Оглавление

Неделя 4

Паттерны проектирования (часть 2)

4.1 Паттерн Chain of responsibility

4.1.1 Задача паттерна Chain of Responsibility

Рассмотрим ещё один поведенческий шаблон — Chain of Responsibility (цепочка обязанностей). Представьте, что ваш автомобиль только что сломался и вам необходимо сдать его в ремонт (и можно починить не только саму поломку, но и поменять масло и так далее).
Цепочка со стороны пользователя будет выглядеть так:

- Случилась поломка автомобиля;
- Посещаете автосервис;
- Даёте задание:
    1. Починить двигатель;
    2. Поменять масло;
    3. Залить антифриз;
    4. Поменять резину;
    5. И другие задания по ситуации.
- Ждёте выполнения задания и звоните через день.

Примерно так выглядит спроектированная цепочка событий с точки зрения пользователя. А именно, некоторому классу задаётся список заданий, после чего производится запуск сразу всех действий.

С точки зрения сервиса это выглядит так: работник передаёт задание в первый отдел, и если её может решить первый отдел, то задача переходит к нему. Иначе ко второму отделу и так далее, пока не дойдёт до того отдела, который может выполнить задачу с этой машиной. Когда первая задача выполнена, работник запускает по этой же цепочке следующую задачу и так, пока не будет выполнено всё.

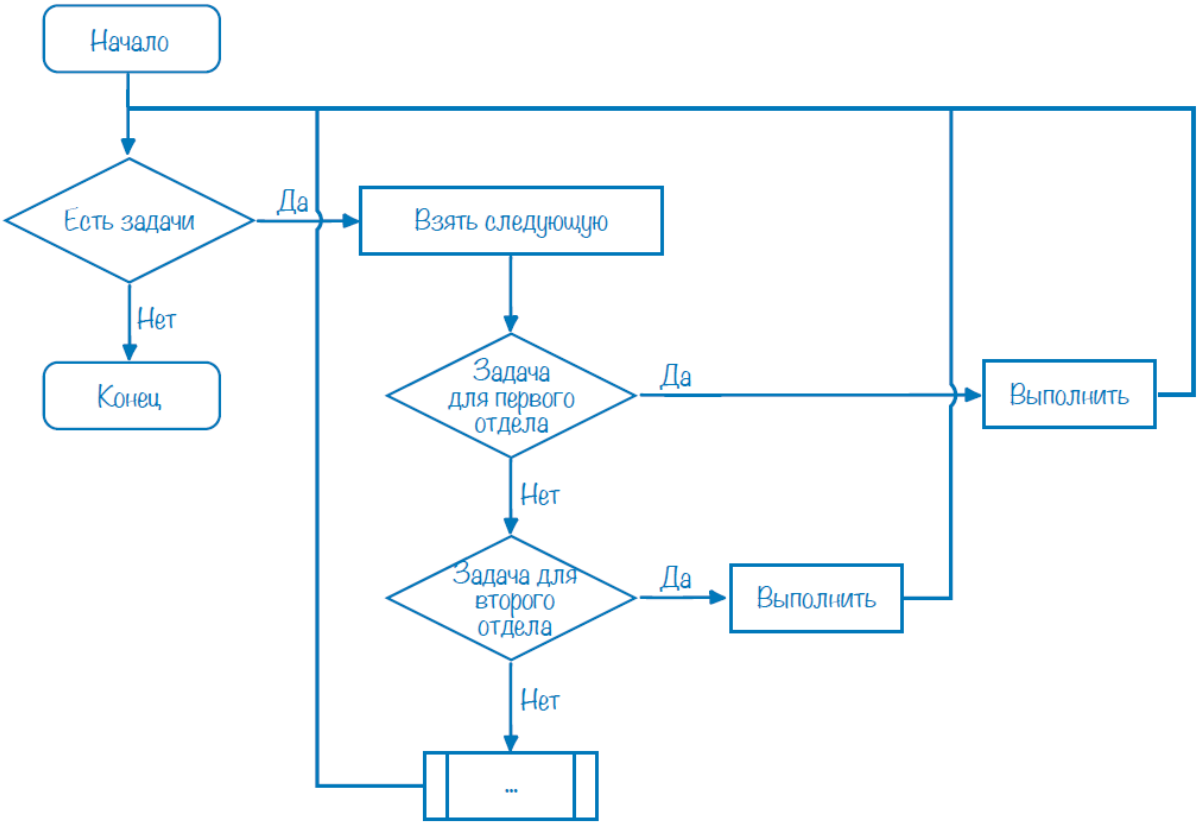

Рис. 4.1: Механизм поведенческого шаблона Chain of responsibility

Таков принцип поведенческого шаблона Chain of responsibility: объекту передаётся одно или же сразу несколько заданий, выполнение которых запускается по цепочке.

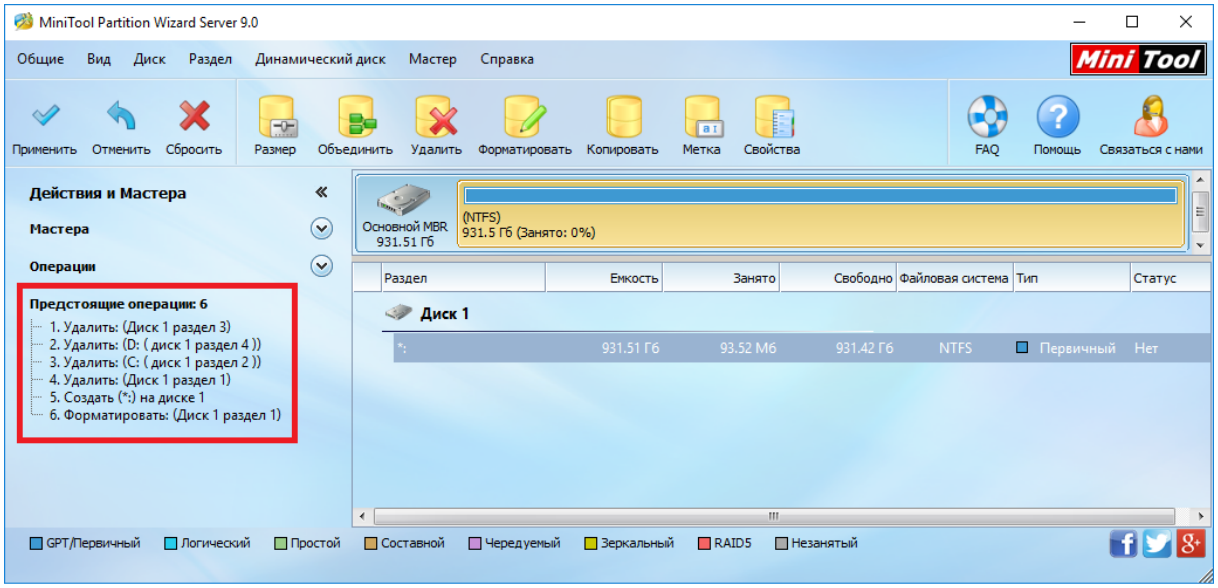

Рис. 4.2: Список операций с диском

Этот шаблон используется в самых разных случаях. Например, программы работы с жёсткими дисками, где вы задаёте, что вам сначала удалить одни разделы, потом создать другие, затем отформатировать, после чего все эти задания по очереди начинают выполняться.

Большинство графических программ применяют этот подход при обработке сообщений. Например, вы запрашиваете контекстное меню у кнопки. Она смотрит, есть ли у неё своё контекстное меню или нет, если есть — выдаёт, если нет — то передаёт сообщение о том, что необходимо показать контекстное меню, своему родителю, форме. Та в свою очередь тоже изучает и смотрит, может ли она показать. Если нет — то передаёт дальше. Может получиться, что запрос на показ контекстного меню может передаться непосредственно операционной системе, в результате чего вы получите стандартное контекстное меню.

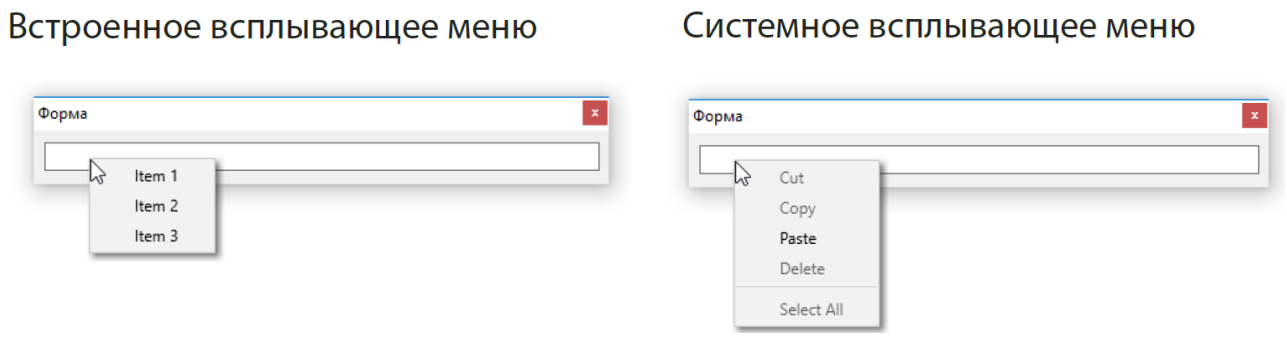

Рис. 4.3: Контекстное меню

Условия применения Chain of Responsibility:

- Присутствуют типизированные сообщения;
- Все сообщения должны быть обработаны хотя бы один раз;
- Работа с сообщением: делай сам или передай другому.

Таким способом можно реализовать, например, работу web-сервера, на который приходит огромное количество сообщений от пользователей, которые после передаются, в некоторую цепочку обработки событий. Одни исполнители записывают в базу данных, другие позволяют скачать файл, а третьи выдают html-страницу текста.

4.1.2 Краткая реализация паттерна Chain of Responsibility

Рассмотрим реализацию цепочки обязанностей на примере квестов в компьютерной игре. Опишем класс игрока, у которого есть имя, опыт и множества сданных и полученных квестов.

```
class Character:
    def __init__(self):
        self.name = "Nagibator"
        self.xp = 0
        self.passed_quests = set()
        self.taken_quests = set()
```

Объявим несколько квестов: поговорить с , поохотиться на крыс и принести доски из сарая. У каждого квеста будет название и опыт за его выполнение. При получении квеста мы проверяем, не был ли он нами получен или сдан. Если он был сдан, то взять его мы не можем. Если же он был получен, то квест можно сдать т.е. написать "квест сдан удаляем из списка полученных, добавляем в список выполненных и получаем опыт.

```
def add_quest_speak(char):
    quest_name = "Поговорить"
    xp = 100
    if quest_name not in (char.passed_quests | char.
       taken_quests):
        print(f"Квест получен: \"{quest_name}\"")
        char.taken_quests.add(quest_name)
    elif quest_name in char.taken_quests:
        print(f"Квест сдан: \"{quest_name}\"")
        char.passed_quests.add(quest_name)
        char.taken_quests.remove(quest_name)
        char.xp += xp

def add_quest_hunt(char):
    quest_name = "Охота "
    xp = 300
    if quest_name not in (char.passed_quests | char.
       taken_quests):
        print(f"Квест получен: \"{quest_name}\"")
        char.taken_quests.add(quest_name)
    elif quest_name in char.taken_quests:
        print(f"Квест сдан: \"{quest_name}\"")
        char.passed_quests.add(quest_name)
        char.taken_quests.remove(quest_name)
        char.xp += xp
```

```
def add_quest_carry(char):
    quest_name = "Принести доски"
    xp = 200
    if quest_name not in (char.passed_quests | char.
      taken_quests):
        print(f"Квест получен: \"{quest_name}\"")
        char.taken_quests.add(quest_name)
    elif quest_name in char.taken_quests:
        print(f"Квест сдан: \"{quest_name}\"")
        char.passed_quests.add(quest_name)
        char.taken_quests.remove(quest_name)
        char.xp += xp
```

Напишем QuestGiver, который будет давать квесты из списка. В этот список можно добавлять квесты (app_quests()). Так же можно дать персонажу квесты (handle_quests()), а для этого мы проходимся по списку квестов и выполняем их персонажем.

```
class QuestGiver:
    def __init__(self):
        self.quests = []

    def add_quest(self, quest):
        self.quests.append(quest)

    def handle_quests(self, character):
        for quest in self.quests:
            quest(character)
```

Для проверки создадим список квестов, QuestGiver (в который добавим квесты из списка) и персонажа, которому передадим все квесты.

```
all_quests = [add_quest_speak, add_quest_hunt,
  add_quest_carry]
quest_giver = QuestGiver()

for quest in all_quests:
    quest_giver.add_quest(quest)

player = Character()
quest_giver.handle_quests(player)
```

```
Квест получен: "Поговорить"
Квест получен: "Охота"
Квест получен: "Принести доски"
```

```
print("Получено: ", player.taken_quests)
print("Сдано: ", player.passed_quests)
```

```
Получено: 'Поговорить' ,'Принести доски' , 'Охота'
Сдано: set()
```

Персонаж получил все квесты, но ни одного не сдал. Изменим список полученных квестов. Пусть на данный момент буду получены квесты "принести доски"и "поговорить".

```
player.taken_quests = {'Принести доски    ', 'Поговорить'}
quest_giver.handle_quests(player)
```

```
Квест сдан: "Принести доски"
Квест получен: "Охота"
Квест сдан: "Поговорить"
```

Уже полученные квесты оказались сданы, и мы получили квест, которого раньше не было.

```
quest_giver.handle_quests(player)
```

```
Квест сдан: "Охота"
```

Полученный на прошлой итерации квест оказался сдан, а остальные два уже были получены и сданы:

```
print("Получено: ", player.taken_quests)
print("Сдано: ", player.passed_quests)
```

```
Получено: set()
Сдано: 'Поговорить', 'Принести доски', 'Охота'
```

Мы реализовали самую простую цепочку выполнения задач. Это пока не цепочка обязанностей, потому что не хватает ключевой особенности: передачи задачи следующему обработчику, если текущий не может её выполнить.

4.1.3 Практическая реализация паттерна Chain of Responsibility

Персонажа оставим без изменений. И создадим список константных названий квестов:

```
QUEST_SPEAK, QUEST_HUNT, QUEST_CARRY = "QSPEAK", "QHUNT", "
   QCARRY"

class Character:
    def __init__(self):
        self.name = "Nagibator"
```

```
        self.xp = 0
        self.passed_quests = set()
        self.taken_quests = set()
```

Чтобы цепочка обязанностей работала корректно, опишем класс события, которое будет происходить:

```
class Event:
    def __init__(self, kind):
        self.kind = kind
```

Опишем нулевой обработчик (нулевое звено цепочки), которое будет передавать событие на обработку следующему обработчику, если таковой имеется.

```
class NullHandler:
    def __init__(self, successor=None):
        # передаём следующее звено
        self.__successor = successor
    def handle(self, char, event):  # обработчик
        if self.__successor is not None: #даём следующему
            self.__successor.handle(char, event)
```

Изменим код квестов из предыдущего параграфа. Теперь все квесты это классы, унаследованные от NullHandler. Переопределим метод handle: если происходит событие, означающее квест поговорить, то мы выполняем. Иначе передаём на обработку следующему звену цепочки.

```
class HandleQSpeak(NullHandler):

    def handle(self, char, event):
        if event.kind == QUEST_SPEAK:
            xp = 100
            quest_name = "Поговорить сфермером "
            if event.kind not in
            (char.passed_quests | char.taken_quests):
                print(f"Квест получен: \"{quest_name}\"")
                char.taken_quests.add(event.kind)
            elif event.kind in char.taken_quests:
                print(f"Квест сдан: \"{quest_name}\"")
                char.passed_quests.add(event.kind)
                char.taken_quests.remove(event.kind)
                char.xp += xp
        else:
            print("Передаю обработкудальше ")
            super().handle(char, event)
```

```
class HandleQHunt(NullHandler):
    def handle(self, char, event):
        if event.kind == QUEST_HUNT:
            xp = 300
            quest_name = "Охота накрыс "
            if event.kind not in
                (char.passed_quests | char.taken_quests):
                print(f"Квест получен: \"{quest_name}\"")
                char.taken_quests.add(event.kind)
            elif event.kind in char.taken_quests:
                print(f"Квест сдан: \"{quest_name}\"")
                char.passed_quests.add(event.kind)
                char.taken_quests.remove(event.kind)
                char.xp += xp
        else:
            print("Передаю обработкудальше ")
            super().handle(char, event)


class HandleQCarry(NullHandler):

    def handle(self, char, event):
        if event.kind == QUEST_CARRY:
            xp = 200
            quest_name = "Принести дроваизсарая  "
            if event.kind not in
            (char.passed_quests | char.taken_quests):
                print(f"Квест получен: \"{quest_name}\"")
                char.taken_quests.add(event.kind)
            elif event.kind in char.taken_quests:
                print(f"Квест сдан: \"{quest_name}\"")
                char.passed_quests.add(event.kind)
                char.taken_quests.remove(event.kind)
                char.xp += xp
        else:
            print("Передаю обработкудальше ")
            super().handle(char, event)
```

Изменим QuestGiver, чтобы тот мог работать с цепочкой обязанностей. Объявим ему цепочку, с которой он будет работать.

```
class QuestGiver:

    def __init__(self):
```

```
        self.handlers = HandleQSpeak(HandleQHunt(
          HandleQCarry(NullHandler())))
        self.events = []  # изначально пустой списко событий

    def add_event(self, event):  # добавляем события
        self.events.append(event)

    # передаём событие цепочке обязанностей
    def handle_quests(self, char):
        for event in self.events:
            self.handlers.handle(char, event)
```

Объявим список всех возможных событий events и наш questgiver, который может реагировать на все события из списка.

```
events = [Event(QUEST_CARRY), Event(QUEST_HUNT), Event(
  QUEST_SPEAK)]

quest_giver = QuestGiver()

for event in events:
    quest_giver.add_event(event)
```

Проверим работу цепочки обязанностей на примере, аналогичном предыдущему:

```
player = Character()

quest_giver.handle_quests(player)
print()
player.taken_quests = {QUEST_CARRY, QUEST_SPEAK}
quest_giver.handle_quests(player)
print()
quest_giver.handle_quests(player)
```

```
Передаю обработку дальше
Передаю обработку дальше
Квест получен: "Принести дрова из сарая"
Передаю обработку дальше
Квест получен: "Охота на крыс"
Квест получен: "Поговорить с фермером"

Передаю обработку дальше
Передаю обработку дальше
Квест сдан: "Принести дрова из сарая"
Передаю обработку дальше
```

```
Квест получен: "Охота на крыс"
Квест сдан: "Поговорить с фермером"

Передаю обработку дальше
Передаю обработку дальше
Передаю обработку дальше
Квест сдан: "Охота на крыс"
```

Видно, что цепочка обязанностей работает, и квесты, обработка которых невозможна на данном этапе, передаются по ней дальше.

4.2 Паттерн Abstract Factory

4.2.1 Задача паттерна Abstract Factory

Для понимания паттерна абстрактной фабрики сначала рассмотрим пример из жизни. Представьте, что вы хотите построить дом. Самое главное это постройка каркаса, где будут располагаться двери и окна, высота потолка и другие важные вопросы. А с дизайном и материалами определитесь уже в процессе строительства.

Суть абстрактной фабрики: для программы не имеет значение, как создаются компоненты. Необходима лишь «фабрика», производящая компоненты, умеющие взаимодействовать друг с другом.

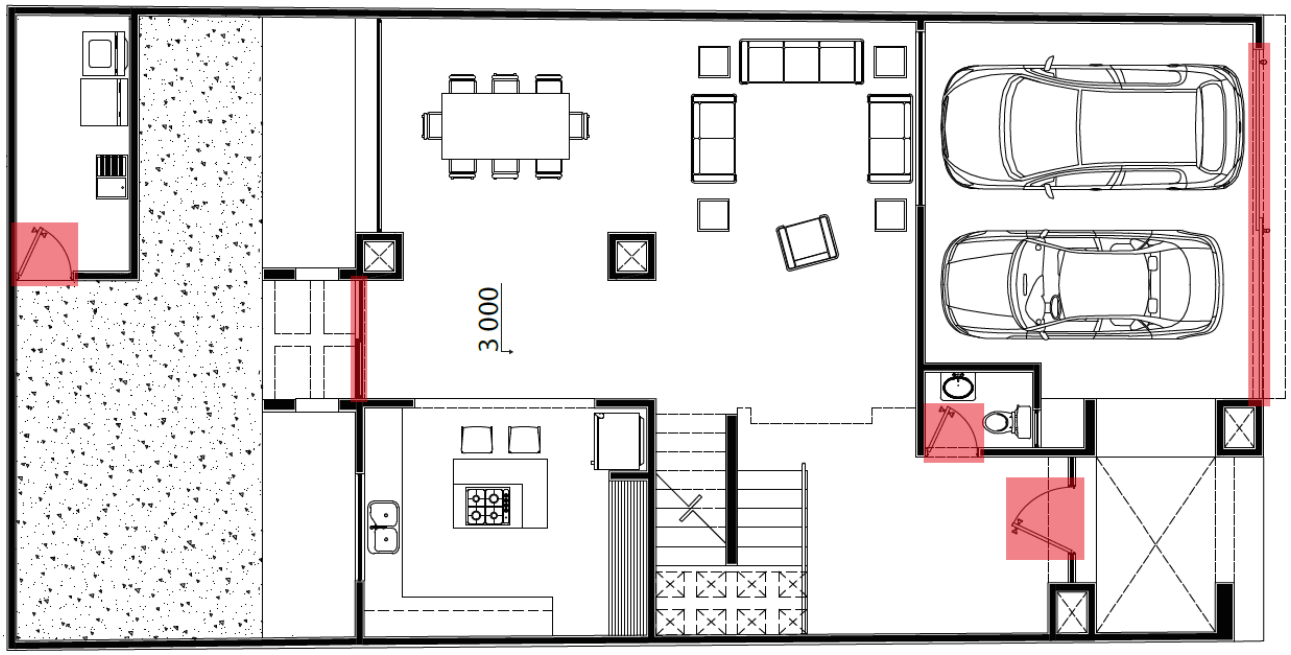

Рис. 4.4

Теперь разберём, как это выглядит в программировании. Предположим, что нам нужна функция создания диалогового окна.

Для этого мы должны уметь:

1. Создавать окно;
2. Создавать кнопки ("OK", "Cancel" и другие);
3. Выводить текстовую информацию ("Are you sure?");
4. (дополнительно) создавать чекбоксы и прочие визуальные эффекты.

Так вот для функции создания диалогового окна нет никакой необходимости знать, как создаются визуальные компоненты внутри него. Функции необходим лишь класс, который умеет как фабрика производить кнопки, текстовые сообщения, чекбоксы и прочее. Сами элементы уже можно производить различным образом и в разных стилях (например, Windows 10, OS X, Linux). Естественно, одна фабрика будет производить элементы только одного типа, а уже другая фабрика какого-то определённого другого типа.

Поэтому фабрика, с которой мы работаем во время написания функции (на этапе проектирования приложения), это некая абстрактная фабрика — класс с описанием всех создаваемых объектов без реализации. Но когда из приложения будет запускаться функция создания диалогового окна (этап выполнения), ей будет передаваться уже реальная фабрика — класс с реализацией создания всех компонент.

Далее мы разберём реализацию шаблонов абстрактной фабрики для формирования текстового отчёта в форматах html и markdown. Причём сначала реализуем её классическим способом, а затем произведём модификацию в духе Python.

4.2.2 Краткая реализация паттерна Abstract Factory

Шаблон абстрактная фабрика позволяет создавать сложные системы взаимодействующих классов, и при помощи конкретных фабрик, специализирует создание определённой структуры для конкретных объектов. На примере реализации классов персонажа в игре рассмотрим реализацию данного шаблона. Создадим абстрактную фабрику с абстрактными методами:

```
from abc import ABC, abstractmethod

class HeroFactory(ABC):
    @abstractmethod # создаёт героя с заданным именем
    def create_hero(self, name):
        pass

    @abstractmethod  # создаёт оружие
    def create_weapon(self):
        pass
```

```
    @abstractmethod  # создаёт заклинание
    def create_spell(self):
        pass
```

Герой будет одного из нескольких классов: воин, маг или убийца. Причём каждому классу даётся свой конкретных предмет экипировки и своё заклинание. Опишем фабрику войнов:

```
class WarriorFactory(HeroFactory):
    def create_hero(self, name):
        return Warrior(name)  # создаём война с заданным именем

    def create_weapon(self):
        return Claymore()  # оружие война - клеймор

    def create_spell(self):
        return Power()  # заклинание война - сила

class Warrior:  # класс войнов
    def __init__(self, name):
        self.name = name
        self.weapon = None
        self.armor = None
        self.spell = None

    def add_weapon(self, weapon):
        self.weapon = weapon

    def add_spell(self, spell):
        self.spell = spell

    def hit(self):  # удар оружием
        print(f"W. {self.name} uses {self.weapon.hit()}")
        self.weapon.hit()

    def cast(self):  # использование заклинания
        print(f"W. {self.name} casts {self.spell.cast()}")
        self.spell.cast()

class Claymore:
    def hit(self):
        return "Claymore"
```

```
class Power:
    def cast(self):
        return "Power"
```

Аналогично описываются фабрики Мага и Убийцы:

```
class MageFactory(HeroFactory):
    def create_hero(self, name):
        return Mage(name)

    def create_weapon(self):
        return Staff()

    def create_spell(self):
        return Fireball()

class Mage:
    def __init__(self, name):
        self.name = name
        self.weapon = None
        self.armor = None
        self.spell = None

    def add_weapon(self, weapon):
        self.weapon = weapon

    def add_spell(self, spell):
        self.spell = spell

    def hit(self):
        print(f"M. {self.name} uses {self.weapon.hit()}")
        self.weapon.hit()

    def cast(self):
        print(f"M. {self.name} casts {self.spell.cast()}")
        self.spell.cast()

class Staff:
    def hit(self):
        return "Staff"

class Fireball:
    def cast(self):
        return "Fireball"
```

```
class AssassinFactory(HeroFactory):
    def create_hero(self, name):
        return Assassin(name)

    def create_weapon(self):
        return Dagger()

    def create_spell(self):
        return Invisibility()


class Assassin:
    def __init__(self, name):
        self.name = name
        self.weapon = None
        self.armor = None
        self.spell = None

    def add_weapon(self, weapon):
        self.weapon = weapon

    def add_spell(self, spell):
        self.spell = spell

    def hit(self):
        print(f"A. {self.name} uses {self.weapon.hit()}")
        self.weapon.hit()

    def cast(self):
        print(f"A. {self.name} casts {self.spell.cast()}")


class Dagger:
    def hit(self):
        return "Dagger"

class Invisibility:
    def cast(self):
        return "Invisibility"
```

Абстрактные фабрики описаны. Теперь с их помощью определим функцию, создающую персонажа. На вход она будет принимать конкретную фабрику, которая создаёт персонажа нужного класса с соответствующей экипировкой:

```
def create_hero(factory):
    hero = factory.create_hero("Nagibator")
    weapon = factory.create_weapon()
    ability = factory.create_spell()

    hero.add_weapon(weapon)
    hero.add_spell(ability)

    return hero
```

Создадим убийцу :

```
factory = AssassinFactory()
player = create_hero(factory)
player.cast()
player.hit()
```

```
A. Nagibator casts Invisibility
A. Nagibator uses Dagger
```

Убийца успешно создан, и может бить оружием и кастовать. Создадим мага:

```
factory = MageFactory()
player = create_hero(factory)
player.cast()
player.hit()
```

```
M. Nagibator casts Fireball
M. Nagibator uses Staff
```

Таким образом мы написали простейшую реализацию абстрактной фабрики, создающую систему взаимосвязанных классов из персонажа, оружия и заклинания. Но этот код можно сделать короче, используя конструкции класс-метод, в Python.

Полная реализация Абстрактной фабрики

4.2.3 Практическая реализация паттерна Abstract Factory

Предыдущий код получился очень громоздким и с повторениями. Изменим его с помощью питоновского механизма "класс-метод". Избавимся от абстрактных классов. Вместо абстрактных методов будем использовать классовые методы. Они позволят пользоваться некоторыми функциями, специфичными для некоторого класса. Вместо self будет передаваться класс, методы которого мы будем использовать.

```
class HeroFactory:
    @classmethod
    def create_hero(Class, name):
```

```
        return Class.Hero(name)

    @classmethod
    def create_weapon(Class):
        return Class.Weapon()

    @classmethod
    def create_spell(Class):
        return Class.Spell()
```

Создадим фабрику война с использованием механизма класс-метод:

```
class WariorFactory(HeroFactory):
    class Hero:  # подкласс героя для данного класса
        def __init__(self, name):
            self.name = name
            self.weapon = None
            self.armor = None
            self.spell = None

        def add_weapon(self, weapon):
            self.weapon = weapon

        def add_spell(self, spell):
            self.spell = spell

        def hit(self):
            print(f"Warior hits with {self.weapon.hit()}")
            self.weapon.hit()

        def cast(self):
            print(f"Warior casts {self.spell.cast()}")
            self.spell.cast()

    class Weapon:  # подкласс оружия
        def hit(self):
            return "Claymore"

    class Spell:  # подкласс заклинания
        def cast(self):
            return "Power"
```

Аналогично с двумя другими фабриками:

```
class MageFactory(HeroFactory):
    class Hero:
        def __init__(self, name):
            self.name = name
            self.weapon = None
            self.armor = None
            self.spell = None

        def add_weapon(self, weapon):
            self.weapon = weapon

        def add_spell(self, spell):
            self.spell = spell

        def hit(self):
            print(f"Mage hits with {self.weapon.hit()}")
            self.weapon.hit()

        def cast(self):
            print(f"Mage casts {self.spell.cast()}")
            self.spell.cast()

    class Weapon:
        def hit(self):
            return "Staff"

    class Spell:
        def cast(self):
            return "Fireball"


class AssassinFactory(HeroFactory):
    class Hero:
        def __init__(self, name):
            self.name = name
            self.weapon = None
            self.armor = None
            self.spell = None

        def add_weapon(self, weapon):
            self.weapon = weapon
```

```
        def add_spell(self, spell):
            self.spell = spell

        def hit(self):
            print(f"Assassin hits with {self.weapon.hit()}"
                  )
            self.weapon.hit()

        def cast(self):
            print(f"Assassin casts {self.spell.cast()}")

    class Weapon:
        def hit(self):
            return "Dagger"

    class Spell:
        def cast(self):
            return "Invisibility"
```

Создание героя методом create_hero() остаётся таким же:

```
def create_hero(factory):
    hero = factory.create_hero("Nagibator")

    weapon = factory.create_weapon()
    spell = factory.create_spell()

    hero.add_weapon(weapon)
    hero.add_spell(spell)

    return hero
```

Попробуем создать персонажей различных классов, передавая различные фабрики:

```
player = create_hero(AssassinFactory)
player.cast()
player.hit()
```

```
Assassin casts Invisibility
Assassin hits with Dagger
```

Создали убийцу, который становится невидимым и бьёт кинжалом.

```
player = create_hero(MageFactory)
player.cast()
player.hit()
```

```
Mage casts Fireball
Mage hits with Staff
```

Таким образом мы сделали громоздкий код простым и читаемым, избавившись от повторений с помощью механизма ”класс-метод”.

Исходный код сокращённой абстрактной фабрики

4.3 Конфигурирование через YAML

4.3.1 Язык YAML. Назначение и структура. PyYAML

В какой-то момент программисты сталкиваются с тем, что может понадобиться написать конфигуратор к программе. Например, создать текстовый файл с основными параметрами программы.
Основные способы конфигурирования программ:

- **xml-файл —** файл разметки тэгами;

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <anc>
      <data>10</data>
      <name>none</name>
   </anc>
   <option1>sample
text
</option1>
   <option2>smaple text
</option2>
   <option3>
      <element>
         <el1>
            <name>data</name>
            <obj><data>20.0</data></obj>
         </el1>
      </element>
      <element>
         <el2>
            <name>data</name>
            <obj><data>10</data>
               <name>none</name></obj>
         </el2>
      </element>
   </option3>
</root>
```

- **ini-файл**, где всё делится на секции ключ **—** значение;

```
[category1]
option1=value1
option2=value2
[category2]
option1=value1
option2=value2
```

- **json-файл —** код на языке javascript;

```
{
  "option1": "sample\ntext\n",
  "option2": "smaple text\n",
  "option3": [
    {
      "el1": {
        "obj": {
          "data": 20.0
        },
        "name": "data"
      }
    },
    {
      "el2": {
        "obj": {
          "data": 10,
          "name": "none"
        },
        "name": "data"
      }
    }
  ],
  "anc": {
    "data": 10,
    "name": "none"
  }
}
```

- **YAML-файл**.

```yaml
# блочные литералы
option1: | # далее следует текст с учётом переносов
  sample
  text
option2: > # в тексте далее переносы учитываться не будут
  smaple
  text

anc: &anc # определяем якорь
  data: 10 # сопоставление имени и значения
  name: none

option3: # далее идёт список элементов
  - el1:
      name: data
      obj: {data: !!float 20} # явное указание типа
  - el2: {name: data, obj: *anc} # используем якорь
```

Изначально YAML разрабатывался как замена xml и его расшифровка была **Y**et **A**nother **M**arkup **L**anguage. Цели создания YAML:

- быть легко понятным человеку;
- поддерживать структуры данных, родные для разных языков;
- быть переносимым между языками программирования;
- использовать цельную модель данных для поддержки обычного инструментария;
- поддерживать потоковую обработку;
- быть выразительным и расширяемым;
- быть лёгким в реализации и использовании.

Со временем он развивался и стал расшифровываться как **Y**AML **A**in’t **M**arkup **L**anguage поскольку он перестал быть просто языком разметки. И теперь стандарт YAML покрывает JSON, то есть фактически любой файл формата JSON является файлом формата YAML.

В YAML можно перечислять последовательности, делать сопоставление имени и значения (записывать словарь), использовать блочные литералы, занимающие больше одной строки и использовать подстановки: в одном месте файла пометить якорь, а в другом используем на него ссылку (это фактически использование переменных). Файл YAML имеет древовидную структуру данных и форматируется он таким же образом, как и в Python (при помощи пробелов). В YAML позволяется явным образом указывать тип хранимой информации. Например, в каком-то месте написано ”20”. И вы хотите подчеркнуть, что это не число 20, а текст ”20”. И этот формат позволяет вам так сделать. Поддержка YAML есть в большинстве современных языков программирования. В Python для этого используется модуль PyYAML.

4.3.2 Использование YAML для конфигурирования паттерна

Сконфигурируем абстрактную фабрику, которую писали ранее. Импортируем YAML и опишем конфигурацию создания героя. Factory покажет, каким классом будет персонаж с именем name.

```
import yaml
hero_yaml = '''
--- !Character
factory:
  !factory assassin
name:
  7NaGiBaToR7
'''
```

Теперь адаптируем код под работу с yaml-файлами. После кода написанного ранее, создадим конструктор фабрик, который по текстовому описанию (т.е. по загрузчику loader и node yaml-файла) делает фабрику. Загрузчик позволяет выгрузить из yaml-файла какие-то данные, а node - ячейка, которую будем обрабатывать.

```
def factory_constructor(loader, node):
    data = loader.construct_scalar(node)
    if data == "mage":
        return MageFactory
    elif data == "warrior":
        return WarriorFactory
    else:
        return AssassinFactory
```

Методом загрузчика construct_scalar(node) мы выгрузим данные. Scalar потому что в node хранится единственное строковое значение, а не несколько разных значений. И в зависимости от содержания переменной data вернём нужный объект. В данном случае мы считаем, что yaml записан корректно.

Теперь поменяем создание персонажа для работы с yaml-файлами. Для этого создадим класс Character, который наследуется от yaml.YAMLObject и будет обрабатывать файл.

```
class Character(yaml.YAMLObject):
    yaml_tag = "!Character" # тэг, на который надо смотреть

    # теперь стала методом класса.
    # а factory стал атрибутом класса
    def create_hero(self):
        hero = self.factory.create_hero(self.name)
```

```
        weapon = self.factory.create_weapon()
        spell = self.factory.create_spell()

        hero.add_weapon(weapon)
        hero.add_spell(spell)

        return hero
```

Попробуем загрузить yaml-файл и создать героя при помощи конфигурации. Нашему загрузчику loader добавим новый конструктор, создающий фабрики из данных с тэгом factory. Затем создадим героя hero, по данным загруженным из yaml-файла с помощью метода creater_hero() у yaml.load(hero_yaml).

```
loader = yaml.Loader
loader.add_constructor("!factory", factory_constructor)
hero = yaml.load(hero_yaml).create_hero()
hero.hit()    # пробуем ударить
hero.cast()   # пробуем ударить
```

```
Assassin 7NaGiBaToR7 hits with Dagger
Assassin 7NaGiBaToR7 casts Invisibility
```

В итоге мы получили убийцу из yaml-файла. Таким образом мы можем конфигурировать абстрактные фабрики с помощью yaml-файлов.

Продвинутый пример на использование YAML