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.

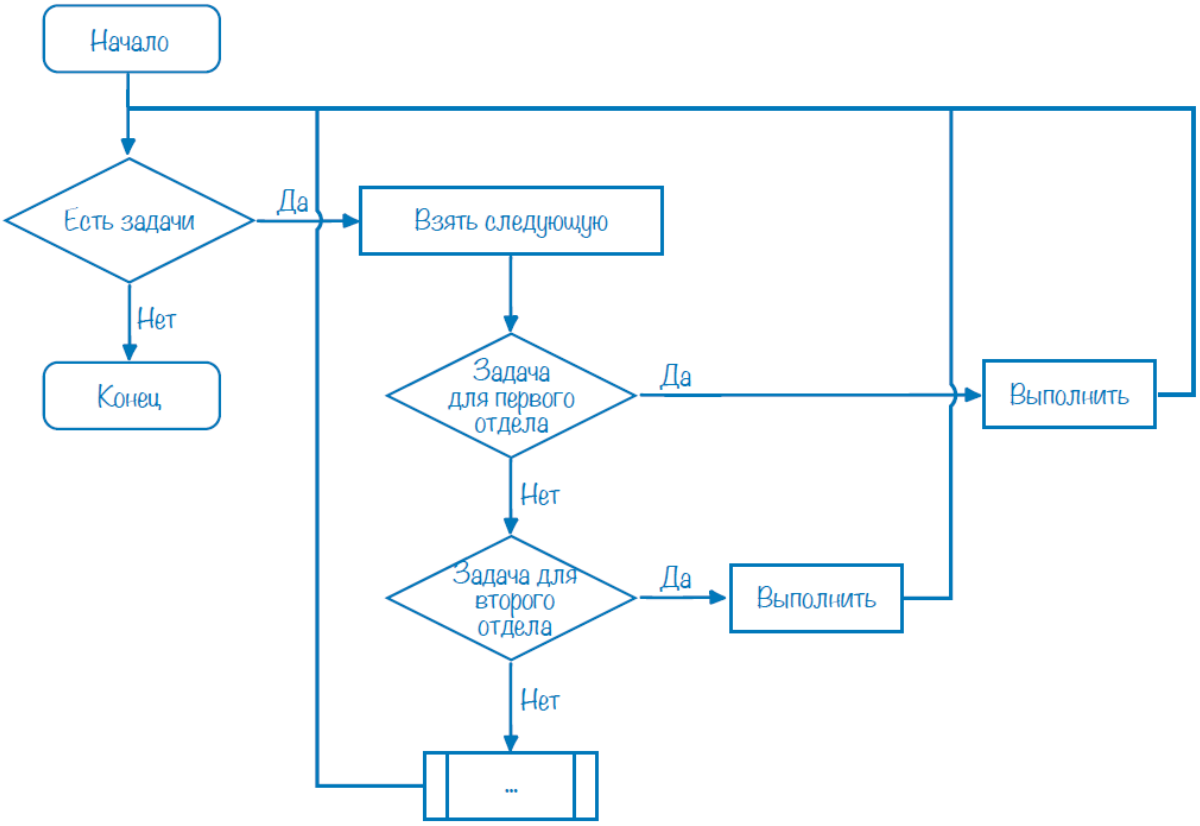

Рис. 4.1: Механизм поведенческого шаблона Chain of responsibility

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

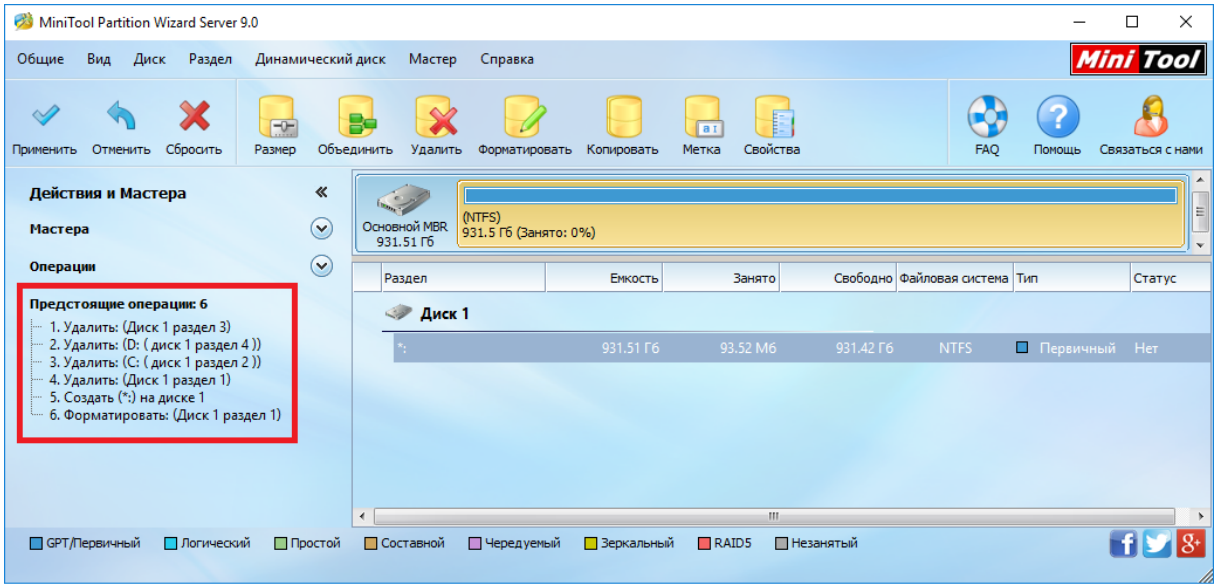

Рис. 4.2: Список операций с диском

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.

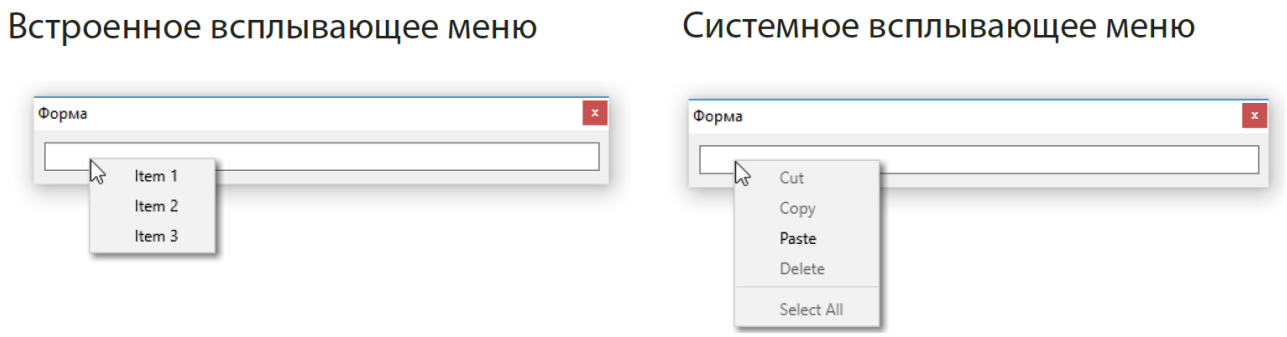

Рис. 4.3: Контекстное меню

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 Цепочка Обязанностей

Рассмотрим готовую реализацию шаблона Цепочка Обязанностей на примере класса Машина, у которой есть километраж, некоторое количество бензина и масла. И далее идут три обработчика событий.

```
class Car:
    def __init__(self):
        self.name = "CAR"  # Название
        self.km = 11100    # Пробег
        self.fuel = 5      # Количество бензина, в процентах
        self.oil = 5       # Количество масла, в процентах

# Заправляем машину, если бензина меньше 10
def handle_fuel(car):
    if car.fuel < 10:
        print("Added fuel.")
        car.fuel = 100

# Сбросить пробег на 0, если он больше 10000
def handle_km(car):
    if car.km > 10000:
        print("Made a car test.")
        car.km = 0

# Залить масло, если его меньше 10
def handle_oil(car):
    if car.oil < 10:
        print("Added oil.")
        car.oil = 100

# Класс для Автосервиса
class CarService:
    def __init__(self):
        self.handlers = []  # цепочка обработчиков

    def add_handler(self, handler):
        self.handlers.append(handler) # добавляем обработчик

    def handle_car(self, car):
        for handler in self.handlers:
            handler(car) # запускаем все обработчики по очереди
```

При инициализации сервисного центра создаём пустую последовательность

обработчиков. Затем идёт метод класса, который заполняет эту последовательность и метод, выполняющий все эти обработчики в цикле.

В основной программе сначала описываются всевозможные обработчики, создаётся класс сервисного центра и в цикле добавляем обработчики в гараж и запускаем их для машины. Сначала отправим машину в автосервис, а затем поотдельности поменяем параметры и снова будем её ремонтировать:

```
def test():
    # определяем список всех обработчиков
    handlers = [handle_fuel, handle_km, handle_oil]

    # создаём автосервис
    service = CarService()

    # Добавляем задания автосервису
    for handle in handlers:
        service.add_handler(handle)

    # Выполняем все задания
    car = Car()
    service.handle_car(car)
    car.fuel = 5
    service.handle_car(car)
    car.oil = 5
    service.handle_car(car)
    car.km = 20000
    service.handle_car(car)

test()
```

```
Added fuel.
Made a car test.
Added oil.
Added fuel.
Added oil.
Made a car test.
```

Далее рассмотрим реализацию главной особенности цепочки обязанностей — "делай сам или передай другому".

Исходный код.

4.1.3 Практическая реализация паттерна Chain of Responsibility

Для начала поймём, какие задачи умеет решать наша цепочка обязанностей и каждую из них определим соответствующей текстовой константой и создадим класс задач, который будет хранить всю информацию о событии — event. kind — это тип события.

```
E_FUEL, E_KM, E_OIL = "E_FUEL", "E_KM", "E_OIL"

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

Чтобы ввести три наших обработчика (handle_fuel, handle_k и handle_oil) в цепочку событий, для начала напишем нулевого обработчика, который будет обрабатывать события и в котором будет система передачи обработки событий следующим в очереди.

Назовём нулевой обработчик NullHandler. Метод __init__() будет зависеть от self и от класса (successor), которому надо передавать обработку, если он сам не сможет сделать. По умолчанию он равен None, то есть у нас нет следующего обработчика. Метод handle() будет передавать машину и событие следующему обработчику (если он существует).

```
class NullHandler:
    def __init__(self, successor=None):
        self.__successor = successor  # сохранили в класс

    def handle(self, car, event): # передаём обработку дальше
        if self.__successor is not None:
            self.__successor.handle(car, event)
```

Далее создаём классы Handler-ов: FuelHandler, KmHandler и OilHandler. Инициализация у них будет как и у предка, а вот метод handle сделаем у каждого свой. Там и проверяем, чтобы тип сообщения совпадал с обрабатываемым. Если это тот тип задач, которыми занимается текущий обработчик, то он выполняет задачу, а в противном случае передает следующему в цепочке (то есть выполняет функцию handle у предка для текущей машины и текущего задания)

```
class FuelHandler(NullHandler):
    def handle(self, car, event):
        if event.kind == E_FUEL:  # заливаем топливо
            if car.fuel < 10:
                print("Added fuel")
                car.fuel = 100
        else:
```

```
            super().handle(car, event)  # передаём
class KmHandler(NullHandler):
    def handle(self, car, event):
        if event.kind == E_KM:
            if car.km > 10000:
                print("Made a car test.")
                car.km = 0
        else:
            super().handle(car, event)

class OilHandler(NullHandler):
    def handle(self, car, event):
        if event.kind == E_OIL:
            if car.oil < 10:
                print("Added oil")
                car.oil = 100
        else:
            super().handle(car, event)
```

Теперь преобразуем класс CarService: будем передавать туда события. А в handle_car будем передавать машину и цепочку (handle) того, что надо сделать с машиной. При создании нового автосервиса, создаём две цепочки:

1. Простая цепочка, обрабатывающая по очереди:
   - не надо ли проверить бензин;
   - не надо ли проверить масло;
   - не надо ли проверить километраж.
2. Простая цепочка, дополненная выводом информации о запросе.

```
class CarService:
    def __init__(self):
        self.handlers = FuelHandler(OilHandler(KmHandler(
          NullHandler())))
        self.handlers_D = DebugHandler(self.handlers)
        self.events = []

    def add_event(self, event):
        self.events.append(event)

    def handle_car(self, car):
        for event in self.events:
            self.handlers.handle(car, event)
```

```
if __name__ == '__main__':
  # создаём набор событий
    events = [Event(E_FUEL), Event(E_KM), Event(E_OIL)]

  # создаём автосервис
    service = CarService()

  # Добавляем задания автосервису
    for event in events:
        service.add_event(event)

   handle=kmhandler(fuelhandler(oilhandler(Nullhandler)))
```

```
Added fuel
Made a car test.
Added oil
```

Как видим, всё заработало: сначала добавилось топливо, затем был проведён тест, и в конце залили масло.

Таким образом мы реализовали цепочку обязанностей, которую можно дополнять. Например, можно создать debug_handler и сделать его самым первым обработчиком, который будет сохранять в log-файл все события, приходящие на обработку.

Исходный код цепочки обязанностей

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.

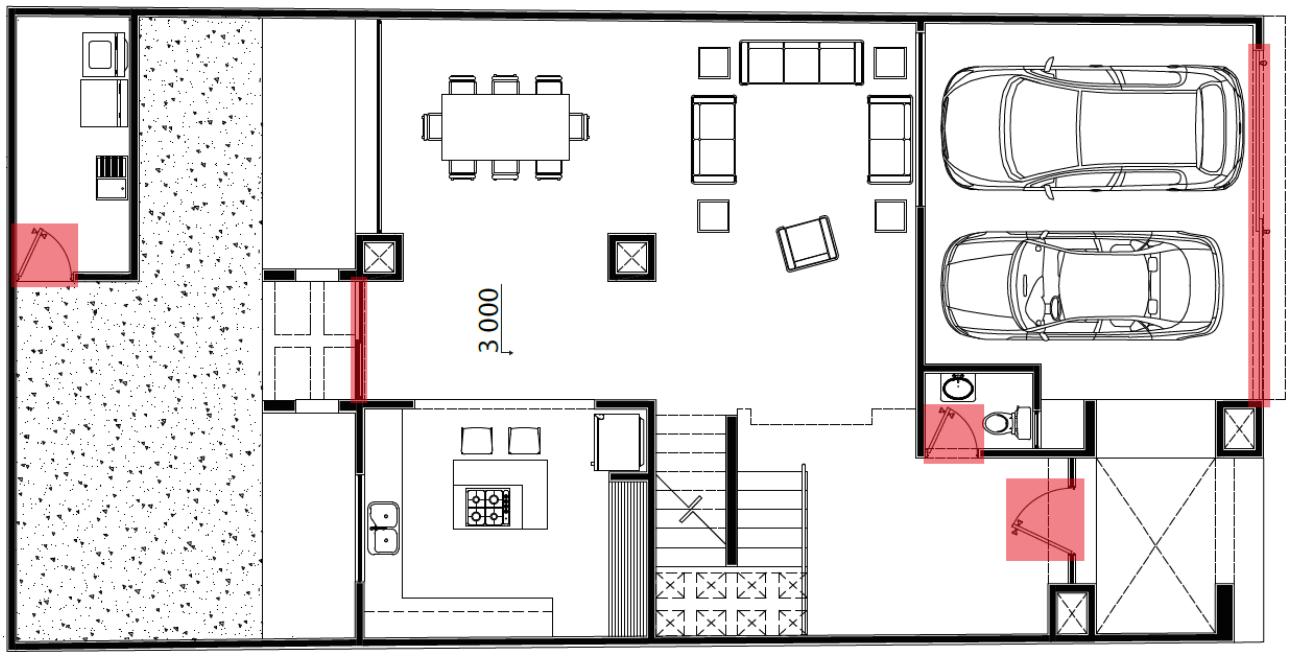

Рис. 4.4

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 Абстрактная фабрика

Импортируем библиотеку abs для работы с абстрактными классами. И в функции main опишем, что мы хотим от нашей программы — создание отчёта в двух форматах: hlml и markdown. Функция create_report() будет создавать отчёт. И для этого передаём в неё все составляющие отчёта: разделы, ссылки, картинки и прочее. То есть передавать мы всё будем именно фабрике и в первом случае назовём её MDreportFactory, а во втором HTMLreportFactory.

```
import abc  # для работы с абстрактным классом

def main():

    mdFilename = "report.md"
    HTMLFilename = "report.html"

    mdreport = create_report(MDreportFactory())
    mdreport.save(mdFilename)
    print("Saved:", mdFilename)

    HTMLreport = create_report(HTMLreportFactory())
    HTMLreport.save(HTMLFilename)
    print("Saved:", HTMLFilename)
```

Далее напишем функцию создания отчёта, которой передаётся просто factory (наша абстрактная фабрика, которая умеет создавать отчёты, их части и вставлять ссылки и картинки). Создадим отчёт из двух частей, в которые поместим текст, ссылки и изображение. В метод make_link надо передавать название ссылки и саму ссылку.

```
def create_report(factory):
    # создаём отчёт
    report = factory.make_report("Report")
    # создаём первую часть отчёта
    chapter1 = factory.make_chapter("Chapter one")
    # создаём вторую часть отчёта
    chapter2 = factory.make_chapter("Chapter two")

  # первая часть отчёта
    # добавляем текст
    chapter1.add("chapter 1 text")
    # создаём ссылку
    link = factory.make_link("coursera",
                             "https://ru.coursera.org")
    # добавляем ссылку
    chapter1.add(link)

  # вторая часть отчёта
    # добавляем текст
    chapter2.add("Chapter 2 header\n\n")
    # создаём картинку
    img = factory.make_img("image",
                           "https://blog.coursera.org/wp-
                             content/uploads/2017/07/
                             coursera-fb.png")
    # создаём ссылку из картинки
    link = factory.make_link(img,
                             "https://ru.coursera.org")
    # добавляем ссылку
    chapter2.add(link)
    # добавляем текст
    chapter2.add("\n\nChapter 2 footer")

  # отчёт
    report.add(chapter1)  # добавляем первую часть
    report.add(chapter2)  # добавляем вторую часть
    return report         # возвращаем отчёт
```

Теперь создадим абстрактную фабрику AbstractFactory. Поскольку все методы абстрактные, дописываем перед каждым @abc.abstractmethod. Методы: make_report() для создания отчёта с названием, make_chapter() для создания главы отчёта с заголовком, make_link() для создания ссылки (получающий на вход объект и саму гиперссылку) и make_img() для создания картинки из ис-

точника с описанием.

```
class ReportFactory(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def make_report(self, title):
        pass

    @abc.abstractmethod
    def make_chapter(self, caption):
        pass

    @abc.abstractmethod
    def make_link(self, obj, href):
        pass

    @abc.abstractmethod
    def make_img(self, alt_text, src):
        pass
```

Таким образом мы описали абсолютно абстрактный класс, в котором нет ничего, но который содержит все необходимые нам методы. На основании данной абстрактной фабрики создадим два класса: MDreportFactory (для отчёта в Markdown) и HTMLreportFactory (для отчёта в html).

```
class MDreportFactory(ReportFactory):

    def make_report(self, title):
        return MD_Report(title)

    def make_chapter(self, caption):
        return MD_Chapter(caption)

    def make_link(self, obj, href):
        return MD_Link(obj, href)

    def make_img(self, alt_text, src):
        return MD_Img(alt_text, src)


class HTMLreportFactory(ReportFactory):
    def make_report(self, title):
        return HTML_Report(title)

    def make_chapter(self, caption):
```

```
        return HTML_Chapter(caption)
    def make_link(self, obj, href):
        return HTML_Link(obj, href)

    def make_img(self, alt_text, src):
        return HTML_Img(alt_text, src)
```

Теперь реализуем все эти классы, каждый из которых занимается только своей вещью. Сначала для MDreportFactory:

```
# Отчёт
class MD_Report:

    # Создание отчёта
    def __init__(self, title):
        self.parts = []  # Создаём список всех частей отчёта
        # и сразу добавляем в список название
        self.parts.append(f"# {title}\n\n")

    # Добавление части
    def add(self, part):
        self.parts.append(part)  # Просто добавляем в список

    # Функция сохранения отчёта
    def save(self, filenameOrFile):
        # Определяем, что было передано: имя файла (file = None)
        # или файловая переменная (file = filenameOrFile).
        file = None if isinstance(filenameOrFile, str)
                                    else filenameOrFile

        try:  # Пытаемся произвести сохранения отчёта
            if file is None:
                # открываем файл для записи
                file = open(filenameOrFile,
                            "w", encoding="utf-8")

            # Печатаем в file все части, преобразованные к строке
            print('\n'.join(map(str, self.parts)),
                                        file=file)

        finally:  # Закрываем файл, если мы его открывали
            if isinstance(filenameOrFile, str) and file is
              not None:
                file.close()
```

```
# Часть
class MD_Chapter:
    # Создание части
    def __init__(self, caption):
        self.caption = caption  # Сохраняем заглавие
        self.objects = []       # Создаём список содержимого
                                # данной части отчёта

    # Добавление содержимого в отчёт
    def add(self, obj):
        self.objects.append(obj)  # Просто добавляем в список
    # Приобразование к строке: Отделяем заглавие, после чего
    # присоединяем всё содержимое, преобразованное к строке.
    def __str__(self):
        return (f'## {self.caption}\n\n' +
                        ''.join(map(str, self.objects)))

# Ссылка
class MD_Link:
    # Создание ссылки - сохранение объекта и адреса
    def __init__(self, obj, href):
        self.obj = obj
        self.href = href
    # Преобразование к строке - вывод ссылки в Markdown виде:
    def __str__(self):
        return f'[{self.obj}]({self.href})'

# Изображение
class MD_Img:
    # Создание изображения - сохранение его описания и расположения
    def __init__(self, alt_text, src):
        self.alt_text = alt_text
        self.src = src
    # Преобразование к строке - вывод ссылки в Markdown виде:
    def __str__(self):
        return f'![{self.alt_text}]({self.src})'
```

И теперь реализация методов для HTMLreportFactory

```
# Отчёт
class HTML_Report:

    # Создание отчёта
```

```
    def __init__(self, title):
        self.title = title
        # Создаём список частей отчёта и добавляем в него шапку:
        self.parts = []
        self.parts.append("<html>")  # Открывающий тег
        self.parts.append("<head>")  # Раздел заголовков
        # Название
        self.parts.append("<title>" + title + "</title>")
        self.parts.append("<meta charset=\"utf-8\">")
        self.parts.append("</head>")
        self.parts.append("<body>")  # Начало содержимого

    # Добавление части
    def add(self, part):
        self.parts.append(part)  # Просто добавляем в список

    # Функция сохранения отчёта
    def save(self, filenameOrFile):
        # Добавляем закрывающие тэги
        self.parts.append("</body></html>")
        # Определяем, что было передано:
        # имя файла (file = None) или
        # файловая переменная (file = filenameOrFile).
        file = None if isinstance(filenameOrFile, str) else
            filenameOrFile

        # Пытаемся произвести сохранения отчёта
        try:
            if file is None:
                # открываем файл для записи
                file = open(filenameOrFile,
                            "w", encoding="utf-8")

            # Печатаем в file все части, преобразованные к строке
            print(''.join(map(str, self.parts)),
                                    file=file)

        # Закрываем файл, если мы его открывали
        finally:
            if isinstance(filenameOrFile, str) and file is
              not None:
                file.close()
```

```
# Часть
class HTML_Chapter:

    # Создание части
    def __init__(self, caption):
        self.caption = caption  # Сохраняем заглавие
        # Создаём список содержимого данной части отчёта
        self.objects = []

    # Добавление содержимого в отчёт
    def add(self, obj):
        self.objects.append(obj)  # Просто добавляем в список

    # Приобразование к строке: обрамляем заглавие тэгами, после
    # чего присоединяем всё содержимое, преобразованное к строке.
    def __str__(self):
        ch = f'<h1>{self.caption}</h1>'
        return ch + ''.join(map(str, self.objects))

# Ссылка
class HTML_Link:
    # Создание ссылки - сохранение объекта и адреса
    def __init__(self, obj, href):
        self.obj = obj
        self.href = href
    # Преобразование к строке - вывод ссылки в HTML виде:

    def __str__(self):
        return f'<a href = "{self.href}">{self.obj}</a>'

# Изображение
class HTML_Img:
    # Создание изображения - сохранение его описания и расположения
    def __init__(self, alt_text, src):
        self.alt_text = alt_text
        self.src = src
    # Преобразование к строке - вывод ссылки в виде HTML виде:

    def __str__(self):
        return f'<img alt = "{self.alt_text}",
                              src = "{self.src}"/>'
```

И в заключение, проверим работу:

```
from IPython.display import display, Markdown, HTML
test()
display(Markdown('# <span style="color:red">report.md</span>'))
display(Markdown(filename="report.md"))
display(Markdown('# <span style="color:red">report.html</span>'))
display(HTML(filename="report.html"))
```

```
Сохранено: report.md
Сохранено: report.html
```

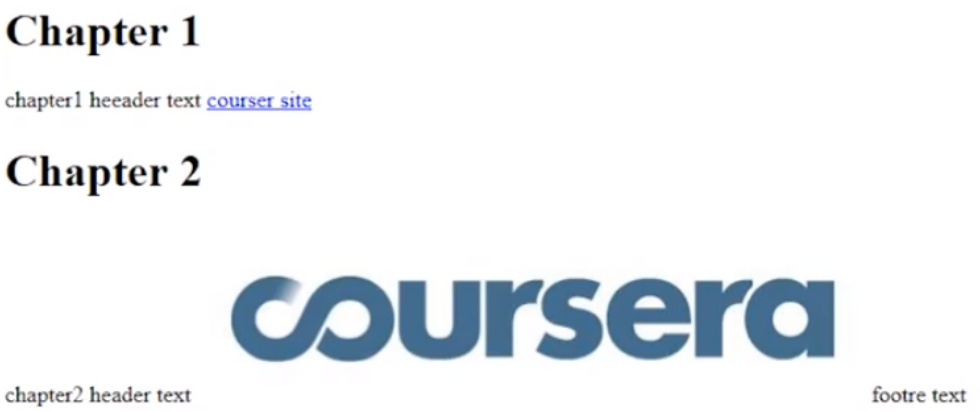

Рис. 4.5: Результат работы HTMLreportFactory

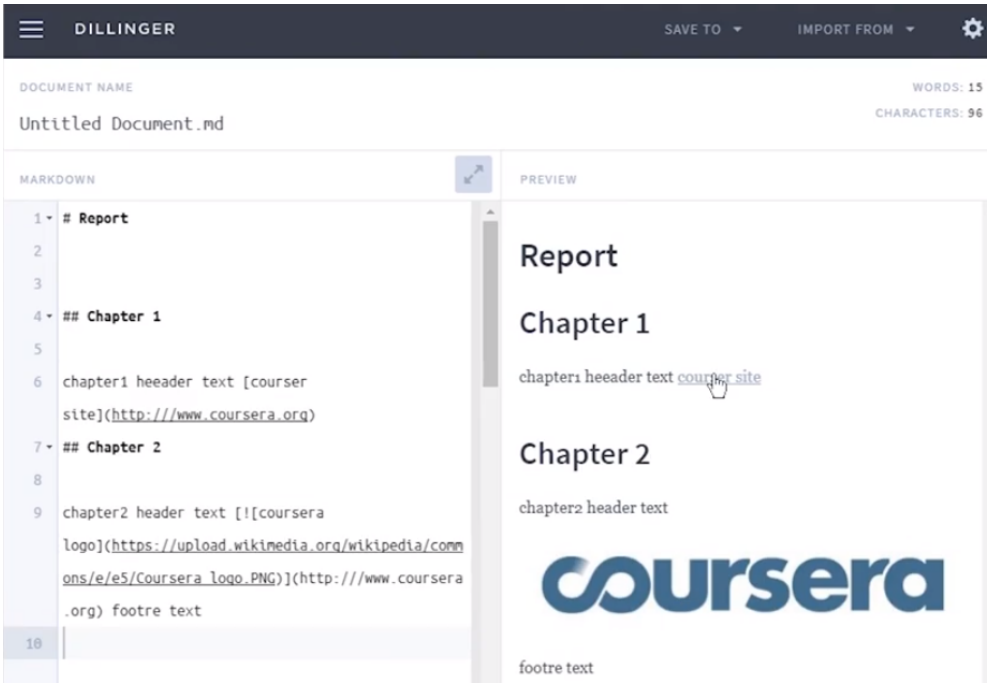

Рис. 4.6: Результат работы MDreportFactory

Но, есть нюанс: Python такой язык, что класс AbstractFactory можно просто удалить, ведь мы все равно передаём уже готовую (полноценный класс) фабрику MDreportFactory или HTMLreportFactory. А знать, какие методы там будут, и не надо. Единственный случай когда это необходимо, это если мы описываем абстрактную фабрику и если мы забыли реализовать какой-то метод, Python скажет, что мы не реализовали.

Посмотрим на MDreportFactory и HTMLreportFactory — они практически идентичны. Используя свойства языка Python мы можем сократить код.

Полная реализация Абстрактной фабрики

4.2.3 Практическая реализация паттерна Абстрактная фабрика

Теперь преобразуем абстрактную фабрику в классическом виде в абстрактную фабрику в стиле Python. Всё, что раньше было в MDreportFactory и HTMLreportFactory классовыми методами и поместим в абстрактную фабрику (которую теперь сделаем ReportFactory). Каждый метод будет вместо self получать Class и возвращать соответствующий тип объекта.

```
class ReportFactory():

    @classmethod                            # Отчёт
    def make_report(Class, title):
        return Class.Report(title)

    @classmethod                            # Часть
    def make_chapter(Class, caption):
        return Class.Chapter(caption)

    @classmethod                            # Ссылка
    def make_link(Class, obj, href):
        return Class.Link(obj, href)

    @classmethod                            # Картинка
    def make_img(Class, alt_text, src):
        return Class.Img(alt_text, src)
```

Теперь MDreportFactory и HTMLreportFactory будут потомками ReportFactory и все их методы не нужны. Но тогда все классы (MDreport, MDchapter, MDlink и MDimg и аналогичные у HTML) необходимо будет сделать составной частью соответствующего класса. И назвать их просто chapter, report, link и img.

Таким образом мы общую часть вынесли в тот класс, который раньше был абстрактным (ReportFactory), а реализацию сделали внутренними классами соответствующих классов.

```
class MDreportFactory(ReportFactory):
    # Отчёт
    class Report:

        # Создание отчёта
        def __init__(self, title):
            self.parts = []  # Создаём список всех частей
            # отчёта и сразу добавляем в список название
            self.parts.append(f"# {title}\n\n")

        # Добавление части
        def add(self, part):
            self.parts.append(part)  # Добавляем в список

        # Функция сохранения отчёта
        def save(self, filenameOrFile):
            # Определяем, что было передано:
            # имя файла (file = None)
            # или файловая переменная (file = filenameOrFile).
            file = None if isinstance(filenameOrFile,str)
                                    else filenameOrFile

            # Пытаемся произвести сохранения отчёта
            try:
                if file is None:
                    # открываем файл для записи
                    file = open(filenameOrFile,
                                "w", encoding="utf-8")

            # Печатаем в file все части, преобразованные к строке
                print('\n'.join(map(str, self.parts)), file
                  =file)

            # Закрываем файл, если мы его открывали
            finally:
                if (isinstance(filenameOrFile, str) and
                            file is not None):
                    file.close()
```

```
# Часть
    class Chapter:

        # Создание части
        def __init__(self, caption):
            self.caption = caption  # Сохраняем заглавие
            # Создаём список содержимого данной части отчёта
            self.objects = []

    # Добавление содержимого в отчёт
        def add(self, obj):
            self.objects.append(obj)  # Добавляем в список

    # Преобразование к строке: отделяем заглавие, после чего
    # присоединяем всё содержимое, преобразованное к строке.
        def __str__(self):
            return f'## {self.caption}\n\n' +
                        ''.join(map(str, self.objects))

# Ссылка
    class Link:
        # Создание ссылки - сохранение объекта и адреса
        def __init__(self, obj, href):
            self.obj = obj
            self.href = href

    # Преобразование к строке - вывод ссылки в Markdown виде:
        def __str__(self):
            return f'[{self.obj}]({self.href})'

# Изображение
    class Img:
        # Создание (сохранение описания и расположения)
        def __init__(self, alt_text, src):
            self.alt_text = alt_text
            self.src = src

    # Преобразование к строке - вывод ссылки в Markdown виде:
        def __str__(self):
            return f'![{self.alt_text}]({self.src})'
```

Теперь аналогично поступим с HTMLreportFactory

```
class HTMLreportFactory(ReportFactory):
    # Отчёт
    class Report:
        def __init__(self, title):# Создание отчёта
            self.title = title
            # Создаём список всех частей отчёта
            # и добавляем в него шапку:
            self.parts = []
            self.parts.append("<html>")  # Открывающий тег
            self.parts.append("<head>")  # Раздел заголовков

            self.parts.append("<title>" +
                              title + "</title>")  #Название
            self.parts.append("<meta charset=\"utf-8\">")
            self.parts.append("</head>")
            self.parts.append("<body>") # Начало содержимого

        # Добавление части
        def add(self, part):
            self.parts.append(part)  # Добавляем в список

        # Функция сохранения отчёта
        def save(self, filenameOrFile):
            # Добавляем закрывающие тэги
            self.parts.append("</body></html>")

            # Определяем, что было передано:
            # имя файла (file = None) или
            # файловая переменная (file = filenameOrFile).
            file = None if isinstance(filenameOrFile,
                                      str) else filenameOrFile

        # Пытаемся произвести сохранение отчёта
            try:
                if file is None:
                    # открываем файл для записи
                    file = open(filenameOrFile,
                                "w", encoding="utf-8")

            # Печатаем в file все части, преобразованные к строке
                print(''.join(map(str, self.parts)),
                                        file=file)
```

```
        # Закрываем файл, если мы его открывали
            finally:
                if (isinstance(filenameOrFile, str) and
                   file is not None):
                    file.close()

# Часть
    class Chapter:
        def __init__(self, caption): # Создание части
            self.caption = caption  # Сохраняем заглавие
            # Создаём список содержимого данной части отчёта
            self.objects = []

        # Добавление содержимого в отчёт
        def add(self, obj):
            self.objects.append(obj)  # Добавляем в список

        # Преобразование к строке: обрамляем заглавие тэгами и
        # присоединяем всё содержимое, преобразованное к строке.
        def __str__(self):
            ch = f'<h1>{self.caption}</h1>'
            return ch + ''.join(map(str, self.objects))
# Ссылка
    class Link:
        def __init__(self, obj, href):  # Создание ссылки
            self.obj = obj  # сохранение объекта и адреса
            self.href = href

    # Преобразование к строке - вывод ссылки в HTML виде:
        def __str__(self):
            return f'<a href =
                           "{self.href}">{self.obj}</a>'

# Изображение
    class Img:
        def __init__(self, alt_text, src):
            self.alt_text = alt_text
            self.src = src
    # Преобразование к строке - вывод ссылки в виде HTML виде:
        def __str__(self):
            return f'<img alt = "{self.alt_text}",
                                    src = "{self.src}"/>'
```

При использовании классовых методов абстрактная фабрика сокращается.
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;

```json
{
  "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 языков программирования;
- 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

Разберёмся, как можно сконфигурировать написанную ранее абстрактную фабрику для создания отчётов при помощи YAML-файла. Для начала рассмотрим простой YAML-файл:

```
--- !Report
factory:
  !factory HTML
filename:
  report.txt
```

Фабрика может быть двух типов: HTML и MD (markdown). Импортируем yaml-файл, который говорит о том, что у класса Report будет два поля: factory (содержит фабрику) и поле filename (содержит имя файла, в который сохраняем). Для начала работы с YAML-файлом необходимо создать класс, который будет по нему строиться. Возьмём за основу итоговую версию кода абстрактной фабрики. Здесь мы переделаем функцию main(): откроем yaml-файл и создадим отчёт report как результат обработки yaml-файла.

```
import abc
import yaml

# функция, преобразующая написанное в нужную фабрику
def factory(loader, node):
    format = loader.construct_scalar(node)
    if format == "HTML":  # создаём html фабрику
        return HTMLreportFactory()
    else:  # создаём markdown фабрику
        return MDreportFactory()

def main():
  #делаем свой загрузчик, понимающий тип factory
    file = open("factory.yml")
    # создаём объект стандартного загрузчика yaml
    loader = yaml.Loader
    # добавляем конструктор
    loader.add_constructor("!factory", factory)

    # загружаем yaml файл отчёта
    report = yaml.load(file.read())
    # создаём объект, который уже можно сохранить
    report.create_report().save(report.filename)
    print("Saved:", HTMLreport.filename)  # вывод
    file.close()  # не забываем закрыть файл
```

Теперь ReportFactory - потомок yaml.YAMLObject. Сделано это для того, чтобы yaml обработчик знал новый тип данных, указанный в yaml_tag (он будет определён в фабриках - потомках). Класс Report мы будем создавать на основе yaml-файла.

Всю функцию create_report помещаем внутрь класса Report. Поскольку factory будет полем внутри нашего класса, внутри функции create_report надо заменить "factory"на "self.factory то есть мы будем пользоваться фабрикой, которая создастся во время инициализации класса Report (остальной код не меняли).

```
class Report(yaml.YAMLObject):
    yaml_tag = u"!Report"  # первая строка yaml-файла

    def create_report(self):
        report = self.factory.make_report("Report")
        chapter1 = self.factory.make_chapter("Chapter 1")
        chapter2 = self.factory.make_chapter("Chapter 2")

        chapter1.add("chapter 1 text")
        link = self.factory.make_link("coursera",
                                      "https://ru.coursera.org")
        chapter1.add(link)

        chapter2.add("Chapter 2 header\n\n")
        # создаём картинку
        img = self.factory.make_img("image",
                                    "https://blog.coursera.org/wp-
                                      content/uploads/2017/07/coursera-
                                      fb.png")

        link = self.factory.make_link(img,
                                      "https://ru.coursera.org")
        chapter2.add(link)
        chapter2.add("\n\nChapter 2 footer")


        report.add(chapter1)  # добавляем первую часть
        report.add(chapter2)  # добавляем вторую часть
        return report         # возвращаем отчёт
```

В итоге мы получили html код в файле report.txt, который ожидали:

```
<html><head><title>Report</title></head><body>
<h1>Chapter 1</h1>chapter1 header text <a  href="http:///
www.coursera.org">coursera site</a>
```

```
<h1>Chapter 2</h1>chapter2 header text <a href="http:///
www.coursera.org">coursera site><img alt="coursera logo",
src = "https://blog.coursera.org/wp-content/uploads
  /2017/07/coursera-fb.png"/></a> footre text
</body></html>
```

Теперь наша программа конфигурируется при помощи yaml-файла и выбирает, какую именно фабрику ей надо загружать. Но конфигурирование может быть гораздо сложнее. Например, файл, который записывает полноценную конфигурацию внутри отчёта. Здесь задаётся имя файла, содержимое с названием и частями: chapter 1 и chapter 2, в которых в свою очередь хранится какой-то текст, ссылки и картинка. В нём очень хорошо видна древовидная структура содержимого отчёта. Перечисления показаны знаком минус в начале строки. А восклицательный знак перед именем это тип данных, который хранится. Для такого файла надо писать больше обработчиков. Кроме уже написанных нужно написать обработчик chapter, link и img.

```
objects:
  - &img !img
      alt_text: google
      src: "https://blog.coursera.org/wp-content/uploads
        /2017/07/coursera-fb.png"
report: !report
  filename: report_yaml.html
  title: Report
  parts:
    - !chapter
      caption: "chapter one"
      parts:
        - "chapter 1 text"
        - !link
            obj: coursera
            href: "https://ru.coursera.org"
    - !chapter
      caption: "chapter two"
      parts:
        - "Chapter 2 header"
        - !link
            obj: *img
            href: "https://ru.coursera.org"
        - "Chapter 2 footer"
```

Подробный разбор этого YAML файла представлен в материалах к уроку