**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра «Информационные Системы и Технологий (ИСиТ)»**

отчет

**по лабораторной работе №3**

**по дисциплине «Программирование»**

Тема: «Логическое разделение классов»

|  |  |  |
| --- | --- | --- |
| Студент гр. 0324 |  | Зимацкий С. Н. |
| Преподаватель |  | Глущенко А.Г. |

Санкт-Петербург

2022

**Цель работы.**

Разработать и реализовать набора классов для взаимодействия пользователя с юнитами и базой. Основные требования:

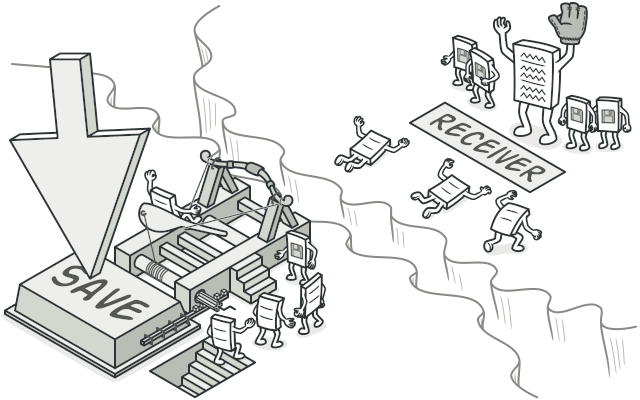
●     Должен быть реализован функционал управления юнитами

●     Должен быть реализован функционал управления базой

**Основные теоретические положения.**

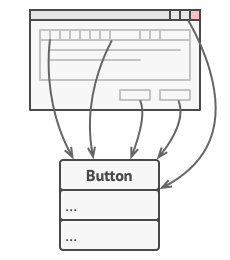
Команда

**Команда** — это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

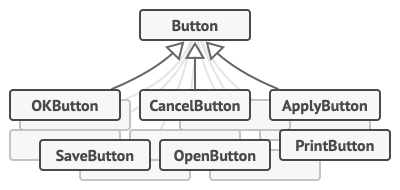


**Проблема**

Представьте, что вы работаете над программой текстового редактора. Дело как раз подошло к разработке панели управления. Вы создали класс красивых Кнопок и хотите использовать его для всех кнопок приложения, начиная от панели управления, заканчивая простыми кнопками в диалогах.



Все эти кнопки, хоть и выглядят схоже, но делают разные вещи. Поэтому возникает вопрос: куда поместить код обработчиков кликов по этим кнопкам? Самым простым решением было бы создать подклассы для каждой кнопки и переопределить в них метод действия под разные задачи.



Но скоро стало понятно, что такой подход никуда не годится. Во-первых, получается очень много подклассов. Во-вторых, код кнопок, относящийся к графическому интерфейсу, начинает зависеть от классов бизнес-логики, которая довольно часто меняется.

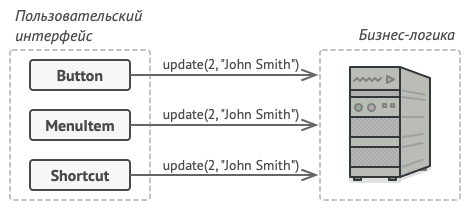


Но самое обидное ещё впереди. Ведь некоторые операции, например, «сохранить», можно вызывать из нескольких мест: нажав кнопку на панели управления, вызвав контекстное меню или просто нажав клавиши Ctrl+S. Когда в программе были только кнопки, код сохранения имелся только в подклассе SaveButton. Но теперь его придётся продублировать ещё в два класса.

**Решение**

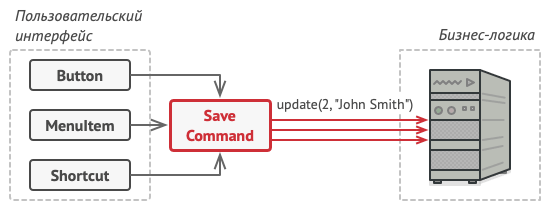
Хорошие программы обычно структурированы в виде слоёв. Самый распространённый пример — слои пользовательского интерфейса и бизнес-логики. Первый всего лишь рисует красивую картинку для пользователя. Но когда нужно сделать что-то важное, интерфейс «просит» слой бизнес-логики заняться этим.

В реальности это выглядит так: один из объектов интерфейса напрямую вызывает метод одного из объектов бизнес-логики, передавая в него какие-то параметры.



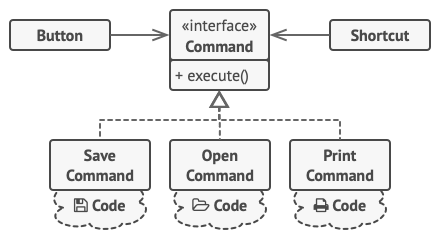
Паттерн Команда предлагает больше не отправлять такие вызовы напрямую. Вместо этого каждый вызов, отличающийся от других, следует завернуть в собственный класс с единственным методом, который и будет осуществлять вызов. Такие объекты называют *командами*.

К объекту интерфейса можно будет привязать объект команды, который знает, кому и в каком виде следует отправлять запросы. Когда объект интерфейса будет готов передать запрос, он вызовет метод команды, а та — позаботится обо всём остальном.



Классы команд можно объединить под общим интерфейсом c единственным методом запуска. После этого одни и те же отправители смогут работать с различными командами, не привязываясь к их классам. Даже больше: команды можно будет взаимозаменять на лету, изменяя итоговое поведение отправителей.

Параметры, с которыми должен быть вызван метод объекта получателя, можно загодя сохранить в полях объекта-команды. Благодаря этому, объекты, отправляющие запросы, могут не беспокоиться о том, чтобы собрать необходимые для получателя данные. Более того, они теперь вообще не знают, кто будет получателем запроса. Вся эта информация скрыта внутри команды.



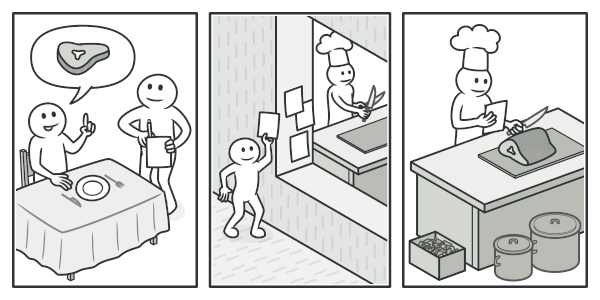
После применения Команды в нашем примере с текстовым редактором вам больше не потребуется создавать уйму подклассов кнопок под разные действия. Будет достаточно единственного класса с полем для хранения объекта команды.

Используя общий интерфейс команд, объекты кнопок будут ссылаться на объекты команд различных типов. При нажатии кнопки будут делегировать работу связанным командам, а команды — перенаправлять вызовы тем или иным объектам бизнес-логики.

Так же можно поступить и с контекстным меню, и с горячими клавишами. Они будут привязаны к тем же объектам команд, что и кнопки, избавляя классы от дублирования.

Таким образом, команды станут гибкой прослойкой между пользовательским интерфейсом и бизнес-логикой. И это лишь малая доля пользы, которую может принести паттерн Команда!

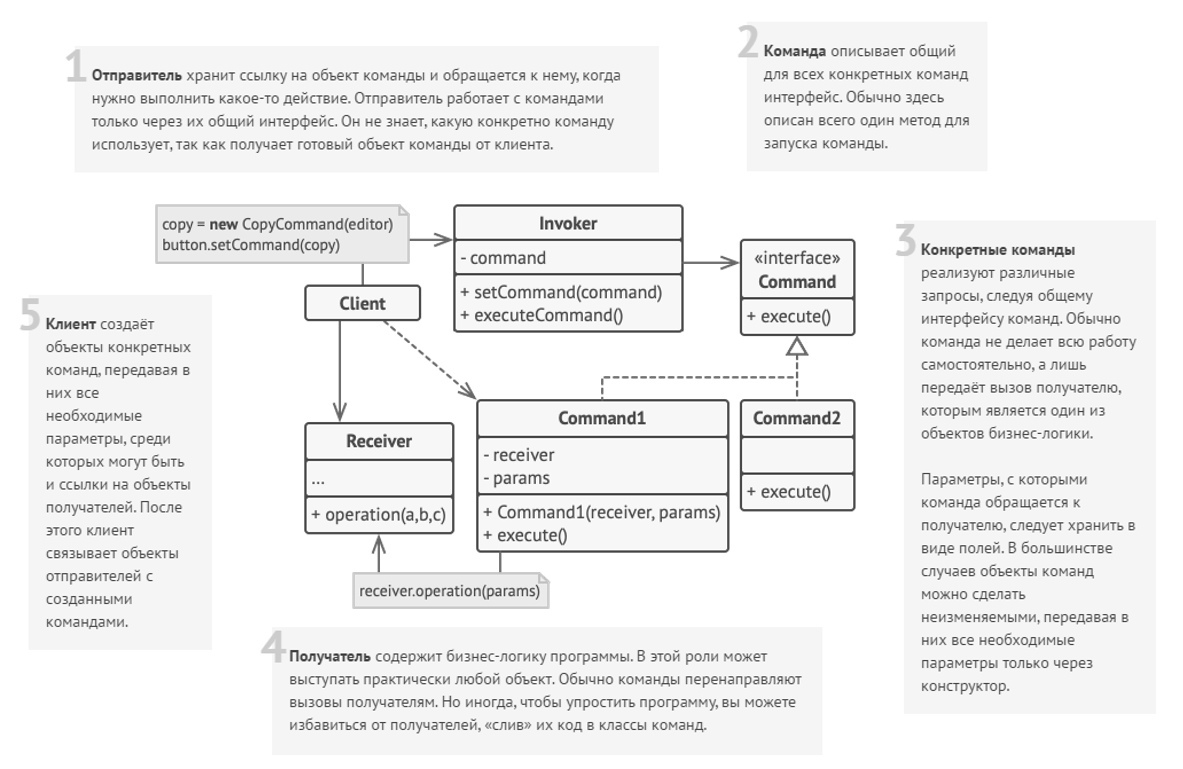
**Аналогия из жизни**



Вы заходите в ресторан и садитесь у окна. К вам подходит вежливый официант и принимает заказ, записывая все пожелания в блокнот. Откланявшись, он уходит на кухню, где вырывает лист из блокнота и клеит на стену. Далее лист оказывается в руках повара, который читает содержание заказа и готовит заказанные блюда.

В этом примере вы являетесь *отправителем*, официант с блокнотом — *командой*, а повар — *получателем*. Как и в паттерне, вы не соприкасаетесь напрямую с поваром. Вместо этого вы отправляете заказ с официантом, который самостоятельно «настраивает» повара на работу. С другой стороны, повар не знает, кто конкретно послал ему заказ. Но это ему безразлично, так как вся необходимая информация есть в листе заказа.

**Структура**



**Применимость**

* Когда вы хотите параметризовать объекты выполняемым действием.

Команда превращает операции в объекты. А объекты можно передавать, хранить и взаимозаменять внутри других объектов.

Скажем, вы разрабатываете библиотеку графического меню и хотите, чтобы пользователи могли использовать меню в разных приложениях, не меняя каждый раз код ваших классов. Применив паттерн, пользователям не придётся изменять классы меню, вместо этого они будут конфигурировать объекты меню различными командами.

* Когда вы хотите ставить операции в очередь, выполнять их по расписанию или передавать по сети.

Как и любые другие объекты, команды можно сериализовать, то есть превратить в строку, чтобы потом сохранить в файл или базу данных. Затем в любой удобный момент её можно достать обратно, снова превратить в объект команды и выполнить. Таким же образом команды можно передавать по сети, логировать или выполнять на удалённом сервере.

* Когда вам нужна операция отмены.

Главная вещь, которая вам нужна, чтобы иметь возможность отмены операций, — это хранение истории. Среди многих способов, которыми можно это сделать, паттерн Команда является, пожалуй, самым популярным.

История команд выглядит как стек, в который попадают все выполненные объекты команд. Каждая команда перед выполнением операции сохраняет текущее состояние объекта, с которым она будет работать. После выполнения операции копия команды попадает в стек истории, все ещё неся в себе сохранённое состояние объекта. Если потребуется отмена, программа возьмёт последнюю команду из истории и возобновит сохранённое в ней состояние.

Этот способ имеет две особенности. Во-первых, точное состояние объектов не так-то просто сохранить, ведь часть его может быть приватным. Но с этим может помочь справиться паттерн Снимок.

Во-вторых, копии состояния могут занимать довольно много оперативной памяти. Поэтому иногда можно прибегнуть к альтернативной реализации, когда вместо восстановления старого состояния команда выполняет обратное действие. Недостаток этого способа в сложности (а иногда и невозможности) реализации обратного действия.

**Шаги реализации**

1. Создайте общий интерфейс команд и определите в нём метод запуска.
2. Один за другим создайте классы конкретных команд. В каждом классе должно быть поле для хранения ссылки на один или несколько объектов-получателей, которым команда будет перенаправлять основную работу.

Кроме этого, команда должна иметь поля для хранения параметров, которые нужны при вызове методов получателя. Значения всех этих полей команда должна получать через конструктор.

И, наконец, реализуйте основной метод команды, вызывая в нём те или иные методы получателя.

1. Добавьте в классы отправителей поля для хранения команд. Обычно объекты-отправители принимают готовые объекты команд извне — через конструктор либо через сеттер поля команды.
2. Измените основной код отправителей так, чтобы они делегировали выполнение действия команде.
3. Порядок инициализации объектов должен выглядеть так:
   * Создаём объекты получателей.
   * Создаём объекты команд, связав их с получателями.
   * Создаём объекты отправителей, связав их с командами.

**Преимущества и недостатки**



**Отношения с другими паттернами**

Цепочка обязанностей, Команда, Посредник и Наблюдатель показывают различные способы работы отправителей запросов с их получателями:

*Цепочка обязанностей* передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.

*Команда* устанавливает косвенную одностороннюю связь от отправителей к получателям.

*Посредник* убирает прямую связь между отправителями и получателями, заставляя их общаться опосредованно, через себя.

*Наблюдатель* передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.

Обработчики в Цепочке обязанностей могут быть выполнены в виде Команд. В этом случае множество разных операций может быть выполнено над одним и тем же контекстом, коим является запрос.

Но есть и другой подход, в котором сам запрос является *Командой*, посланной по цепочке объектов. В этом случае одна и та же операция может быть выполнена над множеством разных контекстов, представленных в виде цепочки.

Команду и Снимок можно использовать сообща для реализации отмены операций. В этом случае объекты команд будут отвечать за выполнение действия над объектом, а снимки будут хранить резервную копию состояния этого объекта, сделанную перед самым запуском команды.

Команда и Стратегия похожи по духу, но отличаются масштабом и применением:

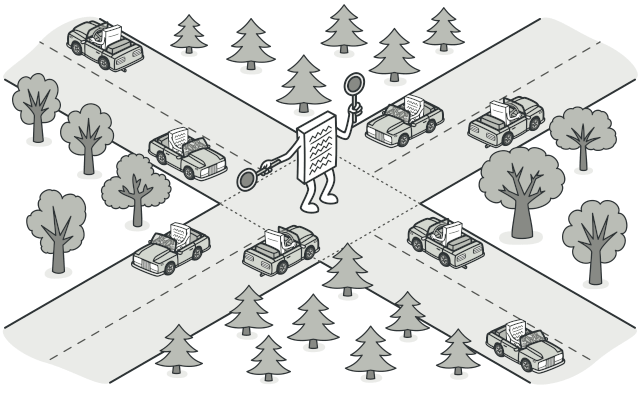
*Команду* используют, чтобы превратить любые разнородные действия в объекты. Параметры операции превращаются в поля объекта. Этот объект теперь можно логировать, хранить в истории для отмены, передавать во внешние сервисы и так далее.

С другой стороны, *Стратегия* описывает разные способы произвести одно и то же действие, позволяя взаимозаменять эти способы в каком-то объекте контекста.

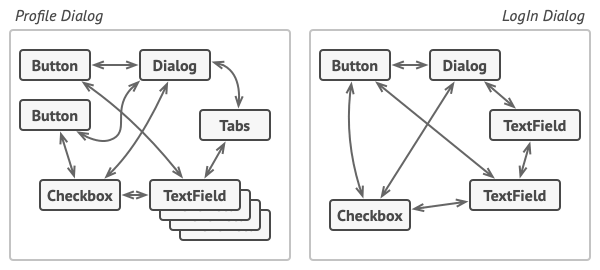
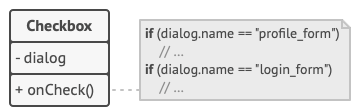
Если Команду нужно копировать перед вставкой в историю выполненных команд, вам может помочь Прототип.

Посетитель можно рассматривать как расширенный аналог Команды, который способен работать сразу с несколькими видами получателей.

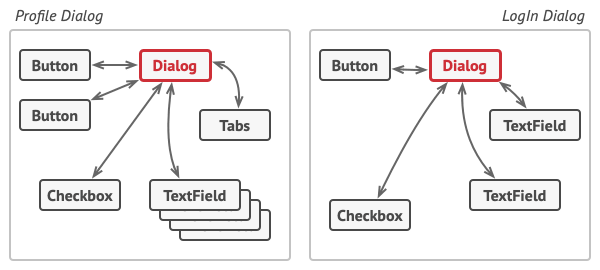
Посредник

* **Посредник** — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.
* 

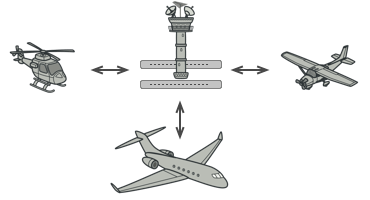
### Проблема

* Предположим, что у вас есть диалог создания профиля пользователя. Он состоит из всевозможных элементов управления — текстовых полей, чекбоксов, кнопок.
* 
* Отдельные элементы диалога должны взаимодействовать друг с другом. Так, например, чекбокс «у меня есть собака» открывает скрытое поле для ввода имени домашнего любимца, а клик по кнопке отправки запускает проверку значений всех полей формы.
* 
* Прописав эту логику прямо в коде элементов управления, вы поставите крест на их повторном использовании в других местах приложения. Они станут слишком тесно связанными с элементами диалога редактирования профиля, которые не нужны в других контекстах. Поэтому вы сможете использовать либо все элементы сразу, либо ни одного.

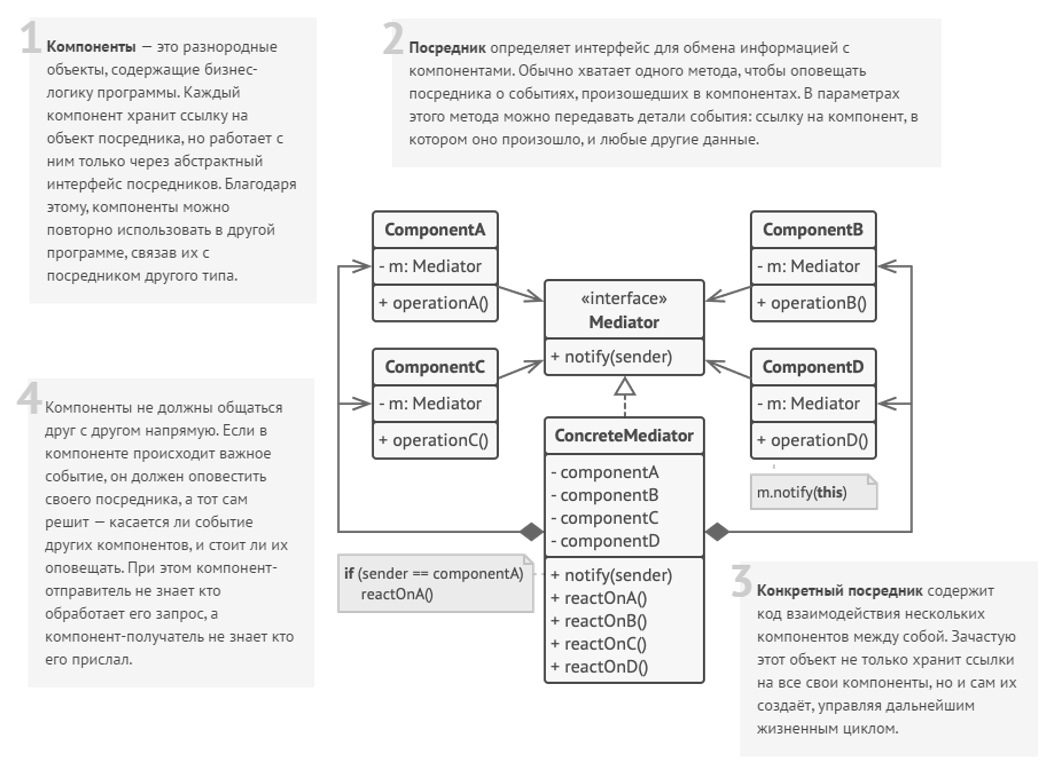
### Решение

* Паттерн Посредник заставляет объекты общаться не напрямую друг с другом, а через отдельный объект-посредник, который знает, кому нужно перенаправить тот или иной запрос. Благодаря этому, компоненты системы будут зависеть только от посредника, а не от десятков других компонентов.
* В нашем примере посредником мог бы стать диалог. Скорее всего, класс диалога и так знает, из каких элементов состоит, поэтому никаких новых связей добавлять в него не придётся.
* 
* Основные изменения произойдут внутри отдельных элементов диалога. Если раньше при получении клика от пользователя объект кнопки сам проверял значения полей диалога, то теперь его единственной обязанностью будет сообщить диалогу о том, что произошёл клик. Получив извещение, диалог выполнит все необходимые проверки полей. Таким образом, вместо нескольких зависимостей от остальных элементов кнопка получит только одну — от самого диалога.
* Чтобы сделать код ещё более гибким, можно выделить общий интерфейс для всех посредников, то есть диалогов программы. Наша кнопка станет зависимой не от конкретного диалога создания пользователя, а от абстрактного, что позволит использовать её и в других диалогах.
* Таким образом, посредник скрывает в себе все сложные связи и зависимости между классами отдельных компонентов программы. А чем меньше связей имеют классы, тем проще их изменять, расширять и повторно использовать.

### Аналогия из жизни

* 
* Пилоты садящихся или улетающих самолётов не общаются напрямую с другими пилотами. Вместо этого они связываются с диспетчером, который координирует действия нескольких самолётов одновременно. Без диспетчера пилотам приходилось бы все время быть начеку и следить за всеми окружающими самолётами самостоятельно, а это приводило бы к частым катастрофам в небе.
* Важно понимать, что диспетчер не нужен во время всего полёта. Он задействован только в зоне аэропорта, когда нужно координировать взаимодействие многих самолётов.

### Структура

* 

### Применимость

* Когда вам сложно менять некоторые классы из-за того, что они имеют множество хаотичных связей с другими классами.
* Посредник позволяет поместить все эти связи в один класс, после чего вам будет легче их отрефакторить, сделать более понятными и гибкими.
* Когда вы не можете повторно использовать класс, поскольку он зависит от уймы других классов.
* После применения паттерна компоненты теряют прежние связи с другими компонентами, а всё их общение происходит косвенно, через объект-посредник.
* Когда вам приходится создавать множество подклассов компонентов, чтобы использовать одни и те же компоненты в разных контекстах.
* Если раньше изменение отношений в одном компоненте могли повлечь за собой лавину изменений во всех остальных компонентах, то теперь вам достаточно создать подкласс посредника и поменять в нём связи между компонентами.

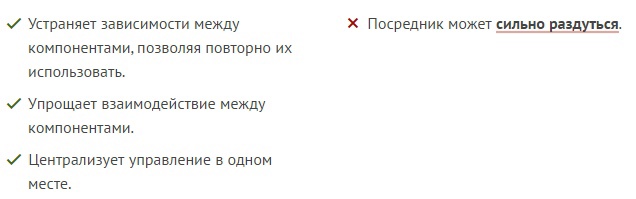
### Шаги реализации

1. Найдите группу тесно переплетённых классов, отвязав которые друг от друга, можно получить некоторую пользу. Например, чтобы повторно использовать их код в другой программе.
2. Создайте общий интерфейс посредников и опишите в нём методы для взаимодействия с компонентами. В простейшем случае достаточно одного метода для получения оповещений от компонентов.

* Этот интерфейс необходим, если вы хотите повторно использовать классы компонентов для других задач. В этом случае всё, что нужно сделать — это создать новый класс конкретного посредника.

1. Реализуйте этот интерфейс в классе конкретного посредника. Поместите в него поля, которые будут содержать ссылки на все объекты компонентов.
2. Вы можете пойти дальше и переместить код создания компонентов в класс посредника, после чего он может напоминать фабрику или фасад.
3. Компоненты тоже должны иметь ссылку на объект посредника. Связь между ними удобнее всего установить, подавая посредника в параметры конструктора компонентов.
4. Измените код компонентов так, чтобы они вызывали метод оповещения посредника, вместо методов других компонентов. С противоположной стороны, посредник должен вызывать методы нужного компонента, когда получает оповещение от компонента.

### Преимущества и недостатки



## Отношения с другими паттернами

Цепочка обязанностей, Команда, Посредник и Наблюдатель показывают различные способы работы отправителей запросов с их получателями:

Цепочка обязанностей передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.

Команда устанавливает косвенную одностороннюю связь от отправителей к получателям.

Посредник убирает прямую связь между отправителями и получателями, заставляя их общаться опосредованно, через себя.

Наблюдатель передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.

Посредник и Фасад похожи тем, что пытаются организовать работу множества существующих классов.

Фасад создаёт упрощённый интерфейс к подсистеме, не внося в неё никакой добавочной функциональности. Сама подсистема не знает о существовании Фасада. Классы подсистемы общаются друг с другом напрямую.

Посредник централизует общение между компонентами системы. Компоненты системы знают только о существовании Посредника, у них нет прямого доступа к другим компонентам.

Разница между Посредником и Наблюдателем не всегда очевидна. Чаще всего они выступают как конкуренты, но иногда могут работать вместе.

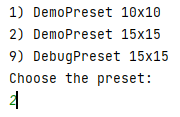
Цель Посредника — убрать обоюдные зависимости между компонентами системы. Вместо этого они становятся зависимыми от самого посредника. С другой стороны, цель Наблюдателя — обеспечить динамическую одностороннюю связь, в которой одни объекты косвенно зависят от других.

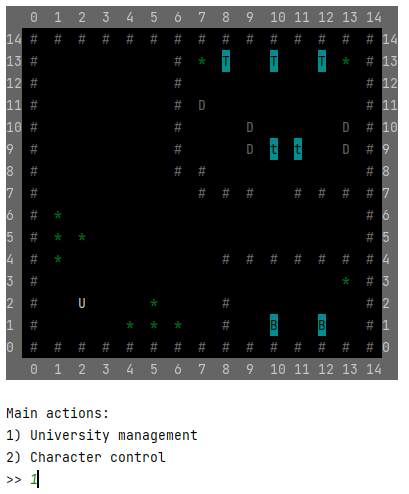
Довольно популярна реализация Посредника при помощи Наблюдателя. При этом объект посредника будет выступать издателем, а все остальные компоненты станут подписчиками и смогут динамически следить за событиями, происходящими в посреднике. В этом случае трудно понять, чем же отличаются оба паттерна.

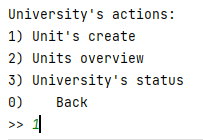
Но Посредник имеет и другие реализации, когда отдельные компоненты жёстко привязаны к объекту посредника. Такой код вряд ли будет напоминать Наблюдателя, но всё же останется Посредником.

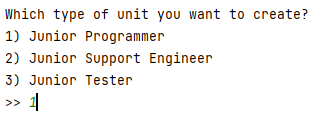
Напротив, в случае реализации посредника с помощью Наблюдателя представим такую программу, в которой каждый компонент системы становится издателем. Компоненты могут подписываться друг на друга, в то же время не привязываясь к конкретным классам. Программа будет состоять из целой сети Наблюдателей, не имея центрального объекта-Посредника.

**Экспериментальные результаты.**

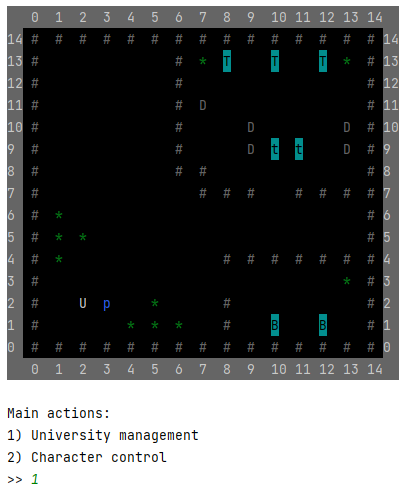


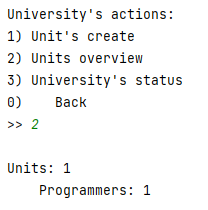


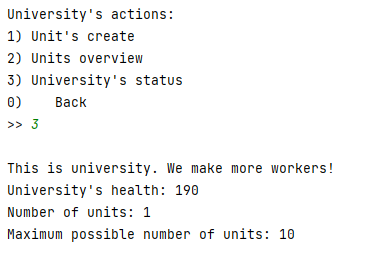


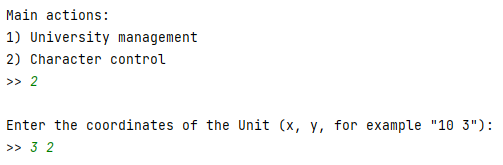


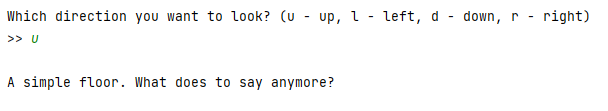


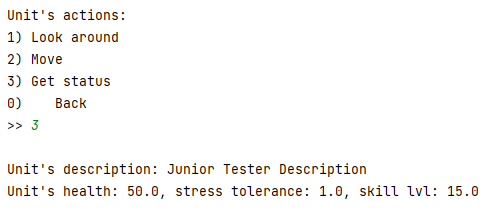


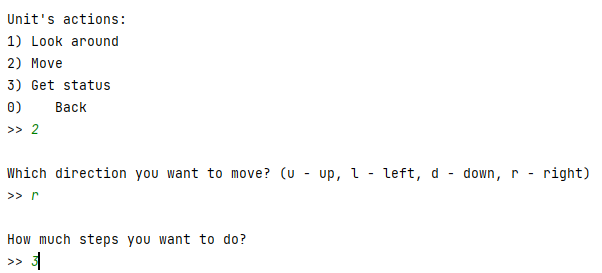


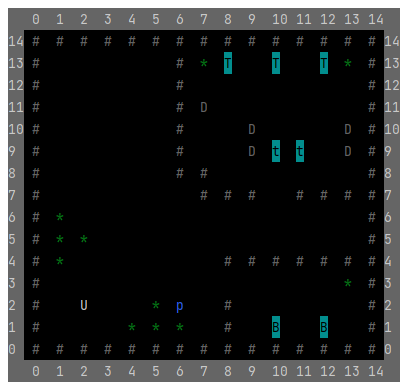










  
Рисунок 5 – Результат работы программы (полный код программы представлен в приложении А)

Приложение а  
полный код программы

<https://github.com/Celeron404/leti2021-2/tree/master/src/main/java/com/leti2021_2>