*Слайд 2 (суть паттерна)*

Паттерн определяет объект – *Посредника*, который инкапсулирует способ взаимодействия множества других объектов. Это позволяет независимо и легко изменять взаимодействие между ними. Говоря проще, у нас есть несколько объектов, которые как-то связаны между собой. Эти связи приносят некоторые неудобства, поэтому для упрощения логики программы можно перенести все связи в один класс-посредник.

*Слайд 3 (пример)*

Вот пример-иллюстрация. Предположим, что у нас есть диалог создания профиля пользователя. Он состоит из всевозможных элементов управления — текстовых полей, чекбоксов, кнопок и тп.

*Слайд 4 (проблема)*

Отдельные элементы диалога должны взаимодействовать друг с другом. Так, например, чекбокс «у меня есть сестра» открывает скрытое поле для ввода имени родственника, а клик по кнопке отправки запускает проверку значений всех полей формы. Прописывание этой логики прямо в коде элементов управления приведёт к тому, что их повторное использование в других местах приложения окажется очень, очень затруднительным. Они станут слишком тесно связанными с элементами диалога редактирования профиля, которые не нужны в других контекстах. Поэтому придется использовать либо все элементы сразу, либо ни одного.

*Слайд 5 (сопряженность и так везде)*

Вот что-то подобное будет в коде каждого элемента. Сильная сопряженность налицо.

*Слайд 6 (проблема)*

Как же избавиться от этой проблемы? Применим паттерн Посредник.

*Слайд 7 (решение)*

Теперь объекты общаются не напрямую друг с другом, а через отдельный объект-посредник, который знает, кому нужно перенаправить тот или иной запрос. Благодаря этому, *компоненты*, т.е. элементы, системы будут зависеть только от посредника, а не от десятков других компонентов.

В этом примере посредником является диалог. Скорее всего, класс диалога и так знает, из каких элементов состоит, поэтому никаких новых связей добавлять в него не придётся.

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.

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.

*Слайд 8 (структура)*

Давайте теперь рассмотрим структуру паттерна более подробно.

*Слайд 9 (структура)*

Для начала определим интерфейс Посредника. Если компоненты системы будут применяться только в одном контексте, то надобность в интерфейсе отпадает, однако повторное использование этих компонентов уже в разных контекстах побуждает все же сделать общий интерфейс для всех возможный посредников. Заметим, что интерфейс описывает публичный метод notify. Именно этот метод будут вызывать компоненты, чтобы оповещать посредника о событиях, произошедших в компонентах. В параметрах этого метода можно передавать детали события: ссылку на компонент, в котором оно произошло, и любые другие данные.

*Слайд 10 (структура)*

И тут на сцену выходит сразу четыре компонента. Для краткости будем называть их A, B, C и D. Компоненты – это разнородные объекты, содержащие бизнес-логику программы.

*Слайд 11 (структура)*

Каждый компонент хранит ссылку на объект посредника. Бизнес логика описана в методе operation. В данном примере работа с посредником осуществляется через интерфейс, что позволяет повторно использовать компоненты в другой программе, связав их с посредником другого типа.

*Слайд 12 (структура)*

Конкретный посредник, реализующий соответствующий интерфейс, содержит код взаимодействия нескольких компонентов между собой.

*Слайд 13 (структура)*

Конкретный посредник хранит ссылки на все свои компоненты (а также может их создавать самостоятельно), управляет их дальнейшим жизненным циклом. Компоненты не должны общаться друг с другом напрямую. Если в компоненте происходит важное событие, он должен оповестить своего посредника, а тот сам решит — касается ли событие других компонентов, и стоит ли их оповещать. При этом компонент-отправитель не знает кто обработает его запрос, а компонент-получатель не знает кто его прислал. В примере показана связь между посредником и компонентом A.

*Слайд 14 (структура)*

Когда какой-нибудь компонент уведомляет посредника о событии (это метод notify), обработка уведомления может выглядеть примерно так, как показано внизу на слайде. Если уведомление пришло от компонента A, то нужно вызвать соответствующий метод-обработчик.

*Слайд 15 (структура)*

Здесь уже показана связь со всеми остальными компонентами. У посредника также есть ссылки на них и соответствующие методы-обработчики.  
Собственно, теперь на слайде отображена полная структура паттерна.

*Слайд 16*

А теперь поговорим о ситуациях, в которых поможет применение этого паттерна.

*Слайд 17*

Когда 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 будет легче их отрефакторить, сделать более понятными и гибкими.

*Слайд 18*

Когда не получается 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.

*Слайд 19*

Когда 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 достаточно создать подкласс посредника и поменять в нём связи между компонентами. Согласитесь, создать один подкласс и поменять в нем связи будет скорее всего куда проще, чем создавать энное количество подклассов компонентов, в которых придется менять логику.

*Слайд 20*

Так, разобрались со структурой паттерна, с ситуациями, когда паттерн можно применить. Давайте теперь подытожим.

*Слайд 21*

Преимущества.

*Слайд 22*

Ослабление связей между компонентами. Устранение зависимостей позволяет повторно использовать эти компоненты. Изменять классы посредников и компонентов можно независимо друг от друга.

*Слайд 23*

При использовании паттерна механизм посредничества выделяется в отдельную концепцию, а ее инкапсуляция в одном объекте позволяет сосредоточится именно на взаимодействии объектов, а не на их индивидуальном поведении. Это способствует прояснению имеющихся в системе взаимодействий.

*Слайд 24*

Посредник заменяет взаимодействие вида «все со всеми» взаимодействиями вида «один со всеми», то есть один посредник взаимодействует со всеми компонентами. Отношения вида «один ко многим» проще для понимания, сопровождения и расширения.

*Слайд 25*

Но все ли однозначно с централизацией управления? До текущего момента я не упомянул ни одного недостатка паттерна. Неужели их нет?

*Слайд 26*

К сожалению, это не так. Посредник заменяет сложность взаимодействия сложностью класса-посредника. Поскольку посредник инкапсулирует логику взаимодействия, то он может быть сложнее отдельных компонентов. В результате сам посредник превращается в монолит, который трудно сопровождать. Говоря проще, посредник может сильно раздуться. Частично, а иногда, может, и полностью, решить эту проблему позволяет обычный рефакторинг.

*Слайд 27*

В завершение моего скромного рассказа давайте рассмотрим применение паттерна на конкретном примере – простом приложении для ведения заметок.