Паттерны проектирования

Егор Суворов

Курс «Парадигмы и языки программирования», подгруппа 3

Понедельник, 9 октября 2017 года

Как живут программы

- Программы нужны для решения каких-то практических задач (кассовый аппарат).
- Программы пишутся на основе каких-то моделей реального мира (формальная постановка задачи).
- Реальный мир меняется (новые акции) $\Rightarrow$ модель меняется (новые правила расчётов) $\Rightarrow$ надо менять программу.
- Следствие: программы постоянно меняются.
- Программисты уходят и приходят.
- Следствие: одну программу пишут и дописывают разные программисты.
- **Одной корректности в данный момент недостаточно!**

Для кого пишутся программы

- Корректности можно добиться разными способами.
- Некоторые способы быстрее работают, некоторые требуют меньше памяти.
- А какие-то *лучше поддерживаются в будущем* (расширяются, изменяются, отлаживаются).
- Текущая мода: программисты дороже компьютеров.
- Следствие: программы пишутся для людей, которые их потом будут менять.
- Следствие: важны читаемость, расширяемость, очевидность поведения, документация...

Инженерные задачи

Аналогия:

- Предположим, вы хотите спроектировать (а потом построить) мост через реку.
- Наверняка конкретно такого моста, как вам надо, ещё никто и никогда не строил.
- Тем не менее, что-то про мосты человечество знает: основные типы, когда их использовать, что-то про материалы...
- Значит, мост вы проектируете не с нуля, а на какой-то основе.
- Но индивидуальные особенности всё равно приходится учитывать.

Задачи в программировании

- Предположим, вы хотите написать сайт.
- Разумеется, ровно такого ещё никто и никогда не писал.
- Тем не менее, про сайты много чего известно: как хранить данные, как обрабатывать несколько запросов одновременно, какие бывают уязвимости, как лучше хранить пароли пользователей...
- Значит, сайт тоже можно сделать не с нуля, а на основе каких-то существующих идей.
- Разумеется, кроме них в сайте будет что-то индивидуальное.
- **В программировании вы постоянно комбинируете готовые идеи, решения и конструкции**

Примеры стандартных идей

- Для обработки данных: разделять ввод, обработку и вывод данных.
- Хранение данных в формате, удобном для обработки.
- Строгая типизация, чтобы ловить ошибки несоответствия размерностей.
- Циклы по элементам массива, а не по индексам.
- Структура данных «словарь» (хэш-таблица).
- Использование библиотек вместо прямой работы с ОС $\Rightarrow$ кроссплатформенность.

Что такое паттерн

- *Паттерн* (или *шаблон проектирования*) — это какая-то стандартная конструкция для решения каких-то *архитектурных* задач.
- Другими словами: какие абстракции и интерфейсы полезно использовать в каких ситуациях.
- Мы пройдёмся очень поверхностно. Вообще есть большие книжки (от «банды четырёх») и курсы, где про это рассказывают.
- Часто называются по-английски.

Поведенческие шаблоны

Рассказывают, как объекты могут между собой взаимодействовать.
Примеры:

- *Итератор* (*Iterator*): уже познакомились. Позволяет итерироваться по произвольным коллекциям.
- *Наблюдатель* (*Observer*): если некоторые объекты могут рассылать события, а некоторые должны на них реагировать, то можно создать интерфейс «наблюдатель за событием» и вспомогательные классы для рассылки событий. Тогда наблюдателю надо лишь добавиться в нужный список, а инициатору события — вызвать метод «оповести наблюдателей из списка».
- *Посетитель* (*Visitor*): будет в домашнем задании, разберём позже.

Структурные шаблоны

Рассказывают, как компоновать между собой классы и объекты.
Примеры:

- *Адаптер* (*Adapter*): если у нас есть класс с интерфейсом A, а нам нужно передать куда-то класс с другим интерфейсом B (другие названия), то можно создать класс, который просто будет конвертировать вызовы интерфейса B в вызовы A.
- *Компоновщик* (*Composite*): если нам часто нужно совершать одинаковые операции над разными объектами (например, отрисовать элементы окна на экране), то их можно объединить в коллекцию, которая предоставляет общий интерфейс для этих объектов.

Порождающие шаблоны

Рассказывают, как создавать и компоновать объекты в коде. Примеры:

- *Строитель* (*Builder*): если есть объект с очень сложным конструктором, то можно создать промежуточный объект, который будет «накапливать» в себе параметры конструктора, а потом создаст объект:

```
def create_button():
    builder = ButtonBuilder()
    builder.set_text("Кнопка")
    if some_complex_condition(): builder.set_disabled()
    return builder.build()
```

- *Абстрактная фабрика* (*Abstract factory*): если объекты постоянно требуют какого-то включения в систему, то можно не вызывать конструкторы напрямую, а выделить кусок системы, который будет правильным образом конструировать объекты.

Замечания

- Некоторые паттерны идут «от капитана» (можно назвать «очевидным здравым смыслом»), некоторые более хитры.
- Часто может казаться, что без паттерна легко обойтись, потому что в программе нужен очень частный случай. Зато если требования поменяются — можно огрести.
- Легко перегнуть палку: «абстрактные фабрики абстрактных фабрик» и прочие радости.

Постановка задачи

Пусть есть разношёрстные классы, например, элементы синтаксического дерева языка ЯТЬ: `Scope`, `Function`, `Read`, `Write`. И ещё есть операции над этими объектами: вывести красиво отформатированный код, соптимизировать кусочек дерева (например, заменить 2+2 на 4), скомпилировать программу.
Вопрос: где описывать эти операции?

Постановка задачи

Пусть есть разношёрстные классы, например, элементы синтаксического дерева языка ЯТЬ: `Scope`, `Function`, `Read`, `Write`. И ещё есть операции над этими объектами: вывести красиво отформатированный код, соптимизировать кусочек дерева (например, заменить 2+2 на 4), скомпилировать программу.
Вопрос: где описывать эти операции? Несколько вариантов:

1. Добавить методы `pretty_print`, `optimize`, `compile` всем классам.
2. Сделать одну функцию на операцию, которая может вызывать сама себя рекурсивно и руками проверяет тип объекта.
3. Паттерн *Visitor*: сделать класс, соответствующий операции, с методами «обработай `Scope`», «обработай `Function`», и так далее.

Пример внутри классов

```
import random
class Cat:
    def pat_head(self): print("Purr!")
    def rub_belly(self): print("Don't you dare!")
    def happiness(self): return random.randint(1, 5)
    def pet(self): self.pat_head()
    def is_safe(self): return self.happiness() >= 3
class Dog:
    def pat_head(self): print("I'm happy!")
    def rub_belly(self): print("I'm very happy!")
    def tail_wagging(self): return True
    def pet(self): self.rub_belly()
    def is_safe(self): return self.tail_wagging()
```

Пример в функциях

```
import random
class Cat:
    def pat_head(self): print("Purr!")
    def rub_belly(self): print("Don't you dare!")
    def happiness(self): return random.randint(1, 5)
class Dog:
    def pat_head(self): print("I'm happy!")
    def rub_belly(self): print("I'm very happy!")
    def tail_wagging(self): return True
def pet(a):
    if isinstance(a, Cat): a.pat_head()
    elif isinstance(a, Dog): a.rub_belly()
def is_safe(a):
    if isinstance(a, Cat): return a.happiness() >= 3
    elif isinstance(a, Dog): return a.tail_wagging()
```

Возможные проблемы

Если добавляем методы в классы, то:

Возможные проблемы

Если добавляем методы в классы, то:

- Логика операции оказывается разнесённой по разным кускам кода.
- Добавили операцию — надо проверить, что ни в одном из классов не забыли (Python-то не проверяет соответствие интерфейсам).
- При добавлении операций изменяются интерфейсы классов, придётся всё перекомпилировать.

Возможные проблемы

Если добавляем методы в классы, то:

- Логика операции оказывается разнесённой по разным кускам кода.
- Добавили операцию — надо проверить, что ни в одном из классов не забыли (Python-то не проверяет соответствие интерфейсам).
- При добавлении операций изменяются интерфейсы классов, придётся всё перекомпилировать.

Если делаем одну функцию, то:

Возможные проблемы

Если добавляем методы в классы, то:

- Логика операции оказывается разнесённой по разным кускам кода.
- Добавили операцию — надо проверить, что ни в одном из классов не забыли (Python-то не проверяет соответствие интерфейсам).
- При добавлении операций изменяются интерфейсы классов, придётся всё перекомпилировать.

Если делаем одну функцию, то:

- Куча неприятного кода для определения типа, захочется разделить определение типа и содержательную обработку.
- В некоторых языках проверить тип переданного объекта в run time очень сложно (например, C — надо что-то руками делать).
- В статически типизированных языках надо ещё и изменять тип переменной.

Волшебный единорог

Паттерн Visitor (*посетитель*):

1. Создаём интерфейс `Visitor` с функциями `visit_scope`, `visit_function`, `visit_read`... [1].
2. Требуем, чтобы каждый класс имел функцию `accept(visitor)`, которая бы вызывала у параметра нужный метод.
3. Для определения операции достаточно создать новый класс, реализующий интерфейс `Visitor`.
4. Для вызова операции на элементе достаточно вызывать `item.accept(visitor)`.

[1] в некоторых языках (C++, Java) все методы назовут visit, а отличия будут лишь в типе аргумента

Пример-1

```
import random
class Cat:
    def pat_head(self): print("Purr!")
    def rub_belly(self): print("Don't you dare!")
    def happiness(self): return random.randint(1, 5)
    def visit(self, v): return v.visit_cat(self)
class Dog:
    def pat_head(self): print("I'm happy!")
    def rub_belly(self): print("I'm very happy!")
    def tail_wagging(self): return True
    def visit(self, v): return v.visit_dog(self)
```

Пример-2

```
class PetVisitor:
    def visit_cat(self, a): a.pat_head()
    def visit_dog(self, a): a.rub_belly()
class IsSafeVisitor:
    def visit_cat(self, a): return a.happiness() >= 3
    def visit_dog(self, a): return a.tail_wagging()

Cat().visit(PetVisitor())  # Purr!
Dog().visit(PetVisitor())  # I'm very happy!
print(Cat().visit(IsSafeVisitor()))  # random
print(Dog().visit(IsSafeVisitor()))  # True
```

Замечание про возвращаемые значения

1. В статически типизированных языках нельзя вернуть произвольное значение из методов, надо указывать какой-то тип.
2. При этом указывать тип «всё что угодно» нехорошо, так как теряется безопасность. Фиксировать конкретный тип для всех посетителей тоже нехорошо.
3. Поэтому часто считают, что `visit`/`accept` вообще ничего не возвращают. А в Python легко забыть `return` и не заметить :)
4. Если посетитель что-то должен вычислять, то он хранит результат вычисления внутри себя:

```
class IsSafeVisitor:
    def visit_cat(self, a):
        self.result = a.happiness() >= 3
    def visit_dog(self, a):
        self.result = a.tail_wagging()
```