Лабораторная работа №2
Шаблоны проектирования

Цель работы: ознакомиться с основными шаблонами проектирования, научиться применять их при проектировании и разработке ПО.

Необходимое ПО для практической части: JDK 8; IntelliJ IDEA 14 Community Edition; плагин PlantUML integration; Graphviz.

Теоретические сведения

Шаблон проектирования или паттерн (англ. design pattern) в разработке программного обеспечения — повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях. Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие именно конечные классы или объекты приложения будут использоваться.

Сообразное использование паттернов проектирования дает разработчику ряд неоспоримых преимуществ. Приведем некоторые из них. Модель системы, построенная в терминах паттернов проектирования, фактически является структурированным выделением тех элементов и связей, которые значимы при решении поставленной задачи. Помимо этого, модель, построенная с использованием паттернов проектирования, более проста и наглядна в изучении, чем стандартная модель. Тем не менее, несмотря на простоту и наглядность, она позволяет глубоко и всесторонне проработать архитектуру разрабатываемой системы с использованием специального языка. Применение паттернов проектирования повышает устойчивость системы к изменению требований и упрощает неизбежную последующую доработку системы. Кроме того, трудно переоценить роль использования паттернов при интеграции информационных систем организации. Также следует упомянуть, что совокупность паттернов проектирования, по сути, представляет собой единый словарь проектирования, который, будучи унифицированным средством, незаменим для общения разработчиков друг другом.

Но самое главное - любой шаблон проектирования может стать палкой о двух концах: если он будет применен не к месту, это может обернуться катастрофой и создать вам много проблем в последующем. В то же время, реализованный в нужном месте, в нужное время, он может стать для вас настоящим спасителем.

Есть три основных вида шаблонов проектирования:

- структурные;
- порождающие;
- поведенческие.

Структурные шаблоны определяют различные сложные структуры, которые изменяют интерфейс уже существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу.

Порождающие шаблоны - шаблоны проектирования, которые абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять инстанцируемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту.

Поведенческие шаблоны определяют взаимодействие между объектами, увеличивая таким образом его гибкость.

Нужно учесть, что шаблонов очень много, и рассмотрение всех шаблонов может занять целый учебный курс. Поэтому ниже кратко рассмотрены лишь *некоторые* из существующих паттернов. Если будет необходима дополнительная информация о конкретном шаблоне, то обращайтесь к литературе.

Структурные шаблоны

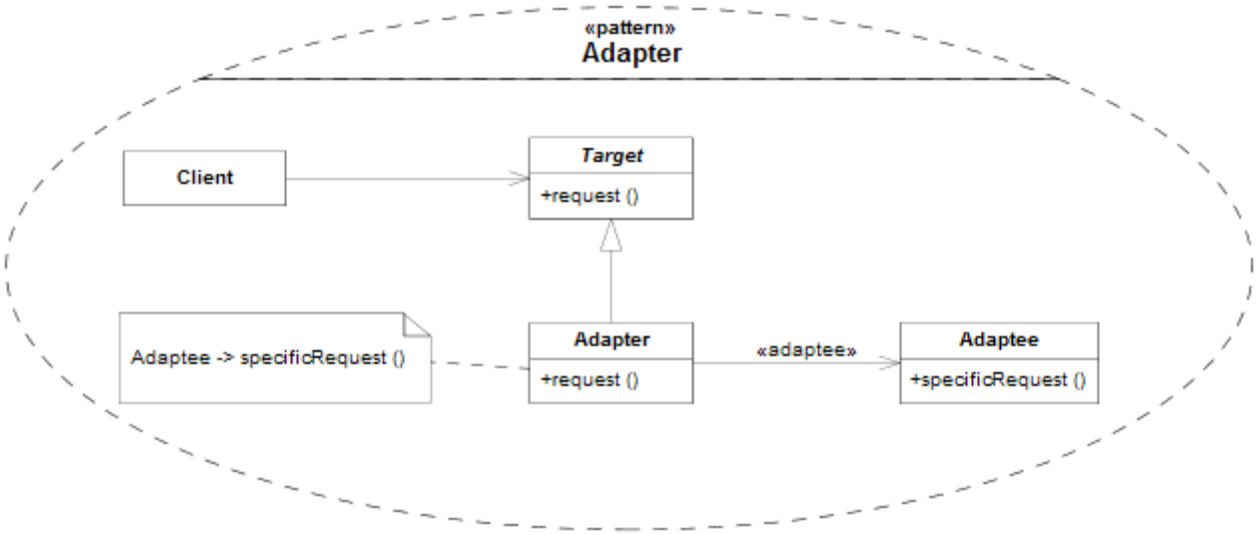
Адаптер (Adapter)

Проблема: необходимо обеспечить взаимодействие несовместимых интерфейсов или создать единый устойчивый интерфейс для нескольких компонентов с разными интерфейсами.

Решение: преобразовать исходный интерфейс компонента к другому виду с помощью промежуточного объекта - адаптера, то есть, добавить специальный объект с общим интерфейсом в рамках данного приложения и перенаправить связи от внешних объектов к этому объекту - адаптеру.

Класс *Adapter* приводит интерфейс класса *Adaptee* в соответствие с интерфейсом класса *Target* (наследником которого является *Adapter*). Это позволяет объекту *Client* использовать объект *Adaptee* (посредством адаптера *Adapter*) так, словно он является экземпляром класса *Target*.

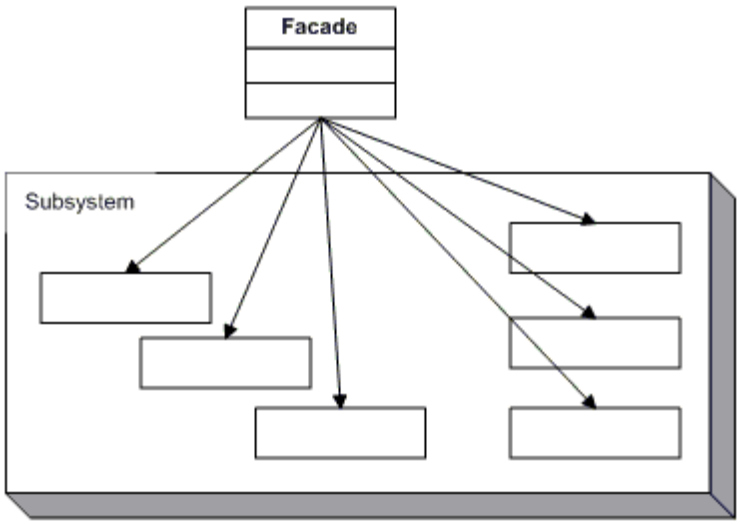
Таким образом *Client* обращается к интерфейсу *Target*, реализованному в наследнике *Adapter*, который перенаправляет обращение к *Adaptee*.



Шаблон Адаптер позволяет включать уже существующие объекты в новые объектные структуры, независимо от различий в их интерфейсах.

Этот шаблон позволяет в процессе проектирования не принимать во внимание возможные различия в интерфейсах уже существующих классов. Если есть класс, обладающий требуемыми методами и свойствами (по крайней мере, концептуально), то при необходимости всегда можно воспользоваться шаблоном Адаптер для приведения его интерфейса к нужному виду.

Фасад (Facade)



Шаблон “фасад” - структурный шаблон проектирования, позволяющий скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

Проблема: как обеспечить унифицированный интерфейс с набором разрозненных реализаций или интерфейсов, например, с подсистемой, если нежелательно высокое связывание с этой подсистемой или реализация подсистемы может измениться?

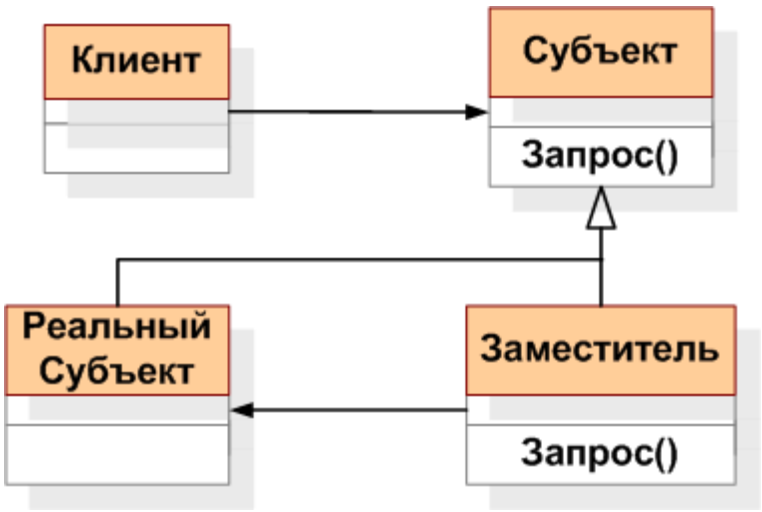
Решение: определить одну точку взаимодействия с подсистемой — фасадный объект, обеспечивающий общий интерфейс с подсистемой, и возложить на него обязанность по взаимодействию с её компонентами. Фасад — это внешний объект, обеспечивающий единственную точку входа для служб подсистемы. Реализация других компонентов подсистемы закрыта и не видна внешним компонентам.

Заместитель (Proxy)

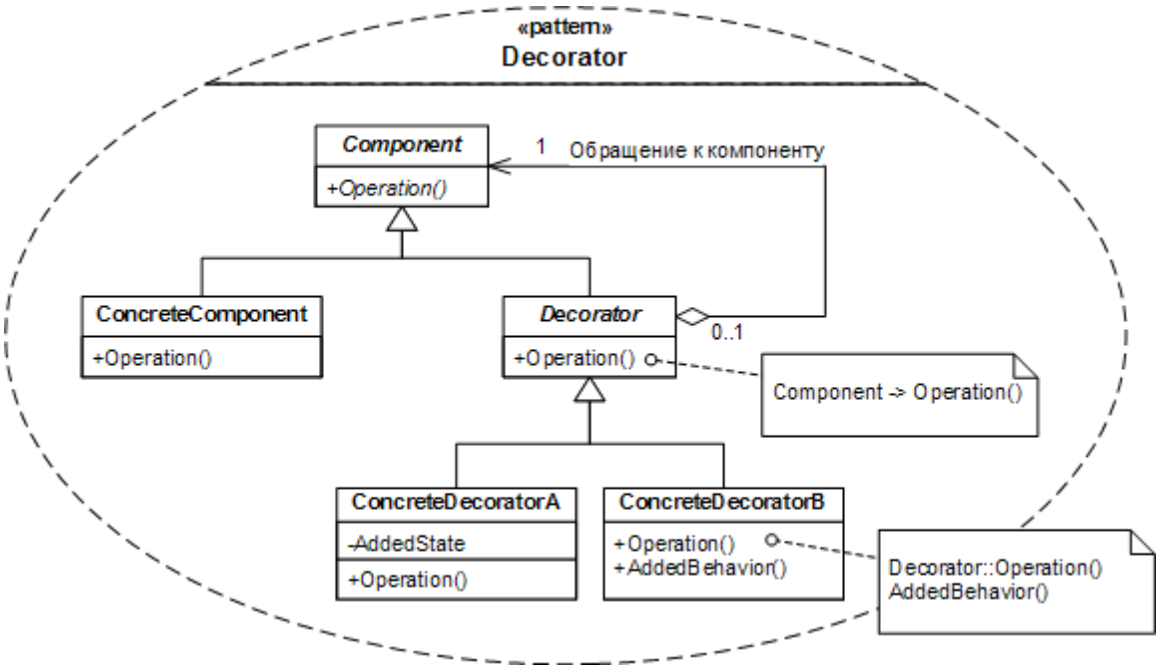
Заместитель — структурный шаблон проектирования, который предоставляет объект, который контролирует доступ к другому объекту, перехватывая все вызовы.

Проблема: необходимо управлять доступом к объекту так, чтобы не создавать громоздкие объекты «по требованию».

Решение: создать суррогат громоздкого объекта. «Заместитель» хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту (объект класса «Заместитель» может обращаться к объекту класса «Субъект», если интерфейсы «Реального Субъекта» и «Субъекта» одинаковы). Поскольку интерфейс «Реального Субъекта» идентичен интерфейсу «Субъекта», так, что «Заместителя» можно подставить вместо «Реального Субъекта». «Заместитель» контролирует доступ к «Реальному Субъекту», может отвечать за создание или удаление «Реального Субъекта». «Субъект» определяет общий для «Реального Субъекта» и «Заместителя» интерфейс, так, что «Заместитель» может быть использован везде, где ожидается «Реальный Субъект». При необходимости запросы могут быть переадресованы «Заместителем» «Реальному Субъекту».



Декоратор (Decorator)



Декоратор — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. Шаблон Декоратор предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.

Задача: объект, который предполагается использовать, выполняет основные функции. Однако может потребоваться добавить к нему некоторую дополнительную функциональность, которая будет выполняться до, после или даже вместо основной функциональности объекта.

Решение: шаблон “декоратор” предусматривает расширение функциональности объекта без определения подклассов.

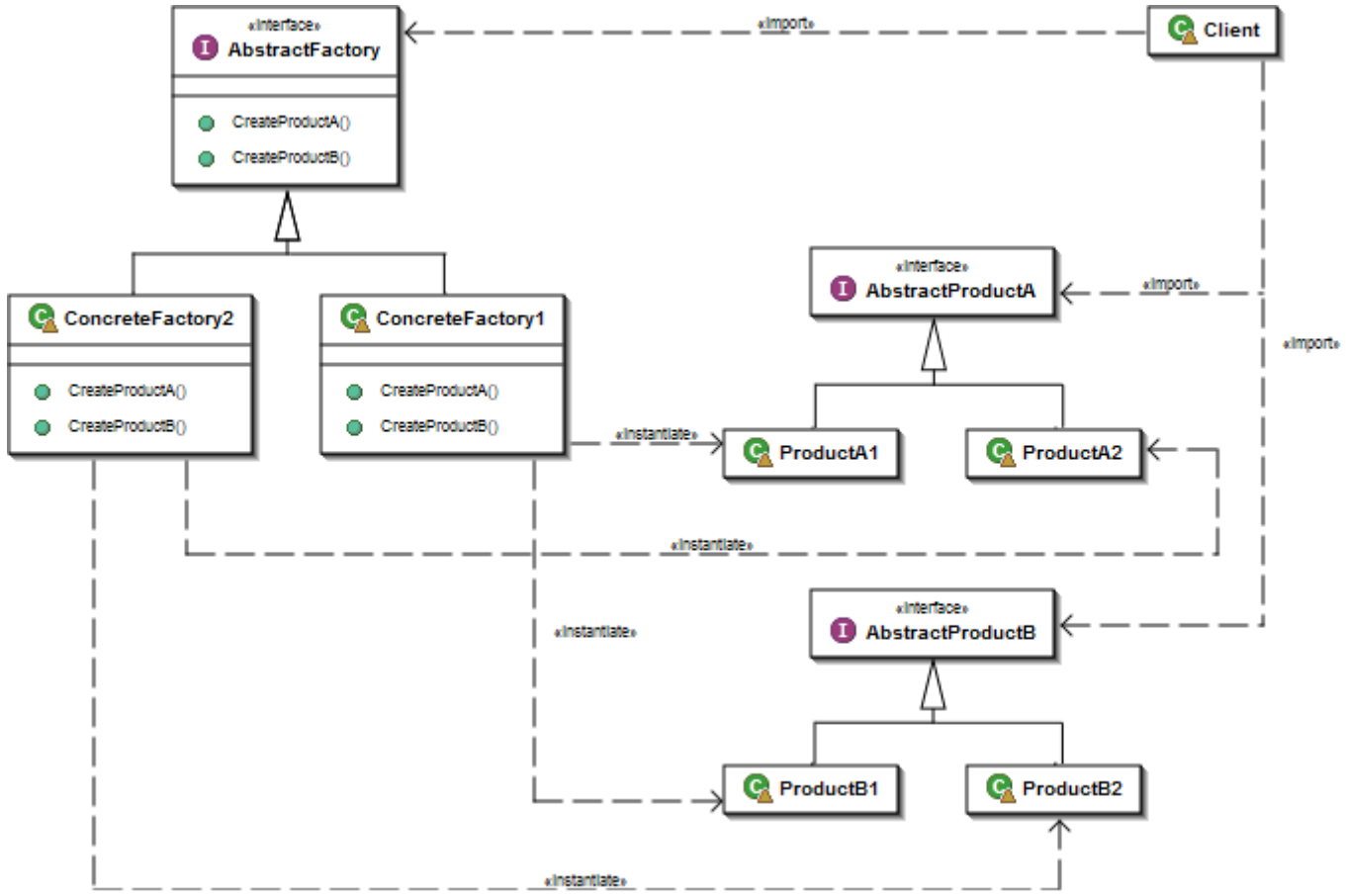
Класс *ConcreteComponent* — класс, в который с помощью шаблона Декоратор добавляется новая функциональность. В некоторых случаях базовая функциональность предоставляется классами, производными от класса *ConcreteComponent*. В подобных случаях класс *ConcreteComponent* является уже не конкретным, а абстрактным. Абстрактный класс *Component* определяет интерфейс для использования всех этих классов.

Порождающие шаблоны

Абстрактная фабрика (Abstract Factory)

Абстрактная фабрика — порождающий шаблон проектирования, позволяющий изменять поведение системы, варьируя создаваемыми объектами, при этом сохраняя интерфейсы. Он позволяет создавать целые группы взаимосвязанных объектов, которые, будучи созданными одной фабрикой, реализуют общее поведение. Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса он может создавать окна и кнопки). Затем пишутся классы, реализующие этот интерфейс.

Этот шаблон предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.



Фабричный метод (Factory Method)

Фабричный метод — порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, Фабрика делегирует создание объектов наследникам

родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

Шаблон определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать создание подклассов. Используется, когда:

- классу заранее неизвестно, объекты каких подклассов ему нужно создавать.
- класс спроектирован так, чтобы объекты, которые он создаёт, специфицировались подклассами.
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и планируется локализовать знание о том, какой класс принимает эти обязанности на себя.

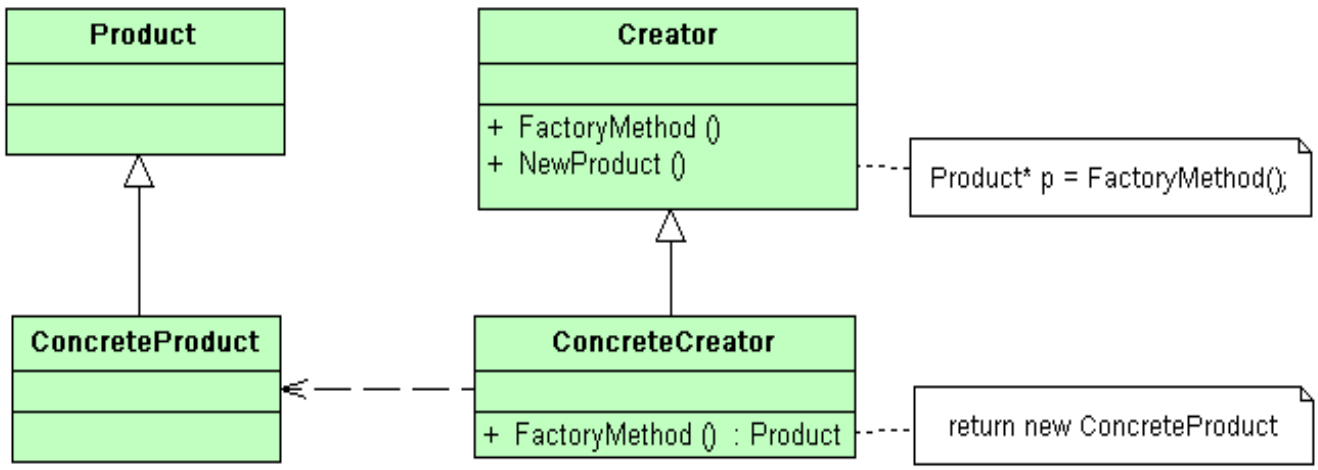
Структура:

Product — продукт; определяет интерфейс объектов, создаваемых абстрактным методом.

ConcreteProduct — конкретный продукт, реализует интерфейс *Product*.

Creator — создатель; объявляет фабричный метод, который возвращает объект типа *Product*. Может также содержать реализацию этого метода «по умолчанию»; может вызывать фабричный метод для создания объекта типа Product.

ConcreteCreator — конкретный создатель; переопределяет фабричный метод таким образом, чтобы он создавал и возвращал объект класса *ConcreteProduct*.



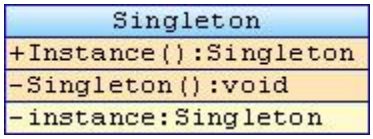
Одиночка (Singleton)

Одиночка — порождающий шаблон проектирования, гарантирующий, что в однопоточном приложении будет единственный экземпляр класса с глобальной точкой доступа.

Существенно то, что можно пользоваться именно *экземпляром* класса, так как при этом во многих случаях становится доступной более широкая функциональность.

Глобальный «одинокий» объект — именно объект, а не набор процедур, не привязанных ни к какому объекту — бывает нужен:

- если используется существующая объектно-ориентированная библиотека;
- если есть шансы, что один объект когда-нибудь превратится в несколько;
- если интерфейс объекта (например, игрового мира) слишком сложен, и не стоит засорять основное пространство имён большим количеством функций;
- если, в зависимости от каких-нибудь условий и настроек, создаётся один из нескольких объектов. Например, в зависимости от того, ведётся лог или нет, создаётся или настоящий объект, пишущий в файл, или «заглушка», ничего не делающая.



Поведенческие шаблоны

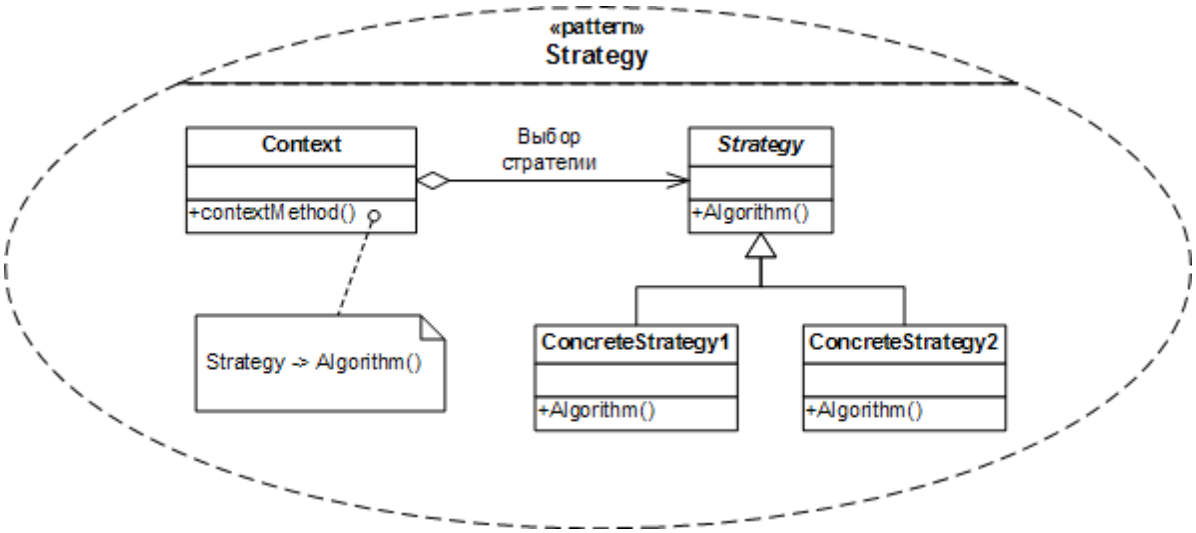
Стратегия (Strategy)

Стратегия — поведенческий шаблон проектирования, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путем определения соответствующего класса. Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.

Проблема: по типу клиента (или по типу обрабатываемых данных) выбрать подходящий алгоритм, который следует применить. Если используется правило, которое не подвержено изменениям, нет необходимости обращаться к шаблону «стратегия».

Решение: отделение процедуры выбора алгоритма от его реализации. Это позволяет сделать выбор на основании контекста.

Класс Strategy определяет, как будут использоваться различные алгоритмы. Конкретные классы ConcreteStrategy реализуют эти различные алгоритмы. Класс Context использует конкретные классы ConcreteStrategy посредством ссылки на конкретный тип абстрактного класса Strategy. Классы Strategy и Context взаимодействуют с целью реализации выбранного алгоритма (в некоторых случаях классу Strategy требуется посылать запросы классу Context). Класс Context пересылает классу Strategy запрос, поступивший от его класса-клиента.



Наблюдатель (Observer)

Наблюдатель — поведенческий шаблон проектирования. Создает механизм у класса, который позволяет получать экземпляру объекта этого класса оповещения от других объектов об изменении их состояния, тем самым наблюдая за ними.

Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

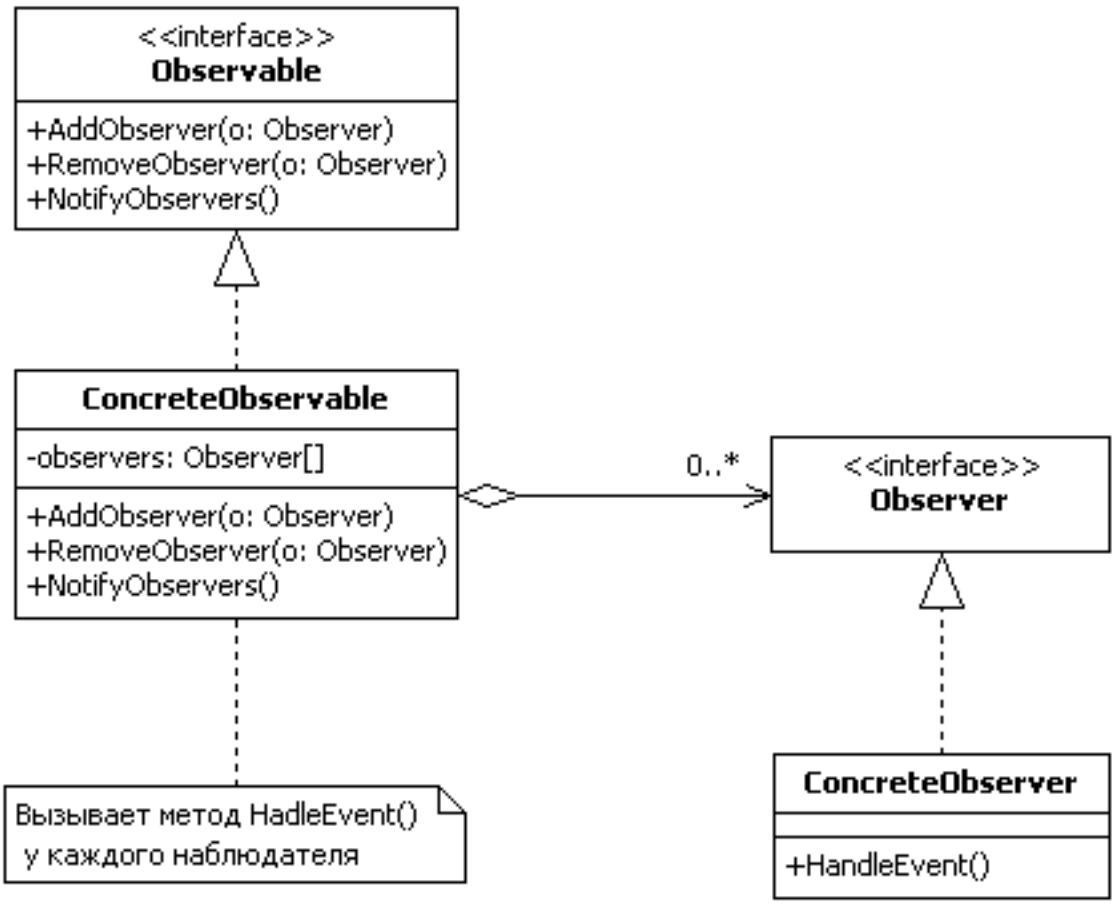
При реализации шаблона «наблюдатель» обычно используются следующие классы:

- **Observable** — интерфейс, определяющий методы для добавления, удаления и оповещения наблюдателей;
- **Observer** — интерфейс, с помощью которого наблюдатель получает оповещение;
- **ConcreteObservable** — конкретный класс, который реализует интерфейс Observable;
- **ConcreteObserver** — конкретный класс, который реализует интерфейс Observer.

Шаблон «наблюдатель» применяется в тех случаях, когда система обладает следующими свойствами:

- существует, как минимум, один объект, рассылающий сообщения;
- имеется не менее одного получателя сообщений, причём их количество и состав могут изменяться во время работы приложения;
- нет надобности очень сильно связывать взаимодействующие объекты, что полезно для повторного использования.

Данный шаблон часто применяют в ситуациях, в которых отправителя сообщений не интересует, что делают получатели с предоставленной им информацией.

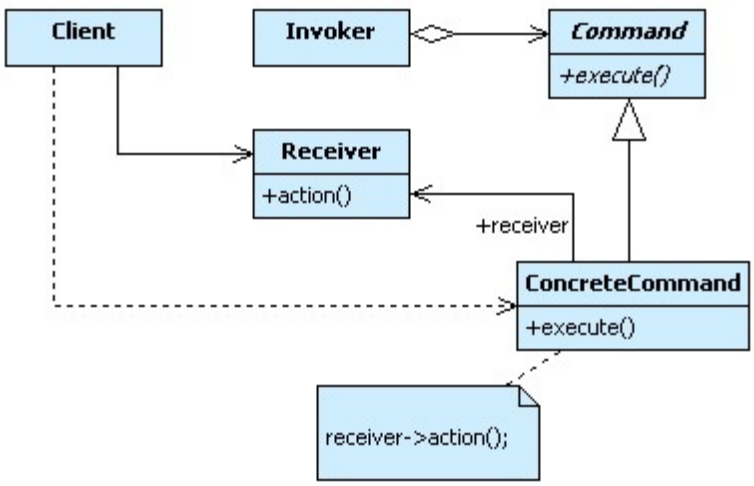


Команда (Command)

Команда — поведенческий шаблон проектирования, используемый при объектно-ориентированном программировании, представляющий действие. Объект команды заключает в себе само действие и его параметры.

Паттерн обеспечивает обработку команды в виде объекта, что позволяет сохранять её, передавать в качестве параметра методам, а также возвращать её в виде результата, как и любой другой объект.

Паттерн Command преобразовывает запрос на выполнение действия в отдельный объект-команду. Такая инкапсуляция позволяет передавать эти действия другим функциям и объектам в качестве параметра, приказывая им выполнить запрошенную операцию. Команда – это объект, поэтому над ней допустимы любые операции, что и над объектом.



Интерфейс командного объекта определяется абстрактным базовым классом Command и в самом простом случае имеет единственный метод execute(). Производные классы определяют получателя

запроса (указатель на объект-получатель) и необходимую для выполнения операцию (метод этого объекта). Метод execute() подклассов Command просто вызывает нужную операцию получателя.

В паттерне Command может быть до трех участников:

- Клиент, создающий экземпляр командного объекта.
- Инициатор запроса, использующий командный объект.
- Получатель запроса.

Сначала клиент создает объект ConcreteCommand, конфигурируя его получателем запроса. Этот объект также доступен инициатору. Инициатор использует его при отправке запроса, вызывая метод execute(). Этот алгоритм напоминает работу функции обратного вызова в процедурном программировании – функция регистрируется, чтобы быть вызванной позднее.

Паттерн Command отделяет объект, инициирующий операцию, от объекта, который знает, как ее выполнить. Единственное, что должен знать инициатор, это как отправить команду. Это придает системе гибкость: позволяет осуществлять динамическую замену команд, использовать сложные составные команды, осуществлять отмену операций.

Задания для самостоятельной работы

В каждом из вариантов указан шаблон для реализации и проект, использующий этот шаблон. Необходимо сделать следующее:

1. Нарисовать в PlantUML диаграмму классов реализуемой программы. (проектирование)
2. Реализовать программу на Java. (реализация)

Для каждого из шаблонов, предложенных в вариантах можно найти пример реализации UML-схемы и кода в приложенной книге “Паттерны проектирования”. Также указана глава, где подробно описан данный шаблон.

Вариант №1, 9, 17, 25

Шаблон “Стратегия”. Проект “Принтеры”. В проекте должны быть реализованы разные модели принтеров, которые выполняют разные виды печати. Пример использования шаблона в главе 1.

Вариант №2, 10, 18, 26

Шаблон “Наблюдатель”. Проект “Оповещение постов ГАИ”. В проекте должна быть реализована отправка сообщений всем постам ГАИ. Пример использования шаблона в главе 2.

Вариант №3, 11, 19, 27

Шаблон “Декоратор”. Проект “Универсальная электронная карта”. В проекте должна быть реализована универсальная электронная карта, в которой есть функции паспорта, страхового полиса, банковской карты и т. д. Пример использования шаблона в главе 3.

Вариант №4, 12, 20, 28

Шаблон “Фабричный метод”. Проект “Фабрика смартфонов”. В проекте должно быть реализовано создание смартфонов с различными характеристиками. Пример использования шаблона в главе 4.

Вариант №5, 13, 21, 29

Шаблон “Абстрактная фабрика”. Проект “Заводы по производству автомобилей”. В проекте должно быть реализована возможность создавать автомобили различных типов на разных заводах. Пример использования шаблона в главе 4.

Вариант №6, 14, 22, 30

Шаблон “Команда”. Проект “Клавиатура настраимаемого калькулятора”. Цифровые и арифметические кнопки имеют фиксированную функцию, а остальные могут менять своё назначение. Пример использования шаблона в главе 6.

Вариант №7, 15, 23

Шаблон “Адаптер”. Проект “Часы”. В проекте должен быть реализован адаптер, который дает возможность пользоваться часами со стрелками так же, как и цифровыми часами. В классе “Часы со стрелками” хранятся повороты стрелок. Пример использования шаблона в главе 7.

Вариант №8, 16, 24

Шаблон “Фасад”. Проект “Компьютер”. В проекте должен быть реализован “компьютер”, который выполняет основные функции, к примеру, включение, выключение, запуск ОС, запуск программы, и т.д, не раскрывая клиенту деталей выполнения этой операции. Пример использования шаблона в главе 7.

Литература, ссылки

1. https://ru.wikipedia.org/wiki/%D8%E0%E1%EB%EE%ED_%EF%F0%EE%E5%EA%F2%E8%F0%EE%E2%E0%ED%E8%FF
2. http://citforum.ru/SE/project/pattern/
3. Э.Фримен, К.Сьерра, Б.Бейтс Паттерны проектирования. - СПб.: Питер, 2011.
4. Классическая книга “банды четырех”. Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. (разные издания, последнее - 2015).