Шаблон проектирования

**Шаблон проектирования** — это именованное эффективное решение задачи проектирования программного обеспечения.

Шаблоны проектирования делятся на 3 вида Creating Создающие шаблоны, Structure структурные шаблоны, Behavioral Поведенческие шаблоны.

**Creating** - создающие шаблоны нацелены на гибкое создание объектов

**Structure** - структурные шаблоны нацелены на эффективное построение взаимосвязей между объектами

**Behavioral** - поведенческие шаблоны нацелены на эффективное взаимодействие между объектами.

Creating Создающие шаблоны:

1. **Factory (фабрика)** - применяется тогда, когда есть суперкласс с несколькими подклассами. На основе ввода нужно вернуть один из подкласса.

К имеющемся суперклассу и его подклассам создается интерфейс фабрики и его реализации, количество соответствует подклассам суперкласса, каждый из них имеет метод который создает соответствующий ему подкласс и возвращает его. В итоге на каждый подкласс можно создать класс, который будет возвращать его новый экземпляр.

1. **Abstract Factory (абстрактная фабрика)** - суперфабрика используется для получения фабрики. После – оная применяется для организации объектов.

Создаются интерфейс фабрики от которой пойдет будущая фабрика и интерфейсы, для будущих объектов, которые они будут возвращать. Далее на основе данных интерфейсов создаются классы фабрик с соответствующими классами для генерации данными фабриками. Теперь создавая нужный экземпляр фабрики, можно возвращать нужные объекты.

1. **Singleton (одиночка)** - отвечает за ограничение создания одного экземпляра класса. Обеспечивает доступ к единственному объекту. Объект на основе данного класса может быть только один.

Класс имеет приватный конструктор благодаря чему нельзя создать объект этого класса за его приделами. Он имеет поля объект данного класса. И имеет публичный статичный синхронный метод, который возвращает это поля, если поле пустое, то он записывает в него новый экземпляр этого класса.

1. **Builder (строитель)** - применяется для создания сложного объекта с использованием простых объектов. Он со временем отвечает за появление большего компонента от малого и простого.

Для начала создадим простой объект, который будет иметь поля который должны меняться у разных объектов. Создадим абстрактный класс строителя с полем, который хранит простой класс, методам который создает объект на основе простого класса и записывает в поле, а также ряд абстрактных методов, который будут влиять на поля простого класса и метод, который будет возвращать поле класса. Теперь создадим классы наследники абстрактного класса, который будут записывать новые данные в поля с помощью реализации абстрактных методов. Теперь создадим класс, который будет иметь поле абстрактного класса и метод, который сможет принять и записать в поле экземпляр объекта, метод который будет использовать все методы объекта в поля и возвращать простой класс который принимает от метода который тоже его возвращает. Теперь можно создавать разные экземпляры простого класса в зависимость от строителя, который получает вышеописанный класс.

1. **Prototype (прототип)** - шаблон, который создает дублированный объект с большей производительностью. Вместо нового создается возвращаемый клон существующего элемента.

Создадим интерфейс с методом для копирования который возвращает объект. Создадим класс, который надо копировать, который реализует вышеописанный интерфейс. У данного класса будет ряд полей, конструктор для них, методы для того, чтоб принимать данный в поля и для выдачи данных, хранящихся в полях, метод, который мы получили от интерфейса для копирования будет создавать новый такой же объект, передавать ему поля и возвращать этот объект. И создадим класс фабрики с полем вышеописанного класса, конструктором для этого поля, методом для принятья объекта и записи в поле и методом, который возвращает клон поля, полученный при использовании методе копирования поля.

Structure структурные шаблоны:

1. **Adapter (адаптер)** - конвертер между несовместимыми объектами. Через него можно объединить интерфейсы, которые изначально не соединяются.

Если мы имеем интерфейс и класс с методами которые нужно объединить, мы создадим класс адаптер, который будет реализовывать этот интерфейс и наследовать класс, и в методе реализуемый от интерфейса, мы будем запускать метод, принятый от класса. Таким образом клиент создает участок память от интерфейса и записывает туда новый экземпляр адаптера, теперь вызывая методы интерфейса мы вызываем методы класса.

1. **Composite (компоновщик)** - отвечающий за использование одного класса для представление древовидной структуры.

Создадим интерфейс с методами и ряд классов которые будут его реализовывать. Далее создадим класс, который будет иметь поле коллекцию хранящая экземпляры интерфейса, создадим методы для удаления и добавления объектов в поле коллекцию и методы, которые будут соответствовать методам интерфейса, они будут перебирать всю коллекцию и запускать у каждого объекта соответствующий метод. Теперь используя один класс, мы можем работать с списком классов.

1. **Proxy (прокси)** - дает функциональность другого класса. Используется для контроля доступа к функциональности.

Создадим интерфейс с методами. Создадим класс, который его реализует и описывает методы. Теперь опишем еще класс, который реализует интерфейс, он имеет все поля который принимает конструктор прошлого класса, также поле, которое может хранить предыдущий класс. Конструктор этого класса принимает те-же поля что и предыдущий класс, а в каждом методе, принятом от интерфейса, проверяется есть пустое ли поле, хранящее класс, если да, то создается новый экземпляр и ему передаются все поля, далее запускается соответствующий метод класса поля. Также методы не реализуемы от интерфейса могут быть использованный в нужном нам методе. Теперь класса не будет создан, пока не будет использован хоть один метод, и контроль методом не от интерфейса происходит иначе.

1. **Flyweight (приспособленец)** - вместо того, чтобы использовать большое количество схожих между собой объектов, они применяются повторно.

Создадим интерфейс и его реализации. Создадим класс с приветным статическим финальным поле коллекцией для хранения по типу ключ-значение в виде хеш-таблиц и ключ – строка, значение – экземпляр интерфейса. Создадим метод, который принимает строку. В методе создается хранилище экземпляра интерфейса и туда записывается результат запроса к полю класса по строке, полученной методом. Проверяем есть ли что-то в хранилище, если нет, то следует конструкция, которая проверяет подготовленный список, срок который соответствует каждой реализации интерфейса, когда нахорится соответствие в хранилище записывается соответствующий класс и он в мести с ключом сохраняется в коллекцию, если нет соответствующей строки, то вызывается ошибка, далее объект в хранилище передается клиенту. Таким образом мы просто пери используем объект, который запрашивает клиент и те объекты которые не запросят, не будут созданы.

1. **Facade (фасад)** - обеспечивает элементарный интерфейс для клиентов. Юзеры будут применять его для контактирования с системой.

Фасад, это класс, который хранит ряд классов системы в виде поле и имеет методы для запуска их методов полей и методы который могут выдавать информацию о классах, которые он хранит. Это простой интерфейс, который ограничивает взаимодействие разработчика клиента с внутренним кодом системы.

1. **Bridge (мост)** - который делает определенные классы независимыми от тех, что используются при реализации интерфейсов. Отделяет абстракцию, от реализации что позволяет независимо их изменять.

Создадим интерфейс с методами и классы, реализующие его и методы. Создадим абстрактный класс, который имеет поле интерфейса, и конструктор для него, а также абстрактные методы, которые соответствуют методам интерфейса. Создадим классы наследника абстрактного класса конструктор которого будет принимать экземпляр интерфейса и передавать его суперклассу, а перенятые методы от абстрактного класса будут запускать соответствующие методы интерфейса. Таким образом реализация интерфейса будет отделена от класса.

1. **Decorator (декоратор)** - способ обеспечения новых функциональных возможностей существующего компонента кода. Его структура в ходе операций не будет привязываться.

Создадим интерфейс и классы его реализующий. Теперь создадим класс декоратора, который реализацией интерфейса, будет иметь поле интерфейса и конструктор для него. Также он будет иметь методы от интерфейса, в который он вызывать соответствующие методы у поля. Далее создадим классы, которые бут наследовать класс декоратор. Она будут иметь конструктор, который будет принимать экземпляр интерфейса и передавать его суперклассу, а также методы который будут дополнять наследуемый метод. Благодаря изначальные реализации интерфейса будут дополненный, но их структура не изменится.

Behavioral Поведенческие шаблоны:

1. **Template Method (шаблонный метод)** - определяет основу алгоритма. Дает шанс наследникам переопределять часть алгоритмических шагов. Структура оных не будет корректироваться.

Создадим абстрактный класс с одним абстрактным методом и вторым методом внутри которого будет запускаться абстрактный метод. Далее создадим ряд классов, который будут наследовать абстрактный класс и по-разному реализовывать его абстрактный метод. Теперь при создании участка память для абстрактного метода в зависимости от того какой наследник будет в нем хранится, будет завесить поведение метода класса. Таким образом наследники переопределяют алгоритмические шаги метода в базовом абстрактном классе.

1. **Mediator (посредник)** - дает возможность предоставления класса посредника. Он будет обрабатывать все коммуникации между разными классами.

Создадим интерфейс для объектов пользователя с методами, где они могут принимать и передавать данный. Создадим интерфейс для классов посредников с методом рассылки данных. Создадим класс реализующий интерфейс пользователей, он будет иметь поле хранящие посредника, конструктор для этого поля, и методы для записи и выдачи этого поля. Метод для отправки сообщения будет запускать идентичный метод у поля и передавать ему принятые данный, реализация метода принятья данных пусть будет вывод его, но она может быть любой. Теперь реализуем класс посредник, он имеет поле коллекцию, хранящую пользователей и метод для добавления их в коллекцию. Метод передачи данных перебирает коллекцию и использует их метод для принятия данных. Таким образом в класс посредник можно добавлять классы пользователей и делать рассылку данных по всем добавленным пользователям.

1. **Chain of Responsibility (цепочка ответственности)** - не дает столкнуться с жесткой зависимостью отправителя запроса от его получателя. Сам запрос будет обрабатываться несколькими объектами.

Создадим класс, который будет хранить публичный статические финальные целочисленный поля приоритетов. Создадим абстрактный класс, который будет иметь целочисленное поле с приоритетами и поле, которое будет хранить этот абстрактный класс для создания очереди, поле с приоритетами задается конструктором. Создадим метод, который может принимать класс в поле абстрактного класса. Создадим абстрактный метод, который принимать запрос. Создадим метод, который будет обрабатывать запрос он будет принимать уровень и приоритет, в нем если принятый приоритет равен или больше полю с приоритетом, то запускается метод принимавший запрос, если поле хранящий класс не равно нулю, то у этого поля запускается метод обрабатывающий запрос и в него передается запрос и приоритет. Далее создадим ряд классов наследников абстрактного класса, их конструкторы будут принимать целочисленный приоритет и передавать его суперклассу и реализовывать метод для обработки сообщений. Теперь создадим классы наследника и зададим им приоритет, далее заполним их поля для очереди. Теперь передавая запрос и приоритет первому в очереди, запрос будет передаваться класса у кого больше или равен приоритет.

1. **Observer (наблюдатель)** - с ним одни компоненты следят и реагируют на события с другими.

Создадим интерфейс, для классов который будут реагировать на события, он будет иметь метод для обработки событий который будет принимать список, данный в виде коллекции. Сделаем класс, который будет реализовывать данный интерфейс, а метод, который обрабатывает события может иметь различный функционал. Сделаем интерфейс для создания и передачи событий, он будет иметь 3 метода, для добавления реагирующий классов, для удаления реагирующий классов и для оповещения реагирующих классов. Создадим класс, который будет реализовывать данный интерфейс. Он будет иметь 2 поля коллекции, поле с событиями и поле с классами которые нужно оповещать. Будет 2 методы который добавляют или удаляют событие из поля коллекции и после вызывают метод для оповещения классов. Еще будет два метода который добавляют и удаляют в поле, классы который нужно оповещать. И метод оповещения, который будет перебирать все классы для оповещения и передавать им все события. Теперь если создать объекты для оповещения и объект, который будет оповещать и добавить объекты для оповещения в него, если добавлять и удалять события, он будет передавать их всем другим объектам.

1. **Strategy (стратегия)** — это state (состояние) которое может корректироваться во время исполнения исходной кодификации.

Создадим интерфейс, который будет реализовывается в состояния и общие методы для них. Создадим классы, реализующие интерфейс, перенятые методы могут выполнять различные действия. Создадим класс, который будет зависим от состояний, он будет иметь поле, которое хранит состояние и метод для записи этого поля, а также идентичные методы интерфейсу состояний, каждый из них будет запускать соответствующий метод у поля. Теперь создавая зависимый класс, мы можем передавать ему различные состояния и использовать их методы.

1. **Command (команда)** - которая объявляет метод для проделывания конкретной манипуляции. Упаковывает запросы для объекта.

Создадим класс внешних команд, каждый метод внешняя это команда. Создадим интерфейс для команд с методом для выполнения команды. Создадим классы для реализации интерфейса они буду соответствовать внешним командам, будут иметь поле класса команд, и конструктор для этого поля, наследованный метод будет выполнять соответствующую команду у поля. Далее создадим класс, который будет работать с командами внутри. Он будет иметь поля интерфейса по количеству команд и конструктор для них. Также он будет иметь методы по количеству команд, каждый метод будет вызывать метод для выполнения команд у соответствующего поля. Таким образом при использовании внутреннего класса и вызове команд каждая команда будет упакована в отдельный класс.

1. **State (состояние)** - поведение элемента корректируется в зависимости от его состояния.

Создадим интерфейс для состояний и его методы. Создадим классы, которые будут реализовывать это интерфейс и его методы. Создадим класс, который будет менять состояния у него будет поле интерфейса состояний и метод для его заполнения, также метод для изменения состояния по списку если состояние n то поменять на n1 и все методы интерфейса в которых у поля будет вызываться соответствующий метод. Теперь создав класс, пеняющий состояния и задав начальное состояние, их можно менять и использовать разные методы.

1. **Visitor (посетитель)** - упрощает операции над группированием связанных объектов.

Создадим интерфейс для принимающих объектов с перегруженным методом для обработки получаемой группы объектов, каждый вариант метода может получать один вариант из получаемой группы, количество равно размеру группы. Создадим реализации этого интерфейса, методы могут выполнять что угодно. Создадим интерфейс для группы объекта, с методом принимает принимающий объект. Создадим реализации этого интерфейса, его метод будет вызывать перегруженный метод у полученного объекта и передавать самого себя. Создадим главенствующий класс, который тоже реализует интерфейс для группы, он имеет поле массив, хранящий группу объектов, в конструкторе в массив добавляются все объекты группы. А принятый от интерфейса метод перебирает поле массив и у каждого элемента запускает метод и передавать в него полученный принимающий объект. Теперь если у главенствующего объекта запустить его единственный метод и передать ему принимающий объект, то вся группа объектов передастся в перегруженный метод получаемого объекта.

1. **Interpreter (интерпретатор)** - контролирует грамматику простого языка в зоне обнаруженной проблемы.

Создадим интерфейс для классов проверки, у него будет метод проверки, который принимает (для примера строковые) данные и возвращает булево значение. Создадим реализацию для этих интерфейсов. Их может быть любое количество, и их поведение может быть дорогое суть в том, чтоб они могли сравнивать и проверять данный. Для примера создадим 1 класс с реализацией интерфейса с булевым сравнение and которые будут иметь 2 поля интерфейса и конструктор, который будет их задавать, перенятый от интерфейса метод, будет сравнивать и возвращать вызов у метода полей и передачу им принятых данных. Также у интерфейса будет еще одна реализация для хранения сверочных данных, которая будет иметь поле с строковыми данными и конструктор для него, в методе для проверки будет условие если в полученной строке есть поле, то вернуть true иначе false. Таком образом мы мажем создать два объекта для проверки и передать им строки. Создать объект для сравнения и передать ему объекты для хранения. При использовании метода проверки и передачи строки, строка будет сравнивается с хранящимися данными в хранилище и так-как сравнения типа and, то если в принятой строке есть часть и первого и второго хранилища, то вернется true. Подобными образом может быть организованна любая грамматическая проверка.

1. **Iterator (итератор)** - последовательно предоставляет доступ к компонентам элемента коллекции. Система не знает его ключевого представления.

Создадим интерфейс итератора с методами проверки есть ли следующий элемент, возвращающий болевое значение методом, который возвращает булево значение. Создадим интерфейс коллекции, который будет иметь метод, который будет возвращать итератор. Создадим класс коллекцию, которая может иметь поля данных и массивы полей данных. Для примера возьмём одно строковое приватное поле и один строковый приватный поле массив, у них будет конструктор, и метод для возвращения и принятия этих полей. Также у класса будет метод, который возвращает вложенный класс и сам вложенный класс, который будет реализовывать итератор. Вложенный класс будет иметь целочисленное поле, метод проверки следующего значения будет сравнивать поле с длинной массива внешнего класса, а метод возвращения элемента будет возвращать элемент массива используя для индекса поле вложенного класса и будет прибавлять к полю единицу. Теперь мы можем создать нашу коллекцию передать ей строку и массив строк, получить итератор на массив строк, поле получить методом получения единичного поля, а массив получить по элементной перибирая циклом используя метод проверки на следующее значение. Так мы можем создавать свои коллекции и итераторы.

Описание с примерами:

Creating Создающие шаблоны

**Factory фабрика**

Интерфейс, благодаря которому создаются объекты. Используется для делегирования создания экземпляра другому классу. Фабрика использует схожие классы и очень удобен, когда заранее не известно какие классы нужно будет создавать.

**Пример:**

У нас имеется Интерфейс Coffee и несколько наследников Americano, Cappuccino, Espresso .... Мы создаем интерфейс CoffeeFactory который имеет метод createCoffee (Возвращающий объект Coffee). И мы наследуем от интерфейса классы AmericanoFactory, CappuccinoFactory, EspressoFactory ... которые в методе createCoffee уже возвращают объекты (Americano, Cappuccino, Espresso ....

**Abstract factory абстрактная фабрика**

Отличии абстрактной фабрики в том, что фабрика производит уникальный продукт, абстрактная фабрика производит несколько видов похожих продуктов.

**Пример:**

У нас есть 2 проекта (создание сайта и создание банковской системы). Есть три интерфейса Developer, Tester, ProjectManager и интерфейс ProjectTeamFactory который возвращает эти интерфейсы. Есть три класса для банковских проектов JavaDeveloper реализует Developer, QATester реализует Tester, BankingPM реализует ProjectManager и есть класс BankingTeamFactory реализует ProjectTeamFactory и аналогичный набор классов для написания сайтов. Таким образом с помощь интерфейсов в примере мы можем реализовывать разные команды для разный типов проектов и с помощью ProjectTeamFactory создавать объекты команды для разных сфер деятельности.

**Singleton синглтон**

Шаблон гарантирует что у класса будет только один экземпляр и к нему будет предоставлена глобальная точка доступа.

**Пример:**

Допустим мы создаем логгер, который должен быть в системе в единственном экземпляре. Мы создаем класс. У него есть статичное приватное поле, которое может хранить его экземпляр, и статичное приветное поле, в котором хранятся сами логе. Создаем статичный синхронный (чтоб 2 потока не могли обратится одновременно) метод который возвращает поле с экземпляр этого класса, если поле пустое, то в него записывается новый экземпляр. Также объявляем пустой приватный конструктор, чтоб экземпляр класса нельзя было создать из вне. И 2 метода, первый, который добавляет в поле с логами новый текст с новой строка и метод, который выводит поле с логами. Все методы в данном примере публичны.

**Builder строитель**

Позволяет отделить конструирование объекта от представления, в итоге можно получить разные представления. Создаем разные объекты с одним набором данных.

**Пример:**

Допустим мы создаем различные вебсайты на основе разных cms. Мы создаем публичный класс Website с тремя приватными полями (название строка, cms перечисления, цена число), есть методы set этих полей и метод toString. Создаем абстрактный класс WebsiteBuilder, у него есть поле Website, метод createWebsite который записывает в поле новый объект Website, три абстрактный метода buildName, buildCms, buildPrice и метод getWebsite который возвращает поле Website. Создаем 2 класса которые представляют конкретные разновидности сайта VisitCardWebsiteBuilder и EnterpriceWebsiteBuilder которые наследуют WebsiteBuilder, внутри методов build поля Website заполняются при помощи методов set. Теперь создается метод Derector который решает какой сайт будет создан. У него есть поле WebsiteBuilder. Метод set для этого поля и метод buildWebsite который использует метод createWebsite этого поля и все методы build, создает объект типа Website сохраняя в него поле данного класса WebsiteBuilder с использованием его метода getWebsite, поле возвращаем Website. Теперь пользователь может создать объект Derector и в зависимости от того какую разновидность вебсайта будет в него передана с помощь метода set и зависит какой экземпляр объекта Website он создаст при использование метода buildWebsite.

**Prototype прототип**

Позволяет определить вид создаваемых объектов с помощью экземпляра — прототип и создавать новые объекты копируя этот прототип.

**Пример:**

У нас есть проект, который мы представим в виде класса Project. У него имеются поля id, projectName, codeSource которые принимают данные на этапе конструирования класса, а также они имеют Getter и Setter. Нам нужна возможность регулярно копировать этот объект. Для этого мы создаем interface Copyable который имеет метод сору, возвращающий Object. Класс Project наследует interface Copyable и переопределяет метод сору. В методе сору создается новый объект Project используя поля этого объекта и метод сору возвращает его. Теперь мы создадим фабрику ProjectFactory которая имеет поле Project, конструктор который принимает класс Project и записывает его в поле Project и метод cloneProject который использует метод сору объекта хранящегося в поле Project приводит его к типу Project и возвращает его.

Structure структурные шаблоны

**Adapter адаптер**

Обеспечивает работу несовместимого класса и интерфейса.

**Пример:**

Допустим у нас есть база данных, представим ее как интерфейс (Database). У нее есть 4 метода insert, update, select, remove. У нас есть java приложение (JavaAplication) c методами saveObject, updateObject, loadObject, deleteObject. Пусть каждый из данных методов выводит в консоль текст процесса, который он должен выполнять (пример: save Java Object). Теперь создадим адаптер AdapterJavaToDatabase который наследует JavaAplication и имплементирует Database в методах имплементированных Database используем нужные нам методы JavaAplication (пример: в методе remove мы используем deleteObject). Теперь выделяя память под базу данных мы сохраняем туда объект адаптер и

и мы можем использовать функции JavaAplication через интерфейс Database.

**Bridge мост**

Позволяет отделить абстракцию и реализацию таким образом, что можно изменять и то, и другое независимо друг от друга.

**Пример:**

Создадим интерфейс Developer с методом writeCode. Создаем два класса СppDeveloper и JavaDeveloper которые реализует интерфейс Developer. Теперь описываем абстрактный класс Program который имеет поле Developer, конструктор, который принимает поле Developer и записывает его, и абстрактный метод developProgram. Теперь создадим 2 класса BankSystem и StockExchange которые наследуют класс Program. Оба класса имеют конструктор принимающие экземпляр Developer и сохраняющие в суперкласс, в методе developProgram они используют метод writeCode поля Developer. Теперь мы можем создать массив Program в которые мы можем записывать класса BankSystem и StockExchange которые могут иметь поля СppDeveloper и JavaDeveloper и у всего массива можно запускать их общий метод developProgram.

**Composite компоновщик**

Объединяет объекты в структуру по типу дерева, позволяет клиентам единообразно трактовать отдельные и составные объекты.

**Пример:**

Создадим интерфейс Developer с методом writeCode. Создаем два класса СppDeveloper и JavaDeveloper которые реализует интерфейс Developer. Теперь создадим класс Team, который объединит объекты производные от Developer. Он имеет приватное поле List<Developer>.

Методы, которые добавляют и удаляют (addDeveloper, removeDeveloper) экземпляры Developer в поле List<Developer> и метод createProject который в цикле прогоняет все экземпляры List<Developer> и использует их метод writeCode. Теперь мы можем создавать разных по параметрам разработчиков и объединять их в команду.

**Decorator Декоратор**

Позволяет динамически добавлять новые обязанности объекту.

**Пример:**

Создадим интерфейс Developer с методом makeJob который возвращает строку. Создадим класс JavaDeveloper который реализует интерфейс Developer, а метод makeJob возвращает текст “Write java code…”. Но допустим на м нужен не JavaDeveloper, а SenierJavaDeveloper или JavaTeamLead. Создадим класс DeveloperDecorator который будет иметь поле Developer конструктор который принимает экземпляр Developer и записывает его в поле и переопределенный метод makeJob который возвращает текст, который вернет метод makeJob поля Developer. Теперь создадим SenierJavaDeveloper который наследует DeveloperDecorator, имеет конструктор, который принимает экземпляр Developer и передает его суперклассу, и новый метод makeCodeReview который возвращает текст “Make code review…”, а метод makeJob переопределяется и теперь возвращает текст, который вернут метод makeJob суперкласса и текст, который вернул makeCodeReview. Создадим класс JavaTeamLead который наследует DeveloperDecorator, имеет конструктор, который принимает экземпляр Developer и передает его суперклассу, и новый метод sendWeekReport который возвращает текст “ Send week report to customer…”, а метод makeJob переопределяется и теперь возвращает текст, который вернут метод makeJob суперкласса и текст, который вернул sendWeekReport. Теперь при создании нового экземпляра JavaTeamLead мы передадим ему новый экземпляр SeniorJavaDeveloper, а ему передадим новый экземпляр JavaDeveloper и при выполнении метода makeJob и вывода полученного текста на экран мы видим все дополнения.

Write java code...Make code review...Send week report to customer...

**Facade фасад**

Объединяет несколько интерфейсов в один.

**Пример:**

Создадим класс Job с методом doJob. Создадим класс BugTracker c булевым поле activeSprint, методом isActiveSprint который возвращает поле activeSprint, методом startSprint который записывает в поле activeSprint true и методом finishSprint который записывает в поле activeSprint false. Теперь создадим класс Developer с методом doJobBeforeDeadline принимающий экземпляр BugTracker и имеющий условие если метод BugTracker isActiveSprint выдает true, то выводится текст “ Developer is solving problems...” иначе выводит текст “ Developer is reading Habrahabr...”. Теперь вместо того, чтоб отдельно использовать данные классы клиентом, мы объединим их в фасад. Создадим класс WorkFlow который имеет поля Developer хранящий новый экземпляр, Job хранящий новый экземпляр и BugTracker хранящий новый экземпляр, а также метод solveProblems который использует метод doJob поля Job, метод startSprint поля BugTracker и doJobBeforeDeadline поля Developer и передает ему поле BugTracker.

**Flyweight Приспособленец**

Поддерживает множество мелких объектов.

**Пример:**

Создадим интерфейс Developer с методом writeCode. Создаем два класса СppDeveloper и JavaDeveloper которые реализует интерфейс Developer. Теперь создадим класс DeveloperFactory, с приватным статичным финальным полем коллекцией Map<String, Developer> и методом getDeveloperBySpecialty который принимает строку, в методе создается экземпляр Developer который хранит в себе класс по запросу из поля Map передавая ему полученную методом строку. Далее в методе следует условие, если экземпляр Developer равен null, то начинается оператор switch в котором первое условие гласит - если принятый текст методом равен “java”, то экземпляр Developer принимает новый объект JavaDeveloper, второе условие гласит - если принятый текст методом равен “cpp”, то экземпляр Developer принимает новый объект СppDeveloper. Далее функция switch заканчивается, а в поле Map добавляется экземпляр Developer с ключом (текст принятого методом). Далее условие заканчивается и метод возвращает экземпляр Developer. Теперь мы можем по запросу получать ссылку на объекты СppDeveloper и JavaDeveloper множество раз, а если их не существует в память, то будут объявляться новые.

**Proxy Проси**

Используется для замещения другого объекта и контроля доступа к нему.

**Пример:**

Создадим интерфейс Project с методом run. Теперь сделаем его реализацию RealProject с строковым полем url, конструктором принимающий ссылку и записывающую ее в url и использующий метод load, метод load который выводит текст “"Loading project from " + url + "..."”, и метод run который выводит текст “"Running project " + url + "..."”. Создадим класс ProxyProject который реализует Project. Он имеет строковое поле url и поле RealProject, конструктор, который принимает текст и записывает в поле url и метод run с условием если поле RealProject равно null, то записать в данное поле новый экземпляр данного класса и передать ему url. После условия у поля RealProject запускается метод run. Таким образом используя ProxyProject как промежуточный класс RealProject будет создаваться и использовать метод load в момент использования метода run у объекта ProxyProject.

Behavioral Поведенческие шаблоны

**Chain of Responsibility Цепочка ответственности**

Связывания объектов получателя в цепочку и передача по ней.

**Пример:**

Создадим абстрактный класс Notifier с целочисленным приватный полем priority и приватным полем Notifier. Он имеет конструктор, принимающий целое число и записывающее его в priority. Также у него есть seter поля Notifier, метод notifyManager принимающий текст massage и целочисленное число level. Данный метод начинается с условия, если level больше или равен priority, то использовать метод write данного класса и передать ему massage. Далее следует второе условие, если поле Notifier данного класса не равно null, то использовать метод notifyManager поля Notifier из данного класса и передать ему level и massage. И у данного класса есть абстрактный метод write который принимает строку massage. Далее создаем класс, хранящий приоритеты Priority который имеет три публичных статичный целочисленных поля: ROUTINE равное 1, IMPORTANT равное 2 и ASAP равное 3. Далее опишем три класса наследника класса Notifier (у всех них есть конструктор принимающий целочисленную переменную и передающую ее супер классу):

1. SimpleReportNotifier у которого метод write выводит текст “"Notifier using simple report: "+ massage”
2. EmailNotifier у которого метод write выводит текст “"Sending email: " + massage”
3. SMSNotifier у которого метод write выводит текст “"Sending SMS to manager: " + massage”

Далее мы опишем класс-клиент, который создает 3 объекта:

Notifier reportNotifier = new SimpleReportNotifier(Priority.ROUTINE);  
Notifier emailNotifier = new EmailNotifier(Priority.IMPORTANT);  
Notifier smsNotifier = new SMSNotifier(Priority.ASAP);

Теперь у объекта reportNotifier используем метод seter и передаем ему emailNotifier. Далее вызываем метод seter у emailNotifier и передаем ему smsNotifier. Теперь при вызове метода notifyManager передавая ему текст в зависимости от приоритета будут использоваться классы, выводящие сообщение. Если приоритет 1, то выводит только сообщение от SimpleReportNotifier, если приоритет пр2, то выводит сообщение от SimpleReportNotifier и от

EmailNotifier, а если приоритет 3, то выводит сообщение от всех трех классов.

**Сommand команда**

Инкапсулирование запросов для объектов.

**Пример:**

Создадим класс Database c методом insert который выводит текст "Inserting record...", методом update который выводит текст "Updating record...", методом select который выводит текст "Reading record…", и методом delete который выводит текст "Deleting record...". Теперь создадим интерфейс Command с методом execute. Теперь создадим 4 класса каждый из которых имеет поле Database, и конструктор, который принимает экземпляр Database и записывает в конкретное поле.

Классы:

1. InsertCommand его метод execute вызывает метод insert у поля Database.
2. UpdateCommand его метод execute вызывает метод update у поля Database.
3. SelectCommand его метод execute вызывает метод select у поля Database.
4. DeleteCommand его метод execute вызывает метод delete у поля Database.

Далее создадим класс Developer который имеет 4 поля: Command insert, Command update, Command select, Command delete. У него есть конструктор, который принимает 4 экземпляра класса Command и записывает их в поля. Также он имеет 4 метода:

1. insertRecord который вызывает метод execute у поля insert.
2. updateRecord который вызывает метод execute у поля update.
3. selectRecord который вызывает метод execute у поля select.
4. deleteRecord который вызывает метод execute у поля delete.

Теперь создадим класс клиент. В нем объявляется объект Database. Далее объявляется объект Developer который принимает 4 новых объекта каждый из которых принимает объявленный объект Database: InsertCommand, UpdateCommand, SelectCommand, DeleteCommand. Теперь мы можем вызвать методы insertRecord, updateRecord, selectRecord, deleteRecord.

**Interpreter Интерпретатор**

Определяет представления грамматики объекта.

**Пример:**

Создадим интерфейс Expression c методом interpret который возвращает булево значение, а принимает строку context. Создадим 2 класса реализующий Expression (AndExpression и OrExpression) который имеют 2 поля Expression (expression1, expression2), также они имеют конструктор, который принимает 2 экземпляра Expression и записывают их в поля и 2 метода interpret.

1. AndExpression метод возвращает булевой результат

expression1.interpret(context) && expression2.interpret(context)

1. OrExpression метод возвращает булевой результат

expression1.interpret(context) || expression2.interpret(context)

Также мы создадим класс TerminalExpression реализующий Expression который имеет строковое поле data, и конструктор, который принимает строку и записывает в data. А метод interpret начинается с условия если context.contains(data) вернуть true, иначе вернуть false. Теперь создадим класс, который будет запускать наш пример. Для начала за приделами main создадим 2 статичных метода которые возвращают Expression getJavaExpression и getJavaEEExpresion:

1. Метод getJavaExpression создает 2 объекта TerminalExpression: объект java который принимает текст “Java” и объект javaCore который принимает текст “Java Core”. Метод возвращает новый объект OrExpression принимая объекты java и javaCore.
2. Метод getJavaEEExpresion создает 2 объекта TerminalExpression: объект java который принимает текст “Java” и объект spring который принимает текст “ Spring ”. Метод возвращает новый объект AndExpression принимая объекты java и spring.

Далее в main создаётся 2 экземпляра Expression:

1. isJava в котором хранится то что вернет метод getJavaExpression.
2. isJavaEEDeveloper в котором хранится то что вернет метод getJavaEEExpresion.

Теперь выведем "Does developer knows Java Core: " + isJava.interpret("Java Core") и "Does developer knows Java EE: " + isJavaEEDeveloper.interpret("Java Spring") и в результате получим Does developer knows Java Core: true и Does developer knows Java EE: true.

**Iterator**

Получение последовательного доступа ко всем элемента составного объекта.

**Пример:**

Создадим интерфейс Iterator c публичным методом hasNext который возвращает

boolean и публичным методом next который возвращает object. Далее создадим интерфейс Collection который имеет метод getIterator возвращающий Iterator. Создадим класс JavaDeveloper реализующий Collection. Он имеет строковое поле name и строковое поле массив skills, также есть конструктор, который принимает строковое поле и строковый массив и записывает их в поля. У обоих полей есть geters и seters. Метод getIterator возвращает новый экземпляр SkillIterator. Далее создаем вложенный класс SkillIterator реализующий iterator. Он имеет поле index. Его метод hasNext начинается с условия если index > skills.lenght вернуть true. Далее следует вернуть false. Метод next возвращает skills[index++]. Теперь опишем класс, который запускает наш пример. В классе создается строковый массив skills, который заполняется текстом различных скилов. Создается новый объект JavaDeveloper (javaDeveloper) который принимает имя и массив skills. Далее создается экземпляр Iterator который хранит объект, который вернул метод getIterator объекта javaDeveloper. Выводим текст с новой строки "Developer: " + javaDeveloper.getName().

Выводим текст с новой строки "Skills: ". Теперь следует цикл while с условием iterator.hasNext(), в теле выводится текст iterator.next().toString() + " ". Нам вывело имя и все скилы в классе.

**Mediator Посредник**

Инкапсуляция способа взаимодействия множества объектов.

**Пример:**

Создадим интерфейс Chat с публичным методом, который принимает строку message и User user. Создадим интерфейс User с публичным методом sendMessage который принимает строку message и публичным методом getMessage который принимает строку massage. Создадим класс Admin и SimpleUser которые реализуют интерфейс User. Они имеют поле Chat chat и строковое поле name c конструктором для этих полей, а также geters и seters для них. Метод sendMessage вызывает метод sendMessage у поля chat передает ему поле message и этот класс. Метод getMessage выводи текст this.name + " receiving message: " + message + ".". Далее создадим класс SimpleTextChat который реализует интерфейс Chat. Он имеет поле User admin и поле коллекцию List<User> users = new ArrayList<>();. У поля admin есть seter. Теперь идет метод addUserToChat который принимает User user и записывает user в поле коллекцию users. Метод sendMessage начинается с цикла for который перебирает все элементы коллекции users и хранит их в User u, далее следует условие если u != user(который принимает данный метод), то у u используется метод getMessage которому передается message(тоже принимает данный метод), после условия и цикла у поля admin вызывается метод getMassage и ему передается message. Теперь опишем класс, который будет запускать программу. В начале создадим новый экземпляр объекта SimpleTextChat chat. Далее создадим новый экземпляр объектов: User admin = new Admin и передадим ему chat и текст "Admin", User user1 = new SimpleUser и передадим ему chat и текст " User 1", User user2 = new SimpleUser и передадим ему chat и текст " User 2". Далее у объекта chat используем seter для admin b передаем ему admin и дважды метод addUserToChat передавая user1 и user2. Используем метод sendMassage у объекта user1 и передаем текст "Hello, I am user one", запускаем и видим, что текс был передан user2 и admin. Теперь используем метод sendMassage у объекта admin и передаем текст "Roger that. i am admin!!!", и при закуске он мы видим что он был передан как user1 и user2 так и admin.

**Memento хранитель**

Сохраняет внутреннее состояние класса за его приделы.

**Пример:**

Создадим класс Project который имеет строковое поле version и поле Date date. Далее идет конструктор, который принимает строку и записывает в поле, а в поле date записывает новый экземпляр Date. Теперь идет метод setVersionAndDate который принимает строку version. В методе строка version записывается в поле version, а в поле date записывается новый экземпляр Date. Следующий метод save который возвращает новый экземпляр будущего класса Save и передает ему поле version. Опишем метод load который принимает экземпляр будущего класса Save save. Внутри метода в поле version записывается результат полученный от save.getVersion, а в поле date записывается результат полученный от save.getDate. И в конце класса есть метод toString который выводит наши поля и их названия. Теперь опишем класс Save. Он имеет приватное финальное строковое поле version и приветное финальное поле Data date. Далее идет конструктор, который принимает строку и записывает ее в поле version, а в поле date записывает новый экземпляр класса Date. Далее идет geter для обоих полей. Опишем класс GithubRepo. Он имеет поле save и geter и seter для него. И на конец опишем класс, который будет все это запускать. В начале создадим новый экземпляр Project и запишем в Project project и новый экземпляр GithubRepo который запишем в GithubRepo github. Выведем текст "Creating new project. Version 1.0", а в объекте project используем метод setVersionAndDate и предадим ему текст "Version 1.0". Выведем project. Теперь выведем текст "Saving current version to github..." и используем метод setSave у объекта github, в который передадим полученное при использовании метода save у объекта project. Выведем текст "Updating project to version 1.1" и текст "Writing poor code...". Далее используем Thread.sleep(5000) для задержки в 5 секунд. У объекта project используем метод setVersionAndDate и предадим ему текст "Version 1.1". Выведем project. Выведем текст "Something went wrong..." и текст "Rolling back to version 1.0". Используем метод load у объекта project и передаем в него результат при использовании метода getSave у объекта github. Выведем текст "Project after rollback" и выведем project.

В результате увидим:

Creating new project. Version 1.0

Project

version: Version 1.0

date: Thu Mar 02 16:38:54 MSK 2023

Saving current version to github...

Updating project to version 1.1

Writing poor code...

Project

version: Version 1.1

date: Thu Mar 02 16:38:59 MSK 2023

Something went wrong...

Rolling back to version 1.0

Project after rollback

Project

version: Version 1.0

date: Thu Mar 02 16:38:54 MSK 2023

**Observer Наблюдатель**

Определяет зависимость один ко многим между объектами.

**Пример:**

Создадим интерфейс Observer с публичным методом handleEvent который принимает коллекцию строк vacancies. Создадим интерфейс Observed с тремя публичными методами, addObserver принимает Observer observer, removeObserver принимает Observer observer и notifyObservers. Теперь опишем класс Subscriber который реализует интерфейс Observer. У него есть строковое поле name, и конструктор, который принимает сроку и записывает в данное поле. Его метод hendleEvent выводим текст:

"Dear " + name + "\n We have some changes in vacancies:\n" + vacancies + "\n==================================================\n"

Далее опишем класс JavaDeveloperJobSite который реализует Observed. У него есть строковое поле коллекция List<String> vacancies = new ArrayList<>(); и поле коллекция List<Observer> subscribers = new ArrayList<>();. Далее идет метод addVacancicy который добавляет принятую строку в поле коллекцию vacancies и вызывает метод notifyObservers данного класса. Метод removeVacancy удаляет принятую строку из поля коллекцию vacancies и вызывает метод notifyObservers данного класса. Метод addObserver добавляет принятый экземпляр observer в поле коллекцию subscribers. Метод removeObserver удаляет принятый экземпляр observer из поля коллекцию subscribers. Метод notifyObservers начинается с цикла for в котором перебирает все элементы поля коллекции subscribers и хранит их в Observer observer. В цикле используется метод handleEvent к объекту observer и туда передается поле коллекция vacancies. Теперь опишем класс, который будет запускать программу, ля начала создадим новый объект JavaDeveloperJobSite и запишем его в JavaDeveloperJobSite jobsite. Теперь используем метод addVacancy объекта jobsite дважды: в первый раз передадим в него строку "First Java Position", во второй раз передадим строку "Second Java Position". Теперь создадим 2 экземпляра класса Subscriber: первый запишем в Observer firstSubscriber и передадим ему строку "Hurry Potter", второй запишем в Observer secondSubscriber и передадим ему строку.

Теперь запустим метод addObserver у объекта jobsite дважды, в первый раз передадим ему firstSubscriber, во второй раз secondSubscriber. Используем метод addVacancy объекта jobsite и передадим ему строку "Third Java Position". Используем метод removeVacancy объекта jobsite и передадим ему строку "First Java Position".

Результат при запуске программы:

Dear Hurry Potter

We have some changes in vacancies:

[First Java Position, Second Java Position, Third Java Position]

==================================================

Dear Albus Dumbledore

We have some changes in vacancies:

[First Java Position, Second Java Position, Third Java Position]

==================================================

Dear Hurry Potter

We have some changes in vacancies:

[Second Java Position, Third Java Position]

==================================================

Dear Albus Dumbledore

We have some changes in vacancies:

[Second Java Position, Third Java Position]

==================================================

**State Состояние**

Управляет поведением объекта в зависимости от его состояния.

**Пример:**

Создадим интерфейс Activity с публичным методом justDoIt(). Далее создадим 4 класса которые реализуют этот интерфейс:

1.Coding, метод justDoIt выводит текст "Writing code...".

2.Reading, метод justDoIt выводит текст "Read book...".

3.Sleeping, метод justDoIt выводит текст "Sleaping...".

4.Training, метод justDoIt выводит текст " Training...".

Далее создадим класс Developer с полем Activity и seter для данного поля. У него есть метод changeActivity являющийся связкой условий. Если поле Activity равно Sleeping, то в него записывается новый экземпляр Training, иначе если поле Activity равно Training, то в него записывается новый экземпляр Coding, иначе если поле Activity равно Coding, то в него записывается новый экземпляр Reading, иначе если поле Activity равно Reading, то в него записывается новый экземпляр Sleeping. И метод justDoIt который запускает метод justDoIt у поля Activity. Теперь опишем main. В нем создается поля Activity activity хранящее новый экземпляр Sleeping и поле Developer developer хранящее новый экземпляр developer. Далее у объекта девелопер используется seter принимающий activity. Начинается цикл for с 10 повторениями в котором у объекта developer используется метод justDoIt, а после у него используется метод changeActivity. При запуске мы постоянно видем смену классов у поля Activity объекта developer.

Sleeping...

Training...

Writing code...

Reading book...

Sleeping...

Training...

Writing code...

Reading book...

Sleeping...

Training...

**Strategy Стратегия**

Определяет семейство классов, инкапсулирует каждое из них и организует их взаимодействие.

**Пример:**

Создадим интерфейс Activity с публичным методом justDoIt(). Далее создадим 4 класса которые реализуют этот интерфейс:

1.Coding, метод justDoIt выводит текст "Writing code...".

2.Reading, метод justDoIt выводит текст "Read book...".

3.Sleeping, метод justDoIt выводит текст "Sleaping...".

4.Training, метод justDoIt выводит текст " Training...".

Далее создадим класс Developer с полем Activity и seter для данного поля. У него есть метод executeActivity который запускает метод justDoIt у поля Activity. Теперь опишем main. Создадим поле Developer developer которое будет новый экземпляр Developer.

1. Используем seter объекта developer и предадим новый экземпляр Sleeping и запустим метод executeActivity у данного объекта.
2. Используем seter объекта developer и предадим новый экземпляр Training и запустим метод executeActivity у данного объекта.
3. Используем seter объекта developer и предадим новый экземпляр Coding и запустим метод executeActivity у данного объекта.
4. Используем seter объекта developer и предадим новый экземпляр Reading и запустим метод executeActivity у данного объекта.
5. Используем seter объекта developer и предадим новый экземпляр Sleeping и запустим метод executeActivity у данного объекта.

При запуске мы увидим тоже что и у примера state, но здесь всем управляет разработчик клиент.

Sleeping...

Training...

Writing code...

Reading book...

Sleeping...

**Template method Шаблонный метод**

Определяет основы класса и создает возможность подклассам переопределить его часть.

**Пример:**

Создадим абстрактный класс WebsiteTemplate с методом showPage который выводит текст “header” использует метод showPageContent и выводит текст “footer” и абстрактным методом showPageContent. Теперь создадим 2 класса которые наследуют WebsiteTemplate.

1. WelcomePage, его метод showPageContent выводит текст “Welcome”.
2. NewsPage, его метод showPageContent выводит текст “News”.

Теперь в main мы можем на основе WebsiteTemplate создать объект WelcomePage или NewsPage и в зависимости от объекта будет зависеть что он будет выводить.

**Visitor Посетитель**

Описывает действия который выполняются с каждым объектом в некоторой структуре, что позволяет определить новую операцию без изменения классов этих объектов.

**Пример:**

Создадим интерфейс ProjectElement с методом beWritten принимающий объект Developer developer. Создадим 3 класса которые реализуют данный интерфейс: ProjectClass, Database, Test во всех них в методе beWritten у принятого класса developer вызывается метод create и передается this. Создадим класс Project который тоже реализует ProjectElement, он имеет поле массив ProjectElement[] projectElement. В конструкторе в поле projectElement добавляется новые экземпляры ProjectClass, Database и Test. В методе beWritten открывается цикл который перебирает все элемента массива projectElement и у каждого запускает метод beWritten и передает ему принятый объект developer. Создадим интерфейс developer c тремя методами create

1. Принимает ProjectClass projectClass.
2. Принимает Database database.
3. Принимает Test test.

Создадим 2 реализации данного интерфейса.

1.JuniorDevelper его методы create:

1.1. create(ProjectClass projectClass) выводит текст "Writing poor class..."

1.2. create(Database database) выводит текст "Drop database..."

1.3. create(Test test) выводит текст "Creating not reliable test..."

2.SeniorDeveloper его методы create:

2.1. create(ProjectClass projectClass) выводит текст "Rewriting class after junior..."

* 1. create(Database database) выводит текст "Fixing database..."
  2. create(Test test) выводит текст "Creating reliable test..."

Теперь опишем main. Создадим объект Project project хранящий project, объект Developer junior хранящий JuniorDevelper и Developer senior хранящий SeniorDeveloper. Далее выведем текст "Junior in Action...". Запустим метод beWritten у объекта project и передадим объект junior. Выведем текст "\n=============================================\n". Выведем текст "Senior in Action...". Запустим метод beWritten у объекта project и передадим объект senior. При запуске получим:

Junior in Action...

Writing poor class...

Drop database...

Creating not reliable test...

=============================================

Senior in Action...

Rewriting class after junior...

Fixing database...

Creating reliable test...

Process finished with exit code 0