с нуля вместо использования готового решения.

Вместо применения готового решения для какой-то распространённой задачи разработчик начинает «изобретать велосипед», тратя время и заново проходя через все трудности и ошибки, с которыми сталкивались разработчики уже готовых решений.

Решение: когда встречается распространённая задача, стоит поискать готовые и проверенные решения, а не изобретать собственные.

Reinventing the square wheel // Изобретение квадратного колеса

Создание плохого решения, когда существует хорошее известное решение.

Это разновидность изобретения велосипеда, с той лишь разницей, что и переизобретение решения задачи может быть хуже уже имеющегося варианта.

Архитектурные антипаттерны

Abstract Inversion // Инверсия абстракции

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

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 (`'_','__'`).

Big ball of mud // Большой комок грязи

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.

Input kludge // Затычка на ввод

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.

Magic button // Волшебная кнопка

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.

Mutilation // Членовредительство

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).

Stovepipe Enterprise // Дымоход предприятия

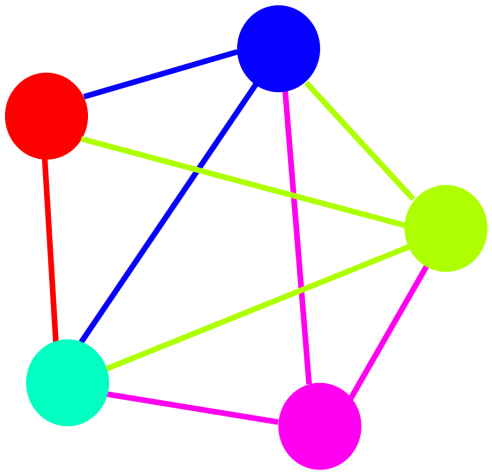
Несогласованность архитектур корпоративных приложений, которая ухудшает или сводит на нет адаптивность, повторное использование и функциональную совместимость. Этот антипаттерн сравнивают с дымоходом печи, который необходимо постоянно ремонтировать. Причём при ремонте используются любые подручные средства.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

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

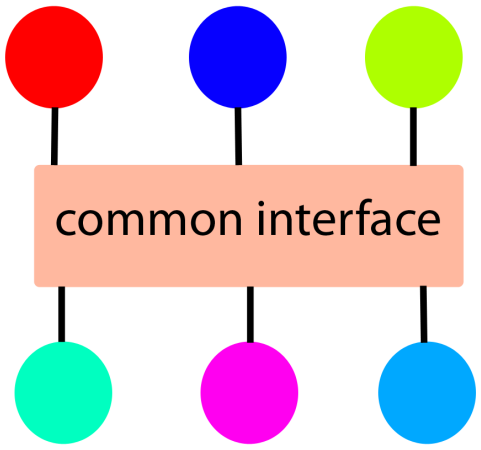
Решение: создание эталонной модели стандартов, создание общей операционной среды, системных профилей, координирующих использование продуктов и стандартов.

Stovepipe System // Дымоход системы

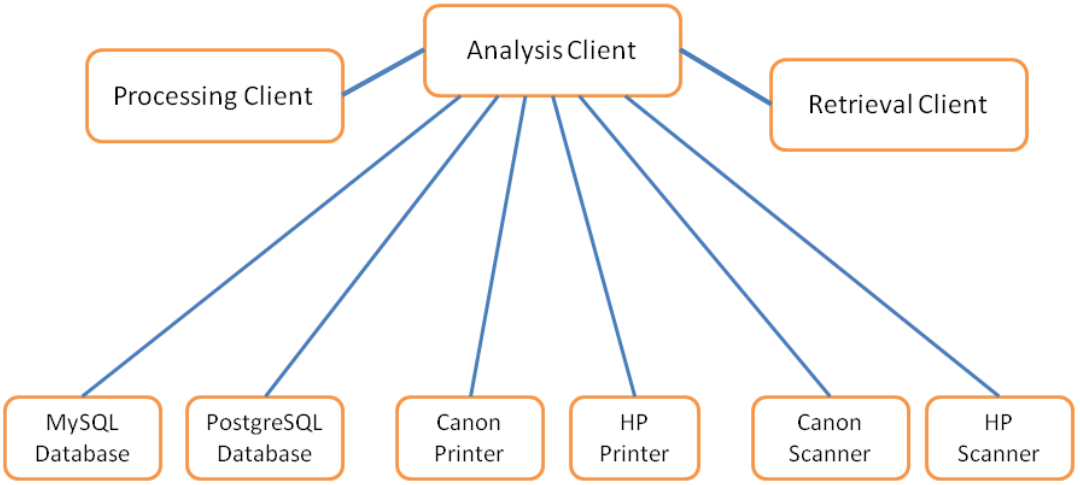
Антипаттерн аналогичен Stovepipe Enterprise, но рассматривается несогласованность в рамках одной системы, а не набора систем. Как правило, это устаревшая (legacy) система, не удовлетворяющая новым потребностям бизнеса. Одна из проблем — отсутствие подсистемных абстракций (в Stovepipe Enterprise это межсистемные 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.



Решение: реорганизация системы в компонентную архитектуру, обеспечивающую гибкую замену программных модулей. Благодаря абстрактному моделированию подсистем, количество открытых интерфейсов получается небольшим.

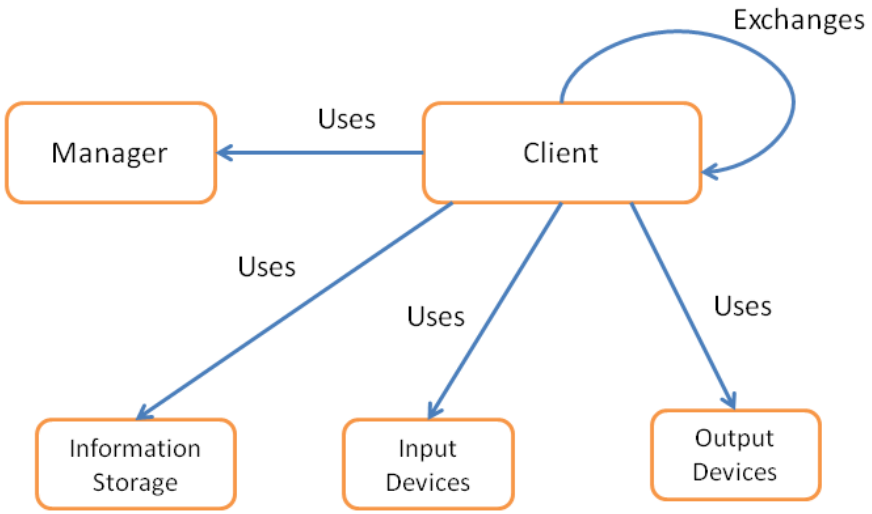


Пример. Есть система работы с графическим контентом: сканирование, печать, сохранение в БД.



В ней три клиентских подсистемы и шесть служебных. Каждая подсистема моделируется классом со своим уникальным интерфейсом. Если мы добавим или изменим любую из них, необходимо модифицировать код клиентов.

Решение. Приведём систему к следующему виду:



Работаем с абстракциями входных, выходных устройств и хранения информации, а также клиентов. Каждое конкретное устройство или службу можно обернуть так, чтобы поддерживались эти абстракции. Службу менеджера добавили для обнаружения и разделения абстрактных сервисов.

Jumble // Путаница

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.

Практическое задание

В этой самостоятельной работе тренируем умения:

1. Находить антипаттерны.
2. Выбирать методы их устранения.

Зачем:

Для улучшения своего кода.

1. Провести анализ своего проекта на использование антипаттернов. Какие антипаттерны удалось обнаружить?
2. Продумать методы устранения.

Дополнительные материалы

1. Что такое антипаттерны? / Хабр.
2. Software Development AntiPatterns.
3. Software Architecture AntiPatterns.
4. Project Management AntiPatterns.
5. Stovepipe Enterprise.
6. Stovepipe System.
7. Jumble.
8. An Introduction to Anti-Patterns. Preventing Software Design Anomalies.
9. 9 антипаттернов, о которых должен знать каждый программист.
10. Признаки плохого кода.
11. Resign Patterns.
12. The Little Book of Python Anti-Patterns.
13. GitHub: quantifiedcode / python-anti-patterns.
14. Idiomatic Python: EAFP versus LBYL.

Используемая литература

Для подготовки методического пособия были использованы следующие ресурсы:

1. Антипаттерн.
2. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2015.