**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, а клик по кнопке отправки запускает проверку значений всех полей.

Прописав эту логику прямо в коде элементов управления, у нас не получится повторно использовать данный элемент.

**Решение**

Решение: использование паттерна “Посредник”

Данный паттерн 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 сам наш диалог, поэтому никаких новых связей добавлять в него не придётся.

Основные изменения произойдут внутри отдельных элементов диалога. Например, если раньше при получении клика от пользователя кнопка сама проверяла значения полей, то теперь её единственной обязанностью будет сообщить диалогу о том, что произошёл клик. Получив извещение, диалог выполнит все необходимые проверки полей.

**Определение**

Таким образом, посредник – это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник. Почему это хорошо: чем меньше связей имеют классы, тем проще их изменять, расширять и повторно использовать.

**Аналогия из жизни**

Пилоты садящихся или улетающих самолетов не общаются напрямую с друг с другом. 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.

**Структура**

1. Компоненты — это разнородные объекты, содержащие бизнес-логику программы. Каждый компонент хранит ссылку на объект посредника, но работает с ним только через абстрактный интерфейс посредников.
2. Интерфейс посредника для обмена информацией с компонентами. Обычно хватает одного метода для оповещения посредника о событиях. В параметрах этого метода можно передавать детали события: ссылку на компонент, в котором оно произошло, и любые другие данные.
3. Конкретный посредник содержит код взаимодействия нескольких компонентов между собой.

Компоненты не должны общаться друг с другом напрямую. Если в компоненте происходит важное событие, он должен известить 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.

**Пример**

open class Component(private val dialog: Mediator) {

fun click() {

dialog.notify(this, "click")

}

}

class TextField(dialog: Mediator) : Component(dialog) {

var value: String = ""

}

class Button(dialog: Mediator) : Component(dialog)

interface Mediator {

fun notify(sender: Component, event: String)

}

class LoginDialog : Mediator {

private val database = Database

private lateinit var loginField: TextField

private lateinit var passwordField: TextField

private lateinit var loginButton: Button

override fun notify(sender: Component, event: String) {

if (sender is Button && event == "click") {

if (database.checkUser(loginField.value, passwordField.value)) {

*println*("Вы успешно вошли")

} else {

*println*("Такого пользователя нет")

}

}

}

fun addLoginField(field: TextField) {

loginField = field

}

fun addPasswordField(field: TextField) {

passwordField = field

}

fun addLoginButton(button: Button) {

loginButton = button

}

}

object Database {

private val users = *listOf*(Pair("qwerty", "qwerty"), Pair("Oleg", "iLoveCats"))

fun checkUser(login: String, password: String) = Pair(login, password) in users

}

fun main() {

val dialog = LoginDialog()

val loginField = TextField(dialog)

val passwordField = TextField(dialog)

val loginButton = Button(dialog)

dialog.addLoginField(loginField)

dialog.addPasswordField(passwordField)

dialog.addLoginButton(loginButton)

loginField.value = "qwerty"

passwordField.value = "qwerty"

loginButton.click() *// Вы успешно вошли*

loginField.click() *// \* Ничего не произойдёт \**

loginField.value = "Oleg"

passwordField.value = "iDontLikeCats"

loginButton.click() *// Такого пользователя нет*

}

**Связь с другими паттернами**

* **Цепочка обязанностей**, **Команда**, **Посредник** и **Наблюдатель** показывают различные способы работы отправителей запросов с их получателями:
  + **Цепочка обязанностей** передает 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** передает 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.
* **Посредник** и **Наблюдатель** могут показаться похожими, но это не так.
  + Во-первых, несмотря на то, что **Посредника** можно реализовать с помощью **Наблюдателя**, существую и другие способы реализации (ранее я как раз привел пример реализации без наблюдателя)
  + Во-вторых, Цель **Посредника** — убрать обоюдные зависимости между компонентами системы. Вместо этого они становятся зависимыми только от самого посредника. Цель **Наблюдателя** — обеспечить динамическую одностороннюю связь, в которой одни объекты косвенно зависят от других.

**Вывод**

* Устраняет зависимости между компонентами, позволяя повторно их использовать.
* Упрощает взаимодействие между компонентами.
* Централизует управление в одном месте.
* Может сильно раздуться