Частное учреждение образования

Колледж бизнеса и права

УТВЕРЖДАЮ

Заведующий

методическим кабинетом

\_\_\_\_\_\_\_\_\_\_\_Е.В. Паскал

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_2021

| Специальность: 2-40 01 01 «Программное обеспечение информационных технологий» | Дисциплина: «Основы кроссплатформенного  программирования» |
| --- | --- |

ЛАБОРАТОРНАЯ РАБОТА № 18

Инструкционно-технологическая карта

Тема: «Паттерны в языке. Поведенческие паттерны»

Цель: Научить работать с поведенческими паттернами в языке, в частности с наблюдателем и посредником.

Время выполнения: 2 часа

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретические сведения;

2. Ответить на контрольные вопросы;

3. Откомпилировать примеры программ из раздела «Теоретические сведения»;

4. Выполнить ИДЗ.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ ДЛЯ ВЫПОЛНЕНИЯ РАБОТЫ

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

Поведенческие:

Наблюдатель — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

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

Проблема

2

Представьте, что вы имеете два объекта: Покупатель и Магазин. В магазин вот-вот должны завезти новый товар, который интересен покупателю. Покупатель может каждый день ходить в магазин, чтобы проверить наличие товара. Но при этом он будет злиться, без толку тратя своё драгоценное время.

С другой стороны, магазин может разослать спам каждому своему покупателю. Многих это расстроит, так как товар специфический, и не всем он нужен.

Получается конфликт: либо покупатель тратит время на периодические проверки, либо магазин тратит ресурсы на бесполезные оповещения. Решение

Давайте называть Издателями те объекты, которые содержат важное или интересное для других состояние. Остальные объекты, которые хотят отслеживать изменения этого состояния, назовём Подписчиками.

Паттерн Наблюдатель предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.

Теперь самое интересное. Когда в издателе будет происходить важное событие, он будет проходиться по списку подписчиков и оповещать их об этом, вызывая определённый метод объектов-подписчиков.

Издателю безразлично, какой класс будет иметь тот или иной подписчик, так как все они должны следовать общему интерфейсу и иметь единый метод оповещения.

Увидев, как складно всё работает, вы можете выделить общий интерфейс, описывающий методы подписки и отписки, и для всех издателей. После этого подписчики смогут работать с разными типами издателей, а также получать оповещения от них через один и тот же метод.

Применимость

Когда после изменения состояния одного объекта требуется что-то сделать в других, но вы не знаете наперёд, какие именно объекты должны отреагировать. Описанная проблема может возникнуть при разработке библиотек пользовательского интерфейса, когда вам надо дать возможность сторонним классам реагировать на клики по кнопкам.

Паттерн Наблюдатель позволяет любому объекту с интерфейсом подписчика зарегистрироваться на получение оповещений о событиях, происходящих в объектах-издателях.

Когда одни объекты должны наблюдать за другими, но только в определённых случаях. Издатели ведут динамические списки. Все наблюдатели могут подписываться или отписываться от получения оповещений прямо во время выполнения программы.

Шаги реализации

3

1.Разбейте вашу функциональность на две части: независимое ядро и опциональные зависимые части. Независимое ядро станет издателем. Зависимые части станут подписчиками.

2.Создайте интерфейс подписчиков. Обычно в нём достаточно определить единственный метод оповещения.

3.Создайте интерфейс издателей и опишите в нём операции управления подпиской. Помните, что издатель должен работать только с общим интерфейсом подписчиков.

4.Вам нужно решить, куда поместить код ведения подписки, ведь он обычно бывает одинаков для всех типов издателей. Самый очевидный способ — вынести этот код в промежуточный абстрактный класс, от которого будут наследоваться все издатели.

Но если вы интегрируете паттерн в существующие классы, то создать новый базовый класс может быть затруднительно. В этом случае вы можете поместить логику подписки во вспомогательный объект и делегировать ему работу из издателей.

5.Создайте классы конкретных издателей. Реализуйте их так, чтобы после каждого изменения состояния они отправляли оповещения всем своим подписчикам.

6.Реализуйте метод оповещения в конкретных подписчиках. Не забудьте предусмотреть параметры, через которые издатель мог бы отправлять какие то данные, связанные с происшедшим событием.

Возможен и другой вариант, когда подписчик, получив оповещение, сам возьмёт из объекта издателя нужные данные. Но в этом случае вы будете вынуждены привязать класс подписчика к конкретному классу издателя.

7.Клиент должен создавать необходимое количество объектов подписчиков и подписывать их у издателей.

Преимущества:

• Издатели не зависят от конкретных классов подписчиков и наоборот.

• Вы можете подписывать и отписывать получателей на лету. • Реализует принцип открытости/закрытости.

Недостатки:

• Подписчики оповещаются в случайном порядке.

Подписка и оповещения. В этом примере Наблюдатель используется для передачи событий между объектами текстового редактора. Всякий раз когда объект редактора меняет своё состояние, он оповещает своих наблюдателей. Объекты EmailNotificationListener и LogOpenListener следят за этими уведомлениями и выполняют полезную работу в ответ.

Классы подписчиков не связаны с классом редактора и могут быть повторно использованы в других приложениях если потребуется. Класс Editor зависит только от общего интерфейса подписчиков. Это позволяет добавлять новые типы подписчиков не меняя существующего кода редактора.

4

Посредник

Проблема

Предположим, что у вас есть диалог создания профиля пользователя. Он состоит из всевозможных элементов управления — текстовых полей, чекбоксов, кнопок. Беспорядочные связи между элементами пользовательского интерфейса. Беспорядочные связи между элементами пользовательского интерфейса.

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.

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.

5

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.

3. Реализуйте этот интерфейс в классе конкретного посредника. Поместите в него поля, которые будут содержать ссылки на все объекты компонентов.

4. Вы можете пойти дальше и переместить код создания компонентов в класс посредника, после чего он может напоминать фабрику или фасад.

5. Компоненты тоже должны иметь ссылку на объект посредника. Связь между ними удобнее всего установить, подавая посредника в параметры конструктора компонентов.

6. 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:

• Устраняет зависимости между компонентами, позволяя повторно их использовать.

• Упрощает взаимодействие между компонентами.

• Централизует управление в одном месте.

Недостатки:

• Посредник может сильно раздуться.

Этот пример показывает, как организовать множество элементов интерфейса при помощи посредника так, чтобы они не знали и не зависели друг от друга.

6

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое паттерн наблюдатель?

2. Когда удобно использовать наблюдатель?

3. Что такое паттерн посредник?

4. Когда стоит использовать посредник?

5. Преимущества и недостатки наблюдателя и посредника. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Варианты индивидуального задания.

Вывод интересующей пользователя информации в окне. Реализовать: нечетный вариант паттерн наблюдатель, четный вариант – посредник. 1. Разработать систему Почтовое отделение. Из издательства в почтовое

отделение поступают издаваемые газеты и журналы. Почтовое отделение отправляет полученные печатные издания соответствующим подписчикам. 2. Имеется фабрика по производству телефонов, планшетов и компьютеров трех различных торговых марок. Пользователь выбирает нужную ему категорию.

1. ДОМАШНЕЕ ЗАДАНИЕ

Индивидуальное задание

2. ЛИТЕРАТУРА

Альфред В., Ахо Компиляторы. Принципы, технологии и инструментарий, Вильямс, 2017.

Преподаватель A.С.Кибисова

Рассмотрено на заседании цикловой комиссии

программного обеспечения информационных

технологий

Протокол № \_\_\_\_ от «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2021

Председатель ЦК \_\_\_\_\_\_\_\_\_\_В.Ю.Михалевич