Паттерны поведения связаны с алгоритмами и распределением обязанностей между объектами. Речь в них идет не только о самих объектах и классах, но и о типичных способах взаимодействия. Паттерны поведения характеризуют сложный поток управления, который трудно проследить во время выполнения программы.

Посредник

Объектно-ориентированное проектирование способствует распределению поведения между объектами. Но при этом в получившейся структуре объектов может возникнуть много связей или (в худшем случае) каждому объекту придется иметь информацию обо всех остальных.

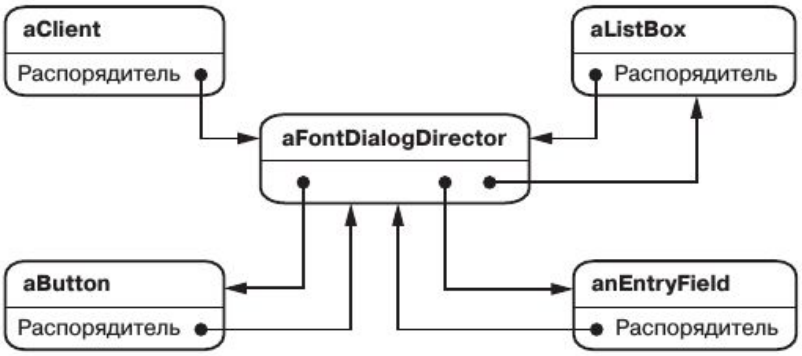
Несмотря на то, что разбиение системы на множество объектов в общем случае повышает степень повторного использования, изобилие взаимосвязей приводит к обратному эффекту. Если взаимосвязей слишком много, тогда система подобна монолиту и маловероятно, что объект сможет работать без поддержки других объектов. Более того, существенно изменить поведение системы практически невозможно, поскольку оно распределено между многими объектами. Если вы предпримете подобную попытку, то для настройки поведения системы вам придется определять множество подклассов.

Рассмотрим реализацию диалоговых окон в графическом интерфейсе пользователя. Часто между разными виджетами в диалоговом окне существуют зависимости. Например, если одно из полей ввода пустое, то определенная кнопка недоступна. При выборе из списка может измениться содержимое поля ввода. И наоборот, ввод текста в некоторое поле может автоматически привести к выбору одного или нескольких элементов списка. Если в поле ввода присутствует какой-то текст, то могут быть активизированы кнопки, позволяющие произвести определенное действие над этим текстом, например изменить либо удалить его.

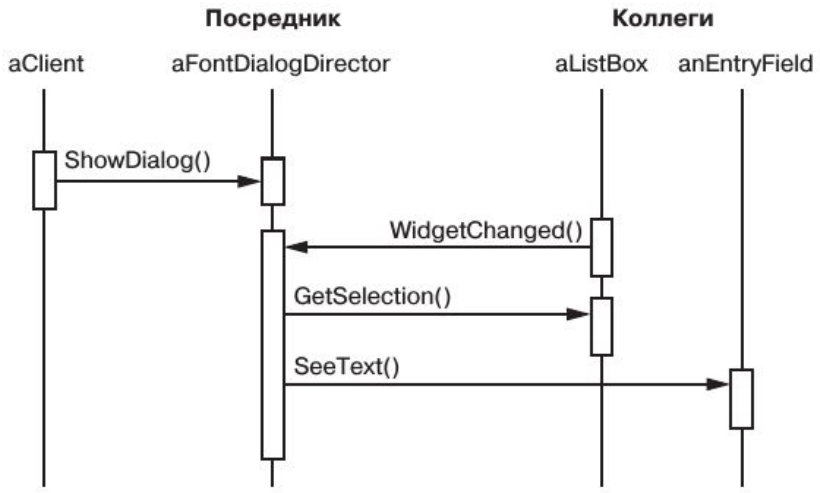
В разных диалоговых окнах зависимости между виджетами могут быть различными. Поэтому, несмотря на то, что во всех окнах встречаются однотипные виджеты, просто взять и повторно использовать готовые классы виджетов не удастся, придется производить настройку с целью учета зависимостей. Индивидуальная настройка каждого виджета – утомительное занятие, ибо участвующих классов слишком много.

Всех этих проблем можно избежать, если инкапсулировать коллективное поведение в отдельном объекте-посреднике. Посредник отвечает за координацию взаимодействий между группой объектов. Он избавляет входящие в группу объекты от необходимости явно ссылаться друг на друга. Все объекты располагают информацией только о посреднике, поэтому количество взаимосвязей сокращается.

Так, класс FontDialogDirector может служить посредником между виджетами в диалоговом окне. Объект этого класса знает обо всех виджетах в окне и координирует взаимодействие между ними, то есть выполняет функции центра коммуникаций.

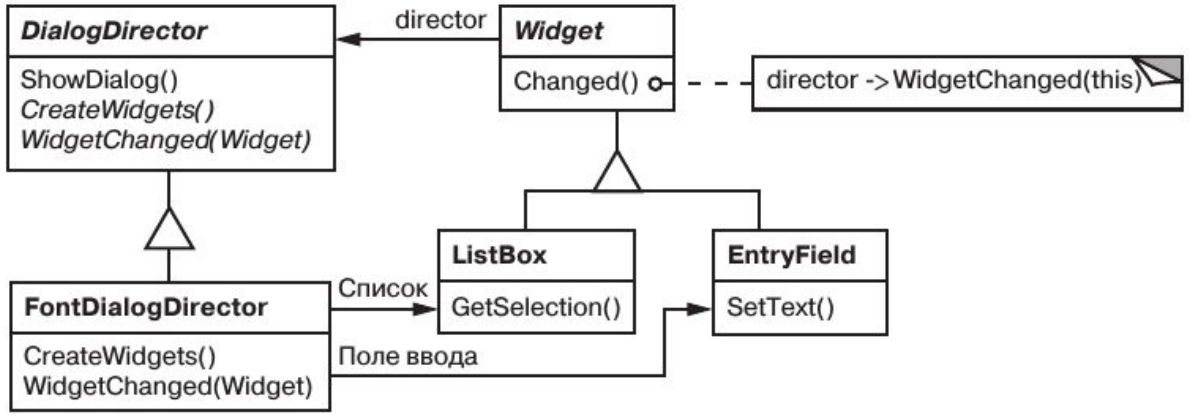


На следующей диаграмме взаимодействий показано, как объекты кооперируются друг с другом, реагируя на изменение выбранного элемента списка.



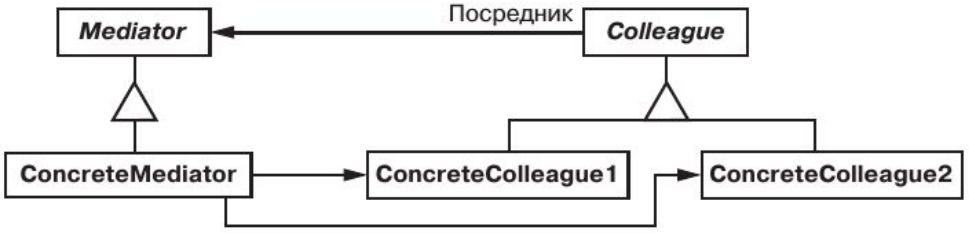
Виджеты общаются друг с другом не напрямую, а через распорядителя. Им вообще не нужно владеть информацией друг о друге, они осведомлены лишь о существовании распорядителя. А коль скоро поведение локализовано в одном классе, то его несложно модифицировать или менять поведение путем расширения или замены этого класса.

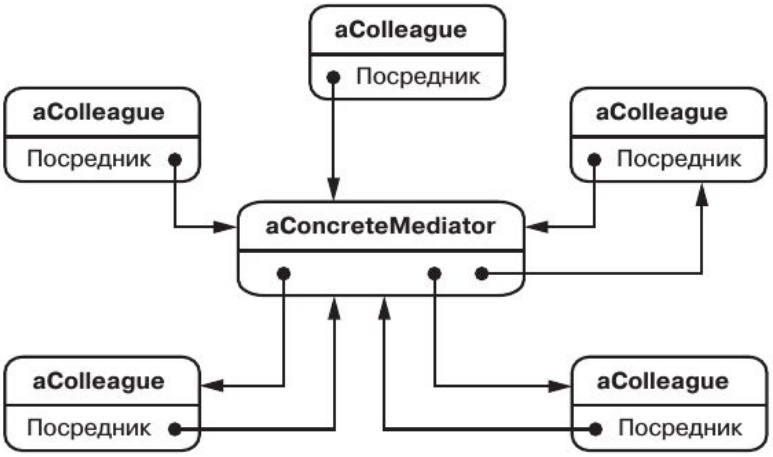
Абстракцию FontDialogDirector можно было бы интегрировать в библиотеку классов так, как показано на рисунке.



DialogDirector – это абстрактный класс, который определяет поведение диалогового окна в целом. Клиенты вызывают его операцию ShowDialog для отображения окна на экране. CreateWidgets – это абстрактная операция для создания виджетов в диалоговом окне. WidgetChanged – еще одна абстрактная операция; с ее помощью виджеты сообщают распорядителю об изменениях. Подклассы DialogDirector замещают операции CreateWidgets (для создания нужных виджетов) и WidgetChanged (для обработки извещений об изменениях).

В общем виде шаблон Посредник выглядит так:





У паттерна посредник есть следующие достоинства и недостатки:

- *устраняет связанность между коллегами*. Посредник обеспечивает слабую связанность коллег. Изменять классы Colleague и Mediator можно независимо друг от друга;
- *повышает переиспользуемость классов*. Посредник локализует поведение, которое в противном случае пришлось бы распределять между несколькими объектами. Для изменения поведения нужно породить подклассы только от класса посредника Mediator, классы коллег Colleague можно использовать повторно без каких бы то ни было изменений;
- *упрощает протоколы взаимодействия объектов*. Посредник заменяет дисциплину взаимодействия «многие ко многим» дисциплиной «один ко многим», то есть один посредник взаимодействует со всеми коллегами. Отношения вида «один ко многим» проще для понимания, сопровождения и расширения;
- *абстрагирует способ кооперирования объектов*. Выделение механизма посредничества в отдельную концепцию и инкапсуляция ее в одном объекте позволяет сосредоточиться именно на взаимодействии объектов, а не на их индивидуальном поведении. Это дает возможность прояснить имеющиеся в системе взаимодействия;

- *централизует управление*. Паттерн посредник переносит сложность взаимодействия в класс-посредник. Поскольку посредник инкапсулирует протоколы, то он может быть сложнее отдельных коллег. В результате сам посредник может стать монолитом, который трудно сопровождать. В перспективе имеет все шансы стать антипаттерном God Object.

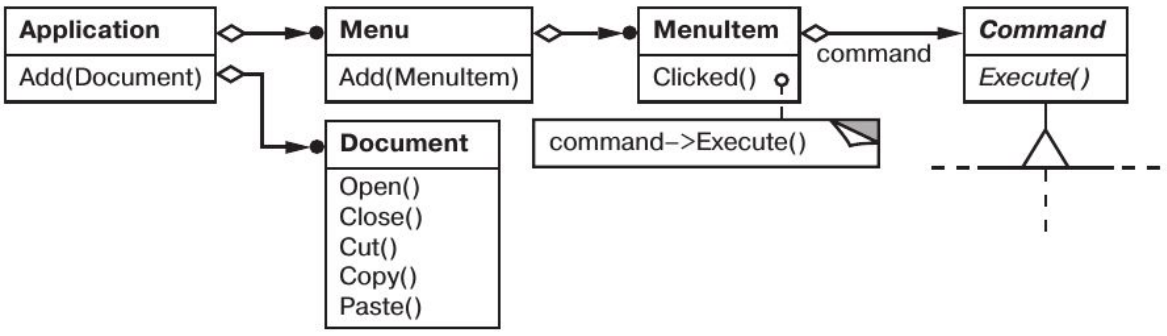
Используйте паттерн посредник, когда:

- имеются объекты, связи между которыми сложны и четко определены. Получающиеся при этом взаимозависимости не структурированы и трудны для понимания;
- нельзя повторно использовать объект, поскольку он обменивается информацией со многими другими объектами;
- поведение, распределенное между несколькими классами, должно поддаваться настройке без порождения множества подклассов.

Команда

Иногда необходимо посылать объектам запросы, ничего не зная о том, выполнение какой операции запрошено и кто является получателем. Например, в библиотеках для построения пользовательских интерфейсов встречаются такие объекты, как кнопки и меню, которые посылают запрос в ответ на действие пользователя. Но в саму библиотеку не заложена возможность обрабатывать этот запрос, так как только приложение, использующее ее, располагает информацией о том, что следует сделать. Проектировщик библиотеки не владеет никакой информацией о получателе запроса и о том, какие операции тот должен выполнить.

Паттерн команда позволяет библиотечным объектам отправлять запросы неизвестным объектам приложения, преобразовав сам запрос в объект. Этот объект можно хранить и передавать, как и любой другой. В основе описываемого паттерна лежит абстрактный класс Command, в котором объявлен интерфейс для выполнения операций. В простейшей своей форме этот интерфейс состоит из одной абстрактной операции Execute. Конкретные подклассы Command определяют пару «получатель-действие», сохраняя получателя в переменной экземпляра, и реализуют операцию Execute, так чтобы она посылала запрос. У получателя есть информация, необходимая для выполнения запроса.

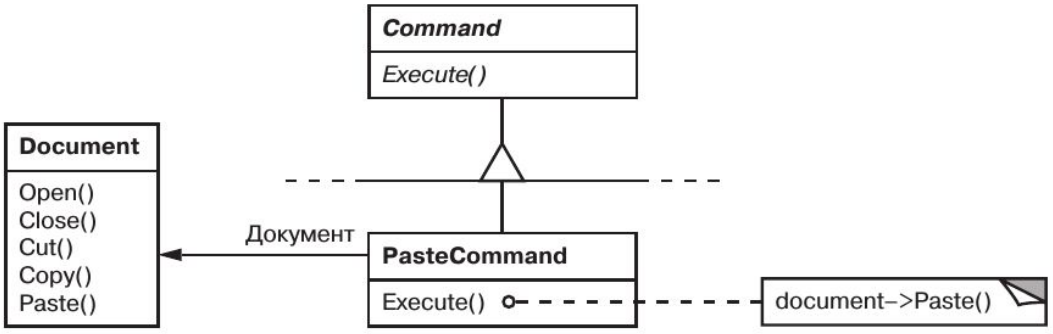


С помощью объектов Command легко реализуются меню. Каждый пункт меню – это экземпляр класса MenuItem. Сами меню и все их пункты создает класс Application

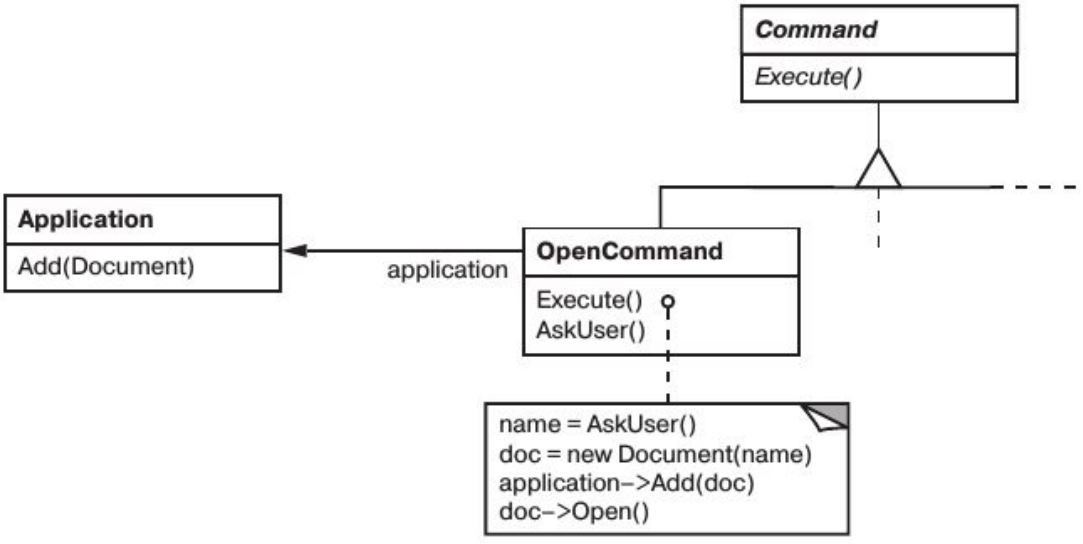
наряду со всеми остальными элементами пользовательского интерфейса. Класс Application отслеживает также открытые пользователем документы.

Приложение конфигурирует каждый объект MenuItem экземпляром конкретного подкласса Command. Когда пользователь выбирает некоторый пункт меню, ассоциированный с ним объект MenuItem вызывает Execute для своего объекта команды, а Execute выполняет операцию. Объекты MenuItem не имеют информации, какой подкласс класса Command они используют. Подклассы Command хранят информацию о получателе запроса и вызывают одну или несколько операций этого получателя.

Например, подкласс PasteCommand поддерживает вставку текста из буфера обмена в документ. Получателем для PasteCommand является Document, который был передан при создании объекта. Операция Execute вызывает операцию Paste документа-получателя.

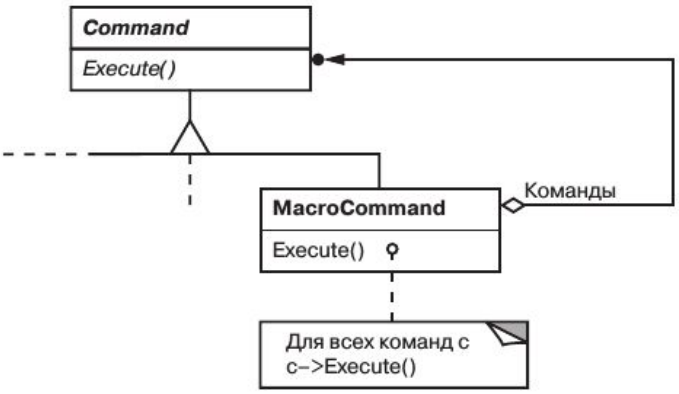


Для подкласса OpenCommand операция Execute ведет себя по-другому: она запрашивает у пользователя имя документа, создает соответствующий объект Document, извещает о новом документе приложение-получатель и открывает этот документ.

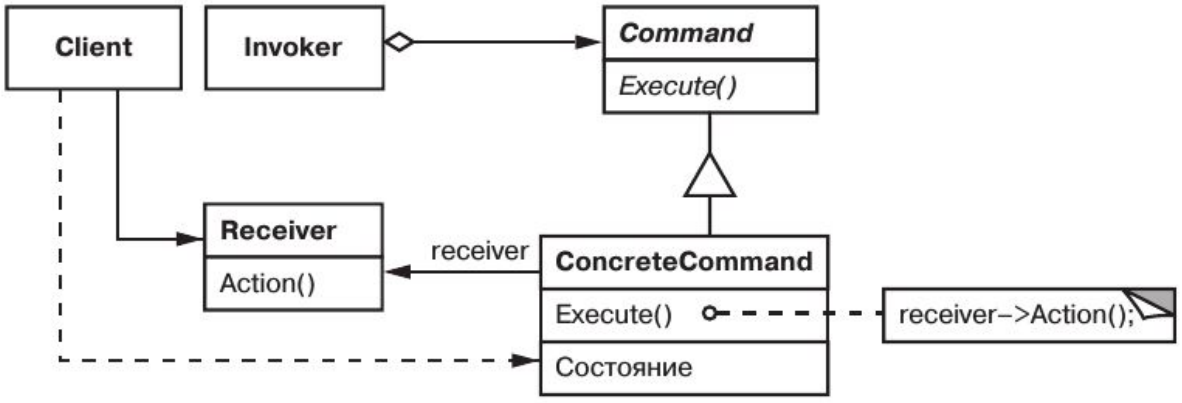


Иногда объект MenuItem должен выполнить последовательность команд. Например, пункт меню для центрирования страницы стандартного размера можно было бы сконструировать сразу из двух объектов: CenterDocumentCommand и NormalSizeCommand. Поскольку такое комбинирование команд – явление обычное, то мы можем определить класс MacroCommand, позволяющий объекту MenuItem

выполнять произвольное число команд. MacroCommand – это конкретный подкласс класса Command, который просто выполняет последовательность команд. У него нет явного получателя, поскольку для каждой команды определен свой собственный.

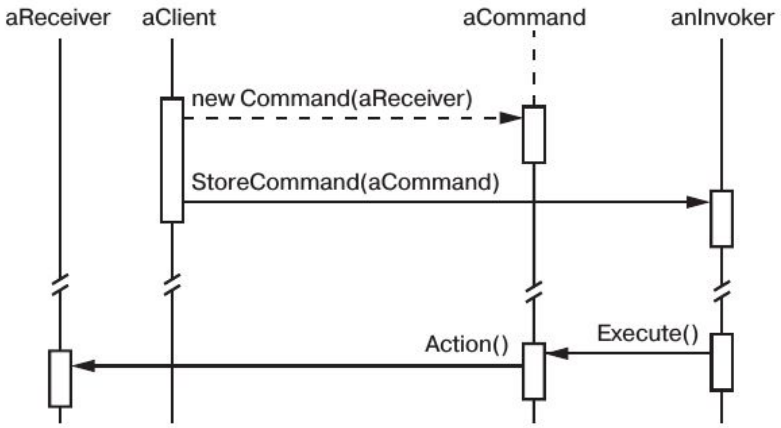


Обратите внимание, что в каждом из приведенных примеров паттерн Команда отделяет объект, инициирующий операцию, от объекта, который знает, как ее выполнить. Это позволяет добиться высокой гибкости при проектировании пользовательского интерфейса. Пункт меню и кнопка одновременно могут быть ассоциированы в приложении с некоторой функцией, для этого достаточно приписать обоим элементам один и тот же экземпляр конкретного подкласса класса Command. Мы можем динамически подменять команды, что очень полезно для реализации контекстно-зависимых меню. Можно также поддержать сценарии, если комбинировать простые команды в более сложные. Все это выполнимо потому, что объект, инициирующий запрос, должен располагать информацией лишь о том, как его отправить, а не о том, как его выполнить.



Итого, в рамках паттерна Команда клиент создает объект ConcreteCommand и устанавливает для него получателя. Инициатор Invoker сохраняет объект ConcreteCommand, а потом по необходимости отправляет запрос, вызывая операцию команды Execute. Если поддерживается отмена выполненных действий, то ConcreteCommand перед вызовом Execute сохраняет информацию о состоянии, достаточную для выполнения отката. Объект ConcreteCommand вызывает операции получателя для выполнения запроса.

На следующей диаграмме видно, как Command разрывает связь между инициатором и получателем (а также запросом, который должен выполнить последний).



Используйте паттерн Команда, когда хотите:

- *параметризовать объекты выполняемым действием*, как в случае с пунктами меню MenuItem. В процедурном языке такую параметризацию можно выразить с помощью callback-функции, то есть такой функции, которая регистрируется, чтобы быть вызванной позднее. Команды представляют собой объектно-ориентированную альтернативу callback-функциям;
- *определять, ставить в очередь и выполнять запросы в разное время*. Время жизни объекта Command необязательно должно зависеть от времени жизни исходного запроса. Если получателя запроса удается реализовать так, чтобы он не зависел от адресного пространства, то объект-команду можно передать другому процессу, который займется его выполнением;
- *поддержать отмену операций*. Операция Execute объекта Command может сохранить состояние, необходимое для отката действий, выполненных командой. В этом случае в интерфейсе класса Command должна быть дополнительная операция Unexecute, которая отменяет действия, выполненные предшествующим обращением к Execute. Выполненные команды хранятся в списке (или стеке) истории. Для реализации произвольного числа уровней отмены и повтора команд нужно обходить этот список соответственно в обратном и прямом направлениях, вызывая при посещении каждого элемента команду Unexecute или Execute;
- *поддержать протоколирование изменений*, чтобы их можно было выполнить повторно после аварийной остановки системы. Дополнив интерфейс класса Command операциями сохранения и загрузки, вы сможете вести протокол изменений во внешней памяти. Для восстановления после сбоя нужно будет загрузить сохраненные команды с диска и повторно выполнить их по очереди с помощью операции Execute;
- *структурировать систему на основе высокоуровневых операций, построенных из примитивных*. Такая структура типична для информационных систем, поддерживающих транзакции. Транзакция инкапсулирует набор изменений данных. Паттерн Команда позволяет моделировать транзакции. У всех команд есть общий интерфейс, что дает возможность работать одинаково

с любыми транзакциями. С помощью этого паттерна можно легко добавлять в систему новые виды транзакций.

Цепочка ответственности

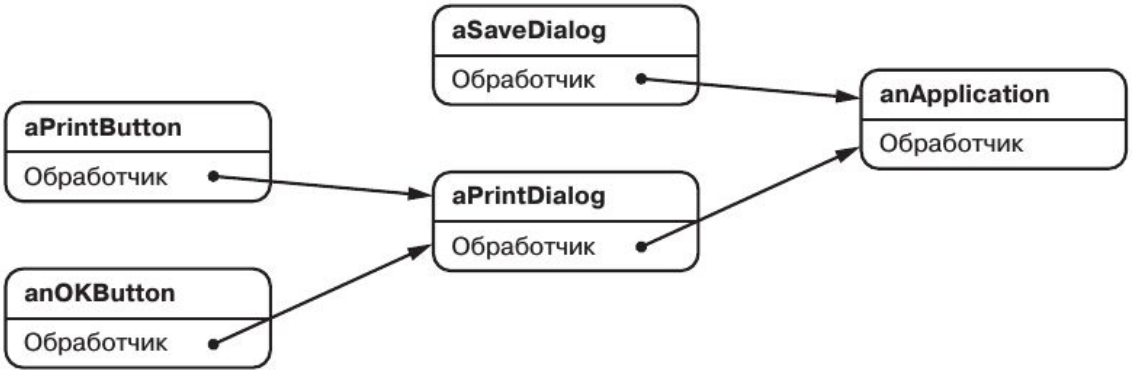
Рассмотрим контекстно-зависимую оперативную справку в графическом интерфейсе пользователя, который может получить дополнительную информацию по любой части интерфейса, просто щелкнув на ней мышью. Содержание справки зависит от того, какая часть интерфейса и в каком контексте выбрана. Например, справка по кнопке в диалоговом окне может отличаться от справки по аналогичной кнопке в главном окне приложения. Если для некоторой части интерфейса справки нет, то система должна показать информацию о ближайшем контексте, в котором она находится, например о диалоговом окне в целом.

Поэтому естественно было бы организовать справочную информацию от более конкретных разделов к более общим. Кроме того, ясно, что запрос на получение справки обрабатывается одним из нескольких объектов пользовательского интерфейса, каким именно – зависит от контекста и имеющейся в наличии информации.

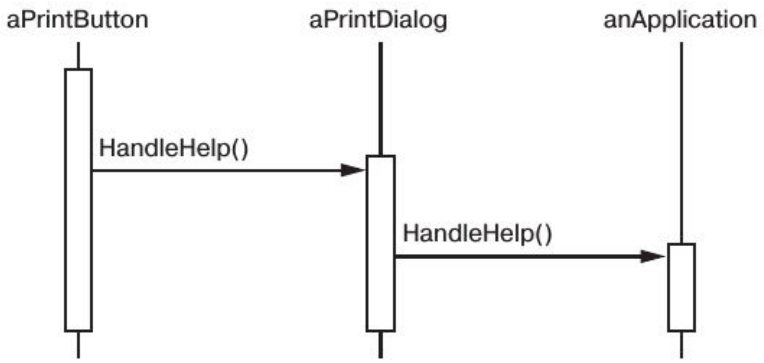
Проблема в том, что объект, инициирующий запрос (например, кнопка), не располагает информацией о том, какой объект в конечном итоге предоставит справку. Нам необходим какой-то способ отделить кнопку-инициатор запроса от объектов, владеющих справочной информацией. Как этого добиться, показывает паттерн Цепочка обязанностей.

Идея заключается в том, чтобы разорвать связь между отправителями и получателями, дав возможность обработать запрос нескольким объектам. Запрос перемещается по цепочке объектов, пока один из них не обработает его.

Первый объект в цепочке получает запрос и либо обрабатывает его сам, либо направляет следующему кандидату в цепочке, который ведет себя точно так же. У объекта, отправившего запрос, информация об обработчике отсутствует.



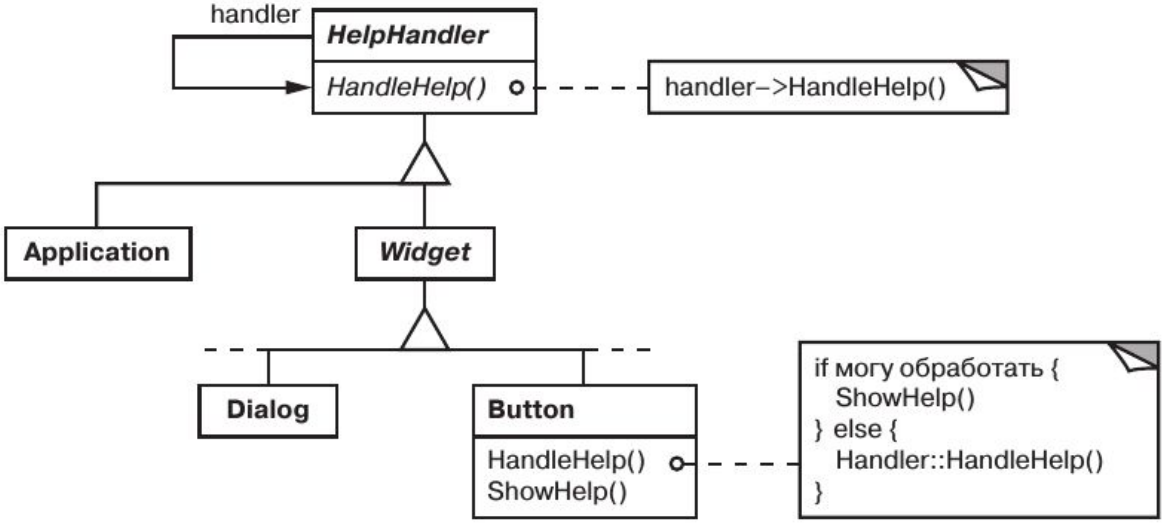
Предположим, что пользователь запрашивает справку по кнопке Print. Она находится в диалоговом окне PrintDialog, содержащем информацию об объекте приложения, которому принадлежит. На представленной диаграмме взаимодействий показано, как запрос на получение справки перемещается по цепочке.



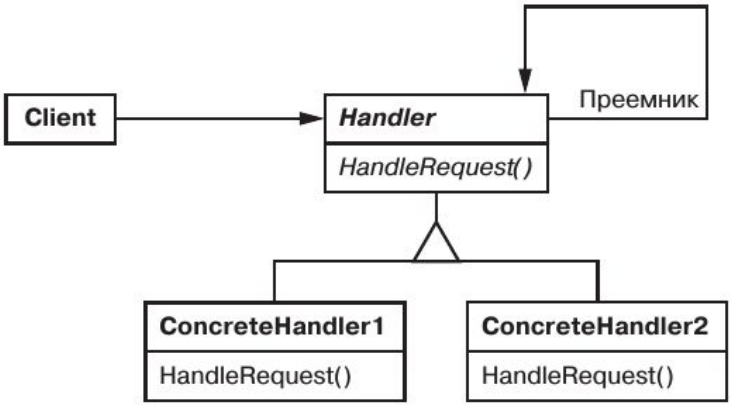
В данном случае ни кнопка aPrintButton, ни окно aPrintDialog не обрабатывают запрос, он достигает объекта anApplication, который может его обработать или игнорировать. У клиента, инициировавшего запрос, нет прямой ссылки на объект, который его в конце концов выполнит.

Чтобы отправить запрос по цепочке и гарантировать анонимность получателя, все объекты в цепочке имеют единый интерфейс для обработки запросов и для доступа к своему преемнику (следующему объекту в цепочке). Например, в системе оперативной справки можно было бы определить класс HelpHandler (предок классов всех объектов-кандидатов или подмешиваемый класс (mixin class)) с операцией HandleHelp. Тогда классы, которые будут обрабатывать запрос, смогут его передать своему родителю.

Для обработки запросов на получение справки классы Button, Dialog и Application пользуются операциями HelpHandler. По умолчанию операция HandleHelp просто перенаправляет запрос своему преемнику. В подклассах эта операция замещается, так что при благоприятных обстоятельствах может выдаваться справочная информация. В противном случае запрос отправляется дальше посредством реализации по умолчанию.



Общий вид шаблона:



Представлять запросы можно по-разному. В простейшей форме, например в случае класса HandleHelp, запрос жестко кодируется как вызов некоторой операции. Это удобно и безопасно, но переадресовывать тогда можно только фиксированный набор запросов, определенных в классе Handler.

Альтернатива – использовать одну функцию-обработчик, которой передается код запроса (скажем, целое число или строка). Так можно поддержать заранее неизвестное число запросов. Единственное требование состоит в том, что отправитель и получатель должны договориться о способе кодирования запроса. Это более гибкий подход, но при реализации нужно использовать условные операторы для раздачи запросов по их коду. Кроме того, не существует безопасного с точки зрения типов способа передачи параметров, поэтому упаковывать и распаковывать их приходится вручную. Очевидно, что это не так безопасно, как прямой вызов операции.

Ещё один вариант -- использовать отдельные объекты-запросы, в которых инкапсулированы параметры запроса (паттерн Команда).

Паттерн Цепочка обязанностей имеет следующие достоинства и недостатки:

- *ослабление связанности*. Этот паттерн освобождает объект от необходимости знать, кто конкретно обработает его запрос. Отправителю и получателю ничего не известно друг о друге, а включенному в цепочку объекту – о структуре цепочки. Таким образом, Цепочка обязанностей помогает упростить взаимосвязи между объектами. Вместо того, чтобы хранить ссылки на все объекты, которые могут стать получателями запроса, объект должен располагать информацией лишь о своем ближайшем преемнике;
- *дополнительная гибкость при распределении обязанностей между объектами*. Цепочка обязанностей позволяет повысить гибкость распределения обязанностей между объектами. Добавить или изменить обязанности по обработке запроса можно, включив в цепочку новых участников или изменив ее каким-то другим образом. Этот подход можно сочетать со статическим порождением подклассов для создания специализированных обработчиков;
- *получение не гарантировано*. Поскольку у запроса нет явного получателя, то нет и гарантий, что он вообще будет обработан: он может достичь конца цепочки и пропасть. Необработанным запрос может оказаться и в случае неправильной конфигурации цепочки.

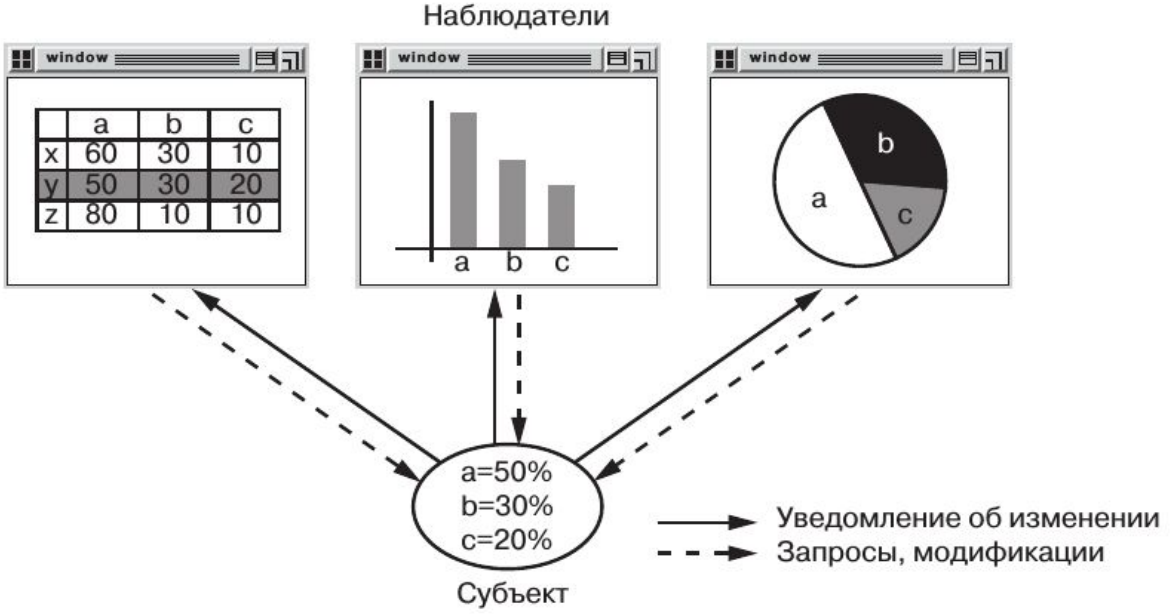
Цепочку обязанностей имеет смысл использовать, когда:

- есть более одного объекта, способного обработать запрос, причем настоящий обработчик заранее неизвестен и должен быть найден автоматически;
- вы хотите отправить запрос одному из нескольких объектов, не указывая явно, какому именно;
- набор объектов, способных обработать запрос, должен задаваться динамически.

Наблюдатель

В результате разбиения системы на множество совместно работающих классов появляется необходимость поддерживать согласованное состояние взаимосвязанных объектов. Но не хотелось бы, чтобы за согласованность надо было платить жесткой связанностью классов, так как это в некоторой степени уменьшает возможности повторного использования.

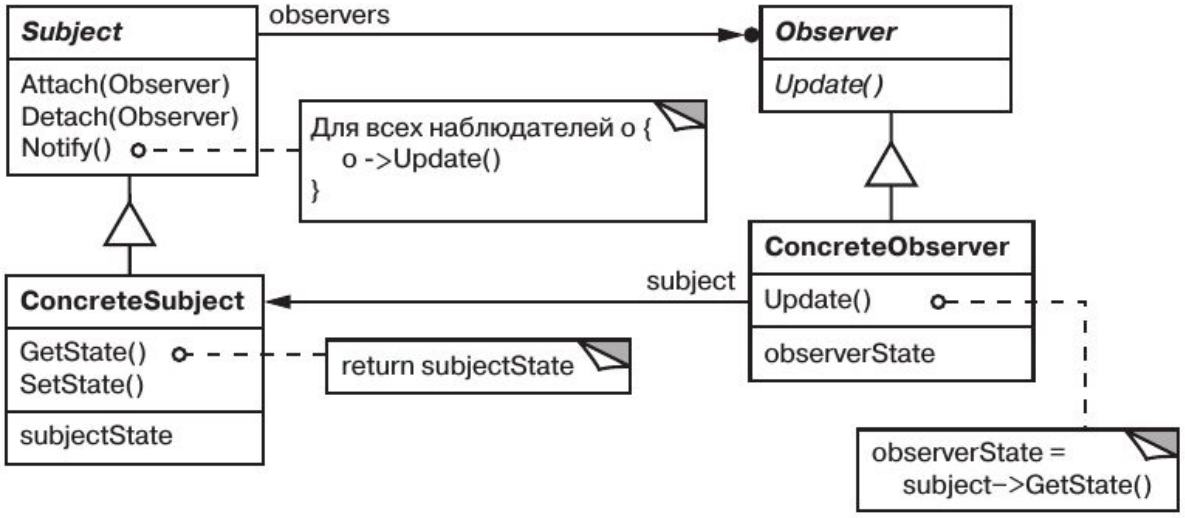
Например, во многих библиотеках для построения графических интерфейсов пользователя презентационные аспекты интерфейса отделены от данных приложения. С классами, описывающими данные и их представление, можно работать автономно. Электронная таблица и диаграмма не имеют информации друг о друге, поэтому вы вправе использовать их по отдельности. Но ведут они себя так, как будто знают друг о друге. Когда пользователь работает с таблицей, все изменения немедленно отражаются на диаграмме, и наоборот.



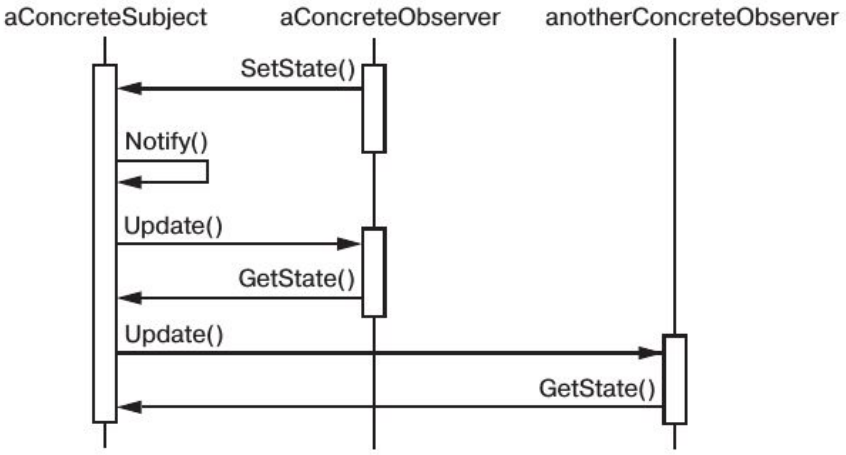
При таком поведении подразумевается, что и электронная таблица, и диаграмма зависят от данных объекта и поэтому должны уведомляться о любых изменениях в его состоянии. И нет никаких причин, ограничивающих количество зависимых объектов; для работы с одними и теми же данными может существовать любое число пользовательских интерфейсов.

Паттерн наблюдатель описывает, как устанавливать такие отношения. Ключевыми объектами в нем являются субъект и наблюдатель. У субъекта может быть сколько угодно зависимых от него наблюдателей. Все наблюдатели

уведомляются об изменениях в состоянии субъекта. Получив уведомление, наблюдатель опрашивает субъекта, чтобы синхронизировать с ним свое состояние.



Объект ConcreteSubject уведомляет своих наблюдателей о любом изменении, которое могло бы привести к рассогласованности состояний наблюдателя и субъекта. После получения от конкретного субъекта уведомления об изменении объект ConcreteObserver может запросить у субъекта дополнительную информацию, которую использует для того, чтобы оказаться в состоянии, согласованном с состоянием субъекта. На диаграмме взаимодействий показаны отношения между субъектом и двумя наблюдателями.



Паттерн Наблюдатель позволяет изменять субъекты и наблюдатели независимо друг от друга. Субъекты разрешается повторно использовать без участия наблюдателей, и наоборот. Это дает возможность добавлять новых наблюдателей без модификации субъекта или других наблюдателей.

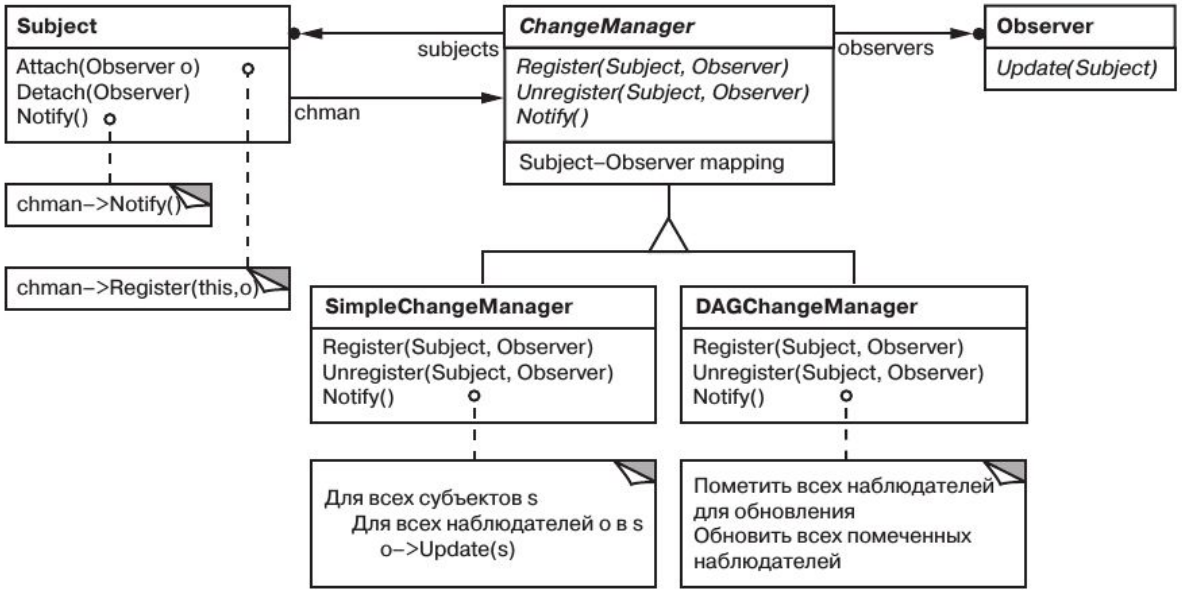
Если отношения зависимости между субъектами и наблюдателями становятся особенно сложными, то может потребоваться объект, инкапсулирующий эти отношения. Будем называть его ChangeManager. Он служит для минимизации объема работы, необходимой для того чтобы наблюдатели могли отразить изменения субъекта. Например, если некоторая операция влечет за собой изменения в нескольких независимых субъектах, то хотелось бы, чтобы наблюдатели

уведомлялись после того, как будут модифицированы все субъекты, дабы не ставить в известность одного и того же наблюдателя несколько раз.

У класса ChangeManager есть три обязанности:

- строить отображение между субъектом и его наблюдателями и предоставлять интерфейс для поддержания отображения в актуальном состоянии. Это освобождает субъектов от необходимости хранить ссылки на своих наблюдателей и наоборот;
- определять конкретную стратегию обновления;
- обновлять всех зависимых наблюдателей по запросу от субъекта.

На следующей диаграмме представлена простая реализация паттерна Наблюдатель с использованием менеджера изменений ChangeManager. Имеется два специализированных менеджера. SimpleChangeManager всегда обновляет всех наблюдателей каждого субъекта, а DAGChangeManager обрабатывает направленные ациклические графы зависимостей между субъектами и их наблюдателями. Когда наблюдатель должен «присматривать» за несколькими субъектами, предпочтительнее использовать DAGChangeManager. В этом случае изменение сразу двух или более субъектов может привести к избыточным обновлениям. Объект DAGChangeManager гарантирует, что наблюдатель в любом случае получит только одно уведомление. Если обновление одного и того же наблюдателя допускается несколько раз подряд, то вполне достаточно объекта SimpleChangeManager.



Рассмотрим некоторые достоинства и недостатки паттерна наблюдатель:

- *абстрактная связанность субъекта и наблюдателя*. Субъект имеет информацию лишь о том, что у него есть ряд наблюдателей, каждый из которых подчиняется простому интерфейсу абстрактного класса Observer. Субъекту неизвестны конкретные классы наблюдателей. Таким образом, связи между субъектами и наблюдателями носят абстрактный характер и сведены к минимуму.

  Поскольку субъект и наблюдатель не являются тесно связанными, то они могут находиться на разных уровнях абстракции системы. Субъект более низкого уровня может уведомлять наблюдателей, находящихся на верхних уровнях, не

нарушая иерархии системы. Если бы субъект и наблюдатель представляли собой единое целое, то получающийся объект либо пересекал бы границы уровней (нарушая принцип их формирования), либо должен был находиться на каком-то одном уровне (компрометируя абстракцию уровня);

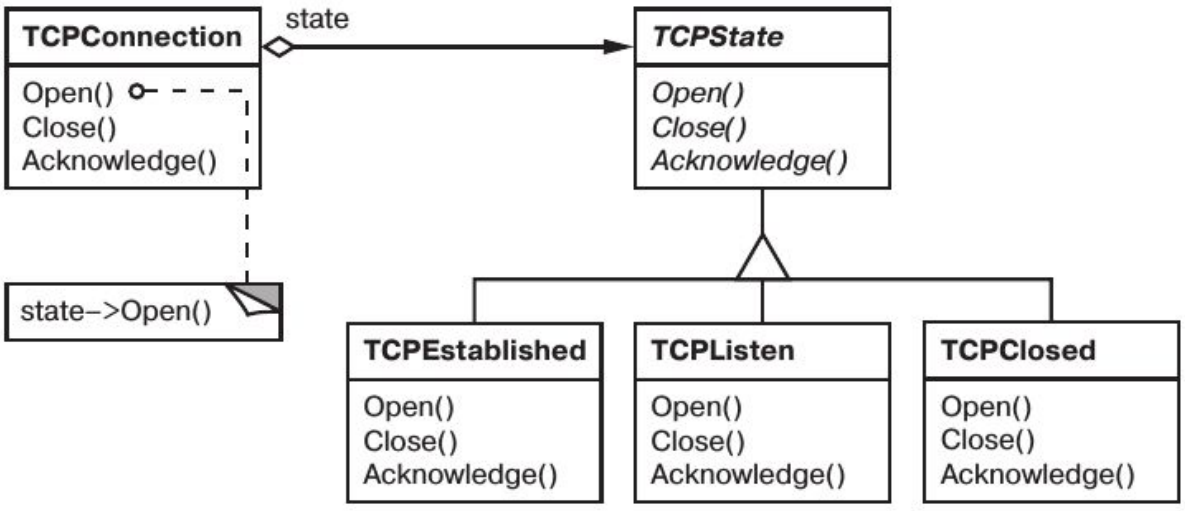
- *поддержка широковещательных коммуникаций*. В отличие от обычного запроса для уведомления, посылаемого субъектом, не нужно задавать определенного получателя. Уведомление автоматически поступает всем подписавшимся на него объектам. Субъекту не нужна информация о количестве таких объектов, от него требуется всего лишь уведомить своих наблюдателей. Поэтому мы можем в любое время добавлять и удалять наблюдателей. Наблюдатель сам решает, обработать полученное уведомление или игнорировать его;
- *неожиданные обновления*. Поскольку наблюдатели не располагают информацией друг о друге, им неизвестно и о том, во что обходится изменение субъекта. Безобидная, на первый взгляд, операция над субъектом может вызвать целый ряд обновлений наблюдателей и зависящих от них объектов. Более того, нечетко определенные или плохо поддерживаемые критерии зависимости могут стать причиной непредвиденных обновлений, отследить которые очень сложно.

  Эта проблема усугубляется еще и тем, что простой протокол обновления не содержит никаких сведений о том, что именно изменилось в субъекте. Без дополнительного протокола, помогающего выяснить характер изменений, наблюдатели будут вынуждены проделать сложную работу для косвенного получения такой информации.

Состояние

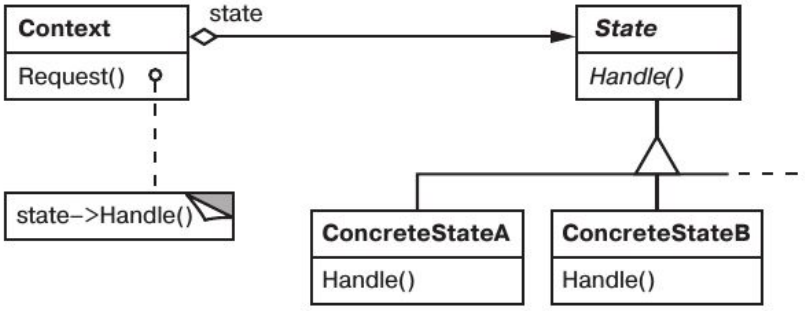
Рассмотрим класс TCPConnection, с помощью которого представлено сетевое соединение. Объект этого класса может находиться в одном из нескольких состояний: Established, Listening, Closed. Когда объект TCPConnection получает запросы от других объектов, то в зависимости от текущего состояния он отвечает по-разному. Например, ответ на запрос Open зависит от того, находится ли соединение в состоянии Closed или Established. Паттерн состояние описывает, каким образом объект TCPConnection может вести себя по-разному, находясь в различных состояниях.

Основная идея этого паттерна заключается в том, чтобы ввести абстрактный класс TCPState для представления различных состояний соединения. Этот класс объявляет интерфейс, общий для всех классов, описывающих различные рабочие состояния. В подклассах TCPState реализовано поведение, специфичное для конкретного состояния. Например, в классах TCPEstablished и TCPClosed реализовано поведение, характерное для состояний Established и Closed соответственно.



Класс TCPConnection хранит у себя объект состояния (экземпляр некоторого подкласса TCPState), представляющий текущее состояние соединения, и делегирует все зависящие от состояния запросы этому объекту. TCPConnection использует свой экземпляр подкласса TCPState для выполнения операций, свойственных только данному состоянию соединения.

При каждом изменении состояния соединения TCPConnection изменяет свой объект-состояние. Например, когда установленное соединение закрывается, TCPConnection заменяет экземпляр класса TCPEstablished экземпляром TCPClosed.



Паттерн Состояние не фиксирует, какой участник определяет критерий перехода между состояниями. Если критерии зафиксированы, то их можно реализовать непосредственно в классе Context. Однако в общем случае более гибкий подход заключается в том, чтобы позволить самим подклассам класса State определять следующее состояние и момент перехода. Для этого в класс Context надо добавить интерфейс, позволяющий объектам State установить состояние контекста.

Такую децентрализованную логику переходов проще модифицировать и расширять – нужно лишь определить новые подклассы State. Недостаток децентрализации в том, что каждый подкласс State должен знать еще хотя бы об одном подклассе, что вносит реализационные зависимости между подклассами.

Результаты использования паттерна состояние:

- *локализует зависящее от состояния поведение и делит его на части, соответствующие состояниям*. Паттерн состояние помещает все поведение, ассоциированное с конкретным состоянием, в отдельный объект. Поскольку

зависящий от состояния код целиком находится в одном из подклассов класса State, то добавлять новые состояния и переходы можно просто путем порождения новых подклассов.

Вместо этого можно было бы использовать данные члены для определения внутренних состояний, тогда операции объекта Context проверяли бы эти данные. Но в таком случае похожие условные операторы или операторы ветвления были бы разбросаны по всему коду класса Context. При этом добавление нового состояния потребовало бы изменения нескольких операций, что затруднило бы сопровождение.

Паттерн Состояние позволяет решить эту проблему, но одновременно порождает другую, поскольку поведение для различных состояний оказывается распределенным между несколькими подклассами State. Это увеличивает число классов. Конечно, один класс компактнее, но если состояний много, то такое распределение эффективнее, так как в противном случае пришлось бы иметь дело с громоздкими условными операторами.

Наличие громоздких условных операторов нежелательно, равно как и наличие длинных процедур. Они слишком монолитны, вот почему модификация и расширение кода становится проблемой. Паттерн Состояние предлагает более удачный способ структурирования зависящего от состояния кода. Логика, описывающая переходы между состояниями, больше не заключена в монолитные операторы if или switch, а распределена между подклассами State. При инкапсуляции каждого перехода и действия в класс состояние становится полноценным объектом. Это улучшает структуру кода и проясняет его назначение;

- *делает явными переходы между состояниями*. Если объект определяет свое текущее состояние исключительно в терминах внутренних данных, то переходы между состояниями не имеют явного представления; они проявляются лишь как присваивания некоторым переменным. Ввод отдельных объектов для различных состояний делает переходы более явными. Кроме того, объекты State могут защитить контекст Context от рассогласования внутренних переменных, поскольку переходы с точки зрения контекста – это атомарные действия. Для осуществления перехода надо изменить значение только одной переменной (объектной переменной State в классе Context), а не нескольких;
- *объекты состояния можно разделять*. Если в объекте состояния State отсутствуют переменные экземпляра, то есть представляемое им состояние кодируется исключительно самим типом, то разные контексты могут разделять один и тот же объект State. Когда состояния разделяются таким образом, они являются, по сути дела, приспособленцами, у которых нет внутреннего состояния, а есть только поведение.

Используйте паттерн Состояние в следующих случаях:

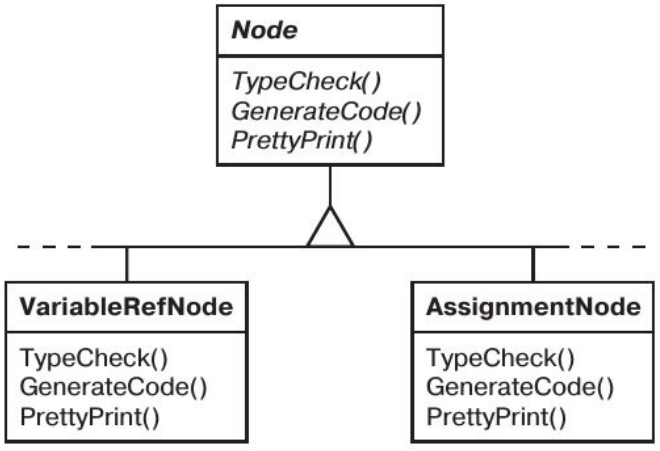
- когда поведение объекта зависит от его состояния и должно изменяться во время выполнения;
- когда в коде операций встречаются состоящие из многих ветвей условные операторы, в которых выбор ветви зависит от состояния. Обычно в таком случае состояние представлено перечисляемыми константами. Часто одна и та же структура условного оператора повторяется в нескольких операциях.

Паттерн состояние предлагает поместить каждую ветвь в отдельный класс. Это позволяет трактовать состояние объекта как самостоятельный объект, который может изменяться независимо от других.

Посетитель

Рассмотрим компилятор, который представляет программу в виде абстрактного синтаксического дерева. Над такими деревьями он должен выполнять операции статического семантического анализа, например проверять, что все переменные определены. Еще ему нужно генерировать код. Аналогично можно было бы определить операции контроля типов, оптимизации кода, анализа потока выполнения, проверки того, что каждой переменной было присвоено конкретное значение перед первым использованием, и т.д. Более того, абстрактные синтаксические деревья могли бы служить для красивой печати программы, реструктурирования кода и вычисления различных метрик программы.

В большинстве таких операций узлы дерева, представляющие операторы присваивания, следует рассматривать иначе, чем узлы, представляющие переменные и арифметические выражения. Поэтому один класс будет создан для операторов присваивания, другой – для доступа к переменным, третий – для арифметических выражений и т.д. Набор классов узлов, конечно, зависит от компилируемого языка, но не очень сильно.



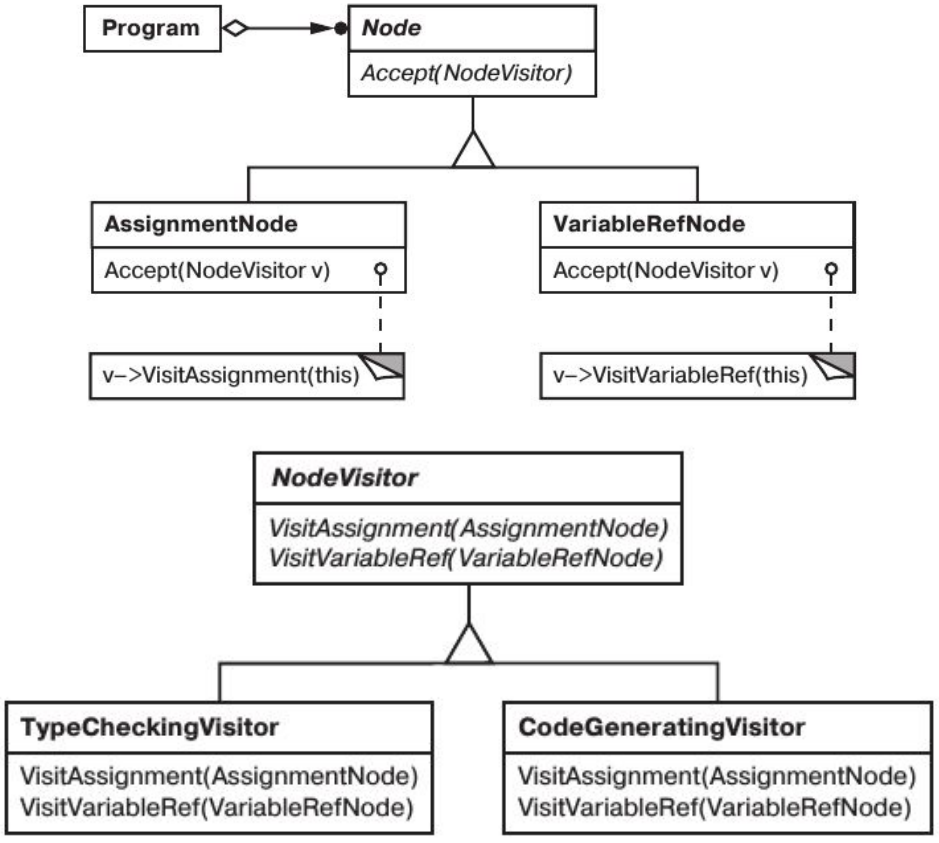
На представленной диаграмме показана часть иерархии классов Node. Проблема здесь в том, что если раскидать все операции по классам различных узлов, то получится система, которую трудно понять, сопровождать и изменять. Вряд ли кто-нибудь разберется в программе, если код, отвечающий за проверку типов, будет перемешан с кодом, реализующим красивую печать или анализ потока выполнения. Кроме того, добавление любой новой операции потребует перекомпиляции всех классов. Оптимальный вариант – наличие возможности добавлять операции по отдельности и отсутствие зависимости классов узлов от применяемых к ним операций.

И того, и другого можно добиться, если поместить взаимосвязанные операции из каждого класса в отдельный объект, называемый посетителем, и передавать его элементам абстрактного синтаксического дерева по мере обхода. Принимая посетителя, элемент посылает ему запрос, в котором содержится, в частности, класс элемента. Кроме того, в запросе присутствует в виде аргумента и сам элемент.

Посетителю в данной ситуации предстоит выполнить операцию над элементом, ту самую, которая наверняка находилась бы в классе элемента.

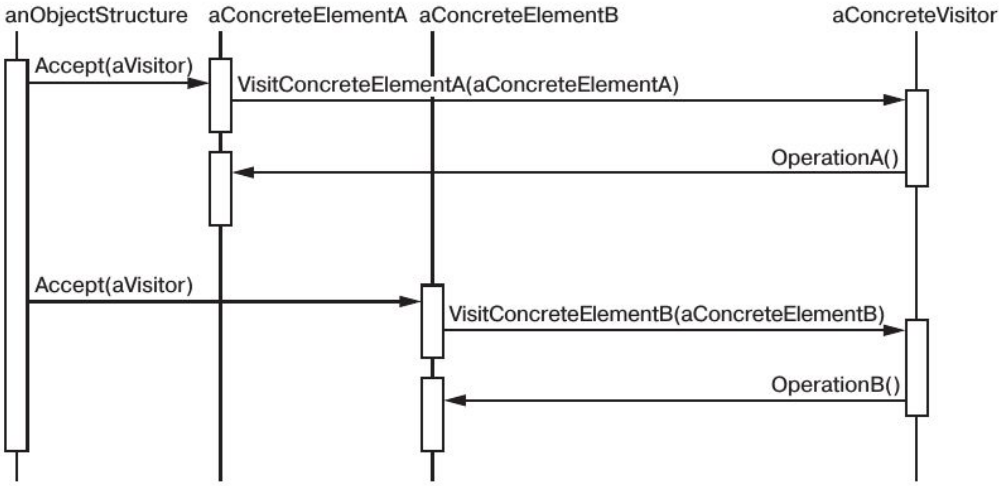
Например, компилятор, который не использует посетителей, мог бы проверить тип процедуры, вызвав операцию TypeCheck для представляющего ее абстрактного синтаксического дерева. Каждый узел дерева должен был реализовать операцию TypeCheck путем рекурсивного вызова ее же для своих компонентов. Если же компилятор проверяет тип процедуры посредством посетителей, то ему достаточно создать объект класса TypeCheckingVisitor и вызвать для дерева операцию Accept, передав ей этот объект в качестве аргумента. Каждый узел должен был реализовать Accept путем обращения к посетителю: узел, соответствующий оператору присваивания, вызывает операцию посетителя VisitAssignment, а узел, ссылающийся на переменную, – операцию VisitVariableReference. То, что раньше было операцией TypeCheck в классе AssignmentNode, стало операцией VisitAssignment в классе TypeCheckingVisitor.

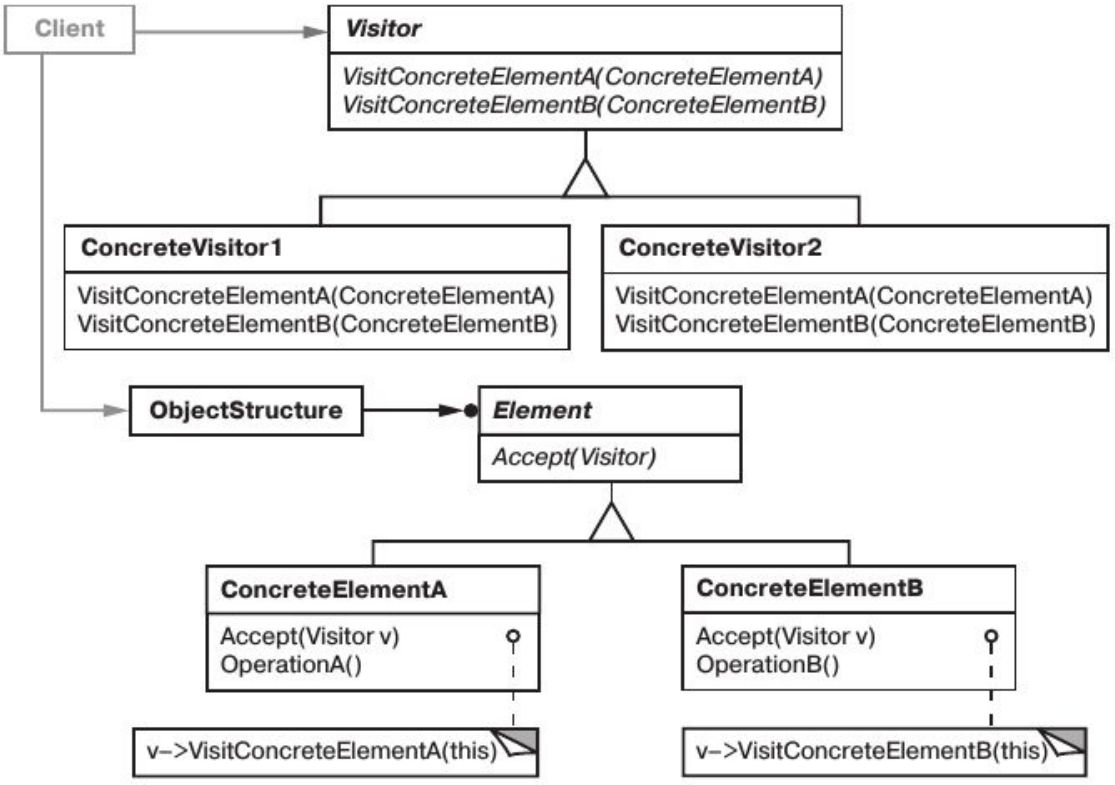
Чтобы посетители могли заниматься не только проверкой типов, нам необходим абстрактный класс NodeVisitor, являющийся родителем для всех посетителей синтаксического дерева. Приложение, которому нужно вычислять метрики программы, определило бы новые подклассы NodeVisitor, так что нам не пришлось бы добавлять зависящий от приложения код в классы узлов. Паттерн посетитель инкапсулирует операции, выполняемые на каждой фазе компиляции, в классе Visitor, ассоциированном с этой фазой.



Применяя паттерн посетитель, вы определяете две иерархии классов: одну для элементов, над которыми выполняется операция (иерархия Node), а другую – для

посетителей, описывающих те операции, которые выполняются над элементами (иерархия NodeVisitor). Новая операция создается путем добавления подкласса в иерархию классов посетителей. До тех пор пока грамматика языка остается постоянной (то есть не добавляются новые подклассы Node), новую функциональность можно получить путем определения новых подклассов NodeVisitor.





А вообще подобная схема -- пример [двойной диспетчеризации](#).

Некоторые достоинства и недостатки паттерна посетитель:

- *упрощает добавление новых операций*. С помощью посетителей легко добавлять операции, зависящие от компонентов сложных объектов. Для определения новой операции над структурой объектов достаточно просто ввести нового посетителя. Напротив, если функциональность распределена по

нескольким классам, то для определения новой операции придется изменить каждый класс;

- *объединяет родственные операции и отсекает те, которые не имеют к ним отношения*. Родственное поведение не разносится по всем классам, присутствующим в структуре объектов, оно локализовано в посетителе. Не связанные друг с другом функции распределяются по отдельным подклассам класса Visitor. Это способствует упрощению как классов, определяющих элементы, так и алгоритмов, инкапсулированных в посетителях. Все относящиеся к алгоритму структуры данных можно скрыть в посетителе;
- *добавление новых классов ConcreteElement затруднено*. Паттерн посетитель усложняет добавление новых подклассов класса Element. Каждый новый конкретный элемент требует объявления новой абстрактной операции в классе Visitor, которую нужно реализовать в каждом из существующих классов ConcreteVisitor. Иногда большинство конкретных посетителей могут унаследовать операцию по умолчанию, предоставляемую классом Visitor, что скорее исключение, чем правило.

  Поэтому при решении вопроса о том, стоит ли использовать паттерн посетитель, нужно прежде всего посмотреть, что будет изменяться чаще: алгоритм, применяемый к объектам структуры, или классы объектов, составляющих эту структуру. Вполне вероятно, что сопровождать иерархию классов Visitor будет нелегко, если новые классы ConcreteElement добавляются часто. В таких случаях проще определить операции прямо в классах, представленных в структуре. Если же иерархия классов Element стабильна, но постоянно расширяется набор операций или модифицируются алгоритмы, то паттерн посетитель поможет лучше управлять такими изменениями;
- *аккумулирование состояния*. Посетители могут аккумулировать информацию о состоянии при посещении объектов структуры. Если не использовать этот паттерн, состояние придется передавать в виде дополнительных аргументов операций, выполняющих обход, или хранить в глобальных переменных;
- *нарушение инкапсуляции*. Применение посетителей подразумевает, что у класса ConcreteElement достаточно развитый интерфейс для того, чтобы посетители могли справиться со своей работой. Поэтому при использовании данного паттерна приходится предоставлять открытые операции для доступа к внутреннему состоянию элементов, что ставит под угрозу инкапсуляцию.

Используйте паттерн Посетитель, когда:

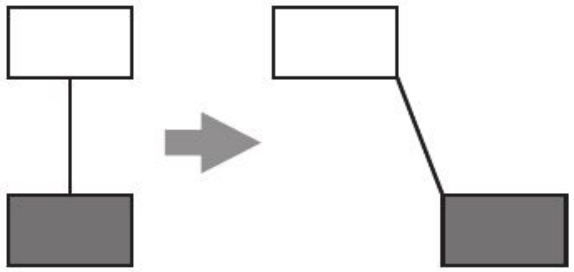
- в структуре присутствуют объекты многих классов с различными интерфейсами и вы хотите выполнять над ними операции, зависящие от конкретных классов;
- над объектами, входящими в состав структуры, надо выполнять разнообразные, не связанные между собой операции и вы не хотите «засорять» классы такими операциями. Посетитель позволяет объединить родственные операции, поместив их в один класс. Если структура объектов является общей для нескольких приложений, то паттерн посетитель позволит в каждое приложение включить только относящиеся к нему операции;
- классы, устанавливающие структуру объектов, изменяются редко, но новые операции над этой структурой добавляются часто. При изменении классов,

представленных в структуре, нужно будет переопределить интерфейсы всех посетителей, а это может вызвать затруднения. Поэтому если классы меняются достаточно часто, то, вероятно, лучше определить операции прямо в них.

Хранитель

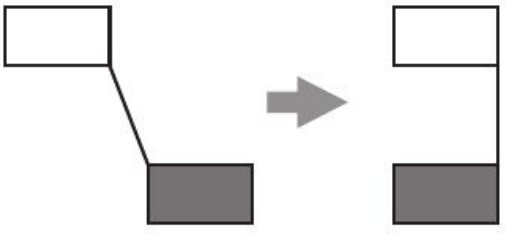
Иногда необходимо тем или иным способом зафиксировать внутреннее состояние объекта. Такая потребность возникает, например, при реализации контрольных точек и механизмов отката, позволяющих пользователю отменить пробную операцию или восстановить состояние после ошибки. Его необходимо где-то сохранить, чтобы позднее восстановить в нем объект. Но обычно объекты инкапсулируют все свое состояние или хотя бы его часть, делая его недоступным для других объектов, так что сохранить состояние извне невозможно. Раскрытие же состояния явилось бы нарушением принципа инкапсуляции и поставило бы под угрозу надежность и расширяемость приложения.

Рассмотрим, например, графический редактор, который поддерживает связность объектов. Пользователь может соединить два прямоугольника линией, и они останутся в таком положении при любых перемещениях. Редактор сам перерисовывает линию, сохраняя связность конфигурации.



Функции поддержания связности могут выполняться объектом класса ConstraintSolver, который регистрирует вновь создаваемые соединения и генерирует описывающие их математические уравнения. А когда пользователь каким-то образом модифицирует диаграмму, объект решает эти уравнения. Результаты вычислений объект ConstraintSolver использует для перерисовки графики так, чтобы были сохранены все соединения.

Поддержка отката операций в приложениях не так проста, как может показаться на первый взгляд. Очевидный способ откатить операцию перемещения – это сохранить расстояние между старым и новым положением, а затем переместить объект на такое же расстояние назад. Однако при этом не гарантируется, что все объекты окажутся там же, где находились. Предположим, что в способе расположения соединительной линии есть некоторая свобода. Тогда, переместив прямоугольник на прежнее место, мы можем не добиться желаемого эффекта.



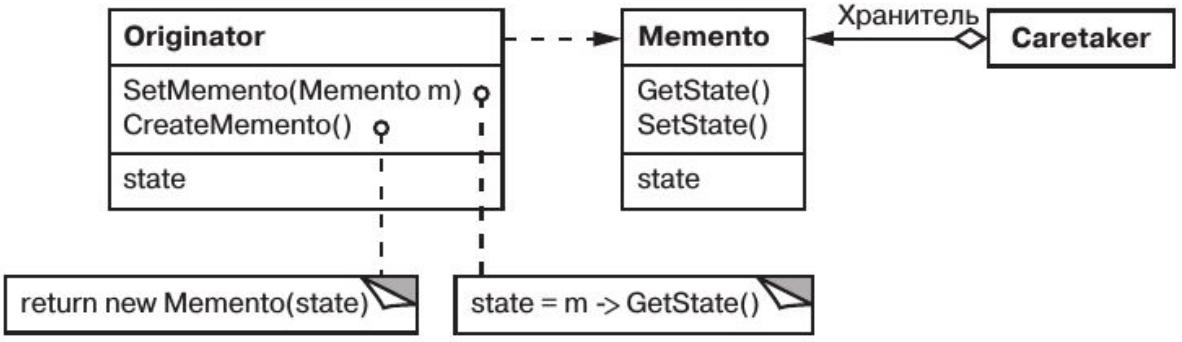
Открытого интерфейса ConstraintSolver иногда не хватает для точного отката всех изменений смежных объектов. Механизм отката должен работать в тесном взаимодействии с ConstraintSolver для восстановления предыдущего состояния, но необходимо также позаботиться о том, чтобы внутренние детали ConstraintSolver не были доступны этому механизму.

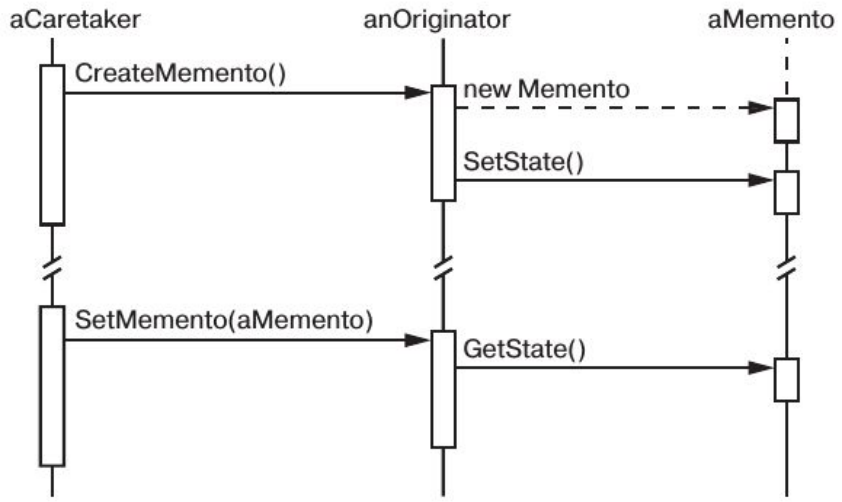
Паттерн Хранитель поможет решить данную проблему. Хранитель – это объект, в котором сохраняется внутреннее состояния другого объекта – хозяина хранителя. Для работы механизма отката нужно, чтобы хозяин предоставил хранитель, когда возникнет необходимость записать контрольную точку состояния хозяина. Только хозяину разрешено помещать в хранитель информацию и извлекать ее оттуда, для других объектов хранитель непрозрачен.

В примере графического редактора, который обсуждался выше, в роли хозяина может выступать объект ConstraintSolver. Процесс отката характеризуется такой последовательностью событий:

1. Редактор запрашивает хранитель у объекта ConstraintSolver в процессе выполнения операции перемещения.
2. ConstraintSolver создает и возвращает хранитель, в данном случае экземпляр класса SolverState. Хранитель SolverState содержит структуры данных, описывающие текущее состояние внутренних уравнений и переменных ConstraintSolver.
3. Позже, когда пользователь отменяет операцию перемещения, редактор возвращает SolverState объекту ConstraintSolver.
4. Основываясь на информации, которая хранится в объекте SolverState, ConstraintSolver изменяет свои внутренние структуры, возвращая уравнения и переменные в первоначальное состояние.

Такая организация позволяет объекту ConstraintSolver «знакомить» другие объекты с информацией, которая ему необходима для возврата в предыдущее состояние, не раскрывая в то же время свою структуру и представление.





Характерные особенности паттерна хранитель:

- *сохранение границ инкапсуляции*. Хранитель позволяет избежать раскрытия информации, которой должен распоряжаться только хозяин, но которую тем не менее необходимо хранить вне последнего. Этот паттерн экранирует объекты от потенциально сложного внутреннего устройства хозяина, не изменяя границы инкапсуляции;
- *упрощение структуры хозяина*. При других вариантах дизайна, направленного на сохранение границ инкапсуляции, хозяин хранит внутри себя версии внутреннего состояния, которое запрашивали клиенты. Таким образом, вся ответственность за управление памятью лежит на хозяине. При перекладывании заботы о запрошенном состоянии на клиентов упрощается структура хозяина, а клиентам дается возможность не информировать хозяина о том, что они закончили работу;
- *значительные издержки при использовании хранителей*. С хранителями могут быть связаны заметные издержки, если хозяин должен копировать большой объем информации для занесения в память хранителя или если клиенты создают и возвращают хранителей достаточно часто. Если плата за инкапсуляцию и восстановление состояния хозяина велика, то этот паттерн не всегда подходит;
- *скрытая плата за содержание хранителя*. Посыльный отвечает за удаление хранителя, однако не располагает информацией о том, какой объем информации о состоянии скрыт в нем. Поэтому нетребовательный к ресурсам посыльный может расходовать очень много памяти при работе с хранителем.

Используйте паттерн хранитель, когда:

- необходимо сохранить мгновенный снимок состояния объекта (или его части), чтобы впоследствии объект можно было восстановить в том же состоянии;
- прямое получение этого состояния раскрывает детали реализации и нарушает инкапсуляцию объекта.

Обсуждение поведенческих паттернов

Инкапсуляция вариаций – суть многих паттернов поведения. Если определенная часть программы подвержена периодическим изменениям, эти паттерны позволяют определить объект для инкапсуляции такого аспекта. Другие части программы, зависящие от данного аспекта, могут кооперироваться с ним. Обычно паттерны поведения определяют абстрактный класс, с помощью которого описывается инкапсулирующий объект. Своим названием паттерн как раз и обязан этому объекту. Например:

- объект-стратегия инкапсулирует алгоритм;
- объект-состояние инкапсулирует поведение, зависящее от состояния;
- объект-посредник инкапсулирует протокол общения между объектами.

Перечисленные паттерны описывают подверженные изменениям аспекты программы. В большинстве паттернов фигурируют два вида объектов: новый объект (или объекты), который инкапсулирует аспект, и существующий объект (или объекты), который пользуется новыми. Если бы не паттерн, то функциональность новых объектов пришлось бы делать неотъемлемой частью существующих. Например, код объекта-стратегии, вероятно, был бы «зашит» в контекст стратегии, а код объекта-состояния был бы реализован непосредственно в контексте состояния.

Но не все паттерны поведения разбивают функциональность таким образом. Например, паттерн цепочка обязанностей связан с произвольным числом объектов (то есть цепочкой), причем все они могут уже существовать в системе.

Цепочка обязанностей иллюстрирует еще одно различие между паттернами поведения: не все они определяют статические отношения взаимосвязи между классами. В частности, цепочка обязанностей показывает, как организовать обмен информацией между заранее неизвестным числом объектов. В других паттернах участвуют объекты, передаваемые в качестве аргументов.

Объекты как аргументы

В нескольких паттернах участвует объект, всегда используемый только как аргумент. Одним из них является Посетитель. Объект-посетитель – это аргумент полиморфной операции Accept, принадлежащей посещаемому объекту. Посетитель никогда не рассматривается как часть посещаемых объектов, хотя традиционным альтернативным вариантом этому паттерну служит распределение кода посетителя между классами объектов, входящих в структуру.

Другие паттерны определяют объекты, выступающие в роли “волшебных палочек”, которые передаются от одного владельца к другому и активизируются в будущем. К этой категории относятся Команда и Хранитель. В паттерне Команда такой “палочкой” является запрос, а в Хранителе она представляет внутреннее состояние объекта в определенный момент. И там, и там “палочка” может иметь сложную внутреннюю структуру, но клиент об этом ничего не знает. Но даже здесь есть различия. В паттерне Команда важную роль играет полиморфизм, поскольку выполнение объекта-команды – полиморфная операция. Напротив, интерфейс хранителя настолько узок, что его можно передавать лишь как значение. Поэтому

вполне вероятно, что хранитель не предоставляет полиморфных операций своим клиентам.

Должен ли обмен информацией быть инкапсулированным или распределенным

Паттерны Посредник и Наблюдатель конкурируют между собой. Различие между ними в том, что Наблюдатель распределяет обмен информацией за счет объектов наблюдатель и субъект, а Посредник, наоборот, инкапсулирует взаимодействие между другими объектами.

В паттерне Наблюдатель участники наблюдатель и субъект должны кооперироваться, чтобы поддержать ограничение. Паттерны обмена информацией определяются тем, как связаны между собой наблюдатели и субъекты; у одного субъекта обычно бывает много наблюдателей, а иногда наблюдатель субъекта сам является субъектом наблюдения со стороны другого объекта. В паттерне Посредник ответственность за поддержание ограничения возлагается исключительно на посредника.

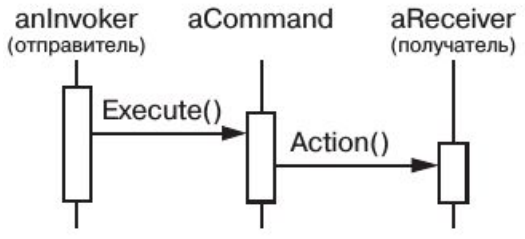
Кажется, что повторно использовать наблюдатели и субъекты проще, чем посредники. Паттерн наблюдатель способствует разделению и ослаблению связей между наблюдателем и субъектом, что приводит к появлению сравнительно мелких классов, более приспособленных для повторного использования.

С другой стороны, потоки информации в Посреднике проще для понимания, нежели в Наблюдателе. Наблюдатели и субъекты обычно связываются вскоре после создания, и понять, каким же образом организована их связь, в последующих частях программы довольно трудно. Если вы знаете паттерн Наблюдатель, то понимаете важность того, как именно связаны наблюдатели и субъекты, и представляете, какие связи надо искать. Однако из-за присущей Наблюдателю косвенности разобраться в системе все же нелегко.

Разделение получателей и отправителей

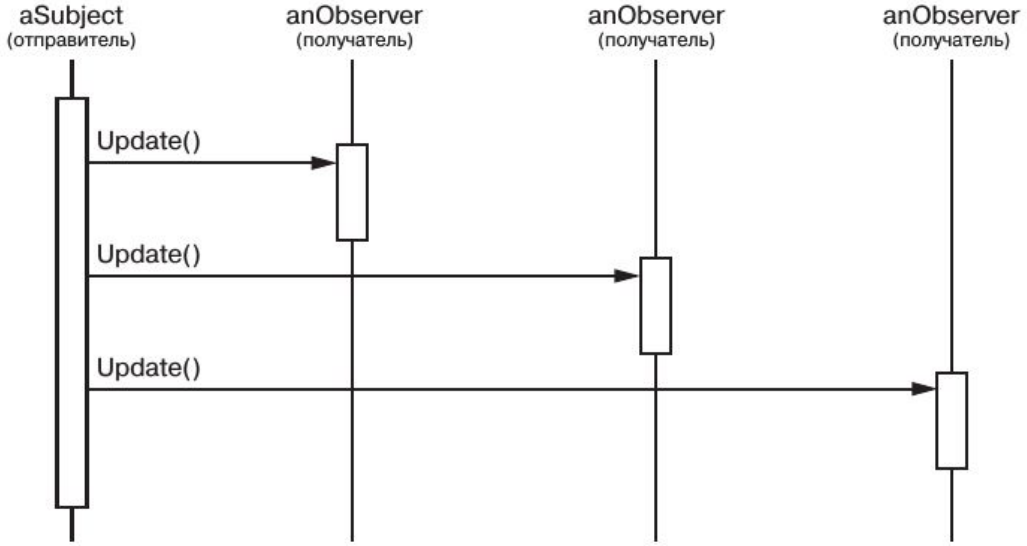
Когда взаимодействующие объекты напрямую ссылаются друг на друга, они становятся зависимыми, а это может отрицательно сказаться на повторном использовании системы и разбиении ее на уровни. Паттерны Команда, Наблюдатель, Посредник и Цепочка обязанностей указывают разные способы разделения получателей и отправителей запросов. Каждый способ имеет свои достоинства и недостатки.

Паттерн Команда поддерживает разделение за счет объекта-команды, который определяет привязку отправителя к получателю.

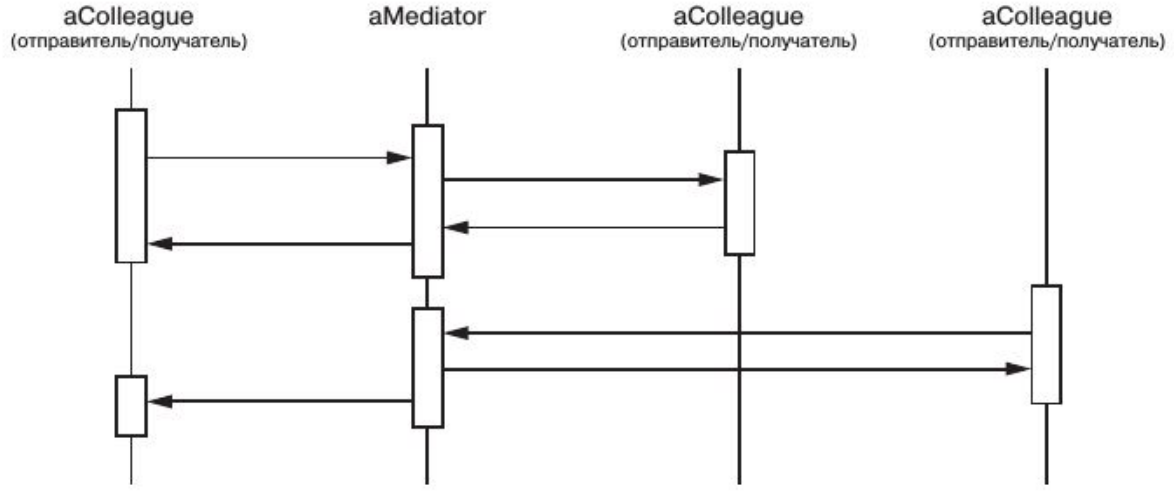


Паттерн Команда предоставляет простой интерфейс для выдачи запроса (операцию Execute). Заключение связи между отправителем и получателем в самостоятельный объект позволяет отправителю работать с разными получателями. Он отделяет отправителя от получателей, облегчая тем самым повторное использование. Кроме того, объект-команду можно повторно использовать для параметризации получателя различными отправителями.

Паттерн Наблюдатель отделяет отправителей (субъектов) от получателей (наблюдателей) путем определения интерфейса для извещения о происшедших с субъектом изменениях. По сравнению с Командой в Наблюдателе связь между отправителем и получателем слабее, поскольку у субъекта может быть много наблюдателей и их число даже может меняться во время выполнения. Интерфейсы субъекта и наблюдателя в паттерне Наблюдатель предназначены для передачи информации об изменениях. Стало быть, этот паттерн лучше всего подходит для разделения объектов в случае, когда между ними есть зависимость по данным.



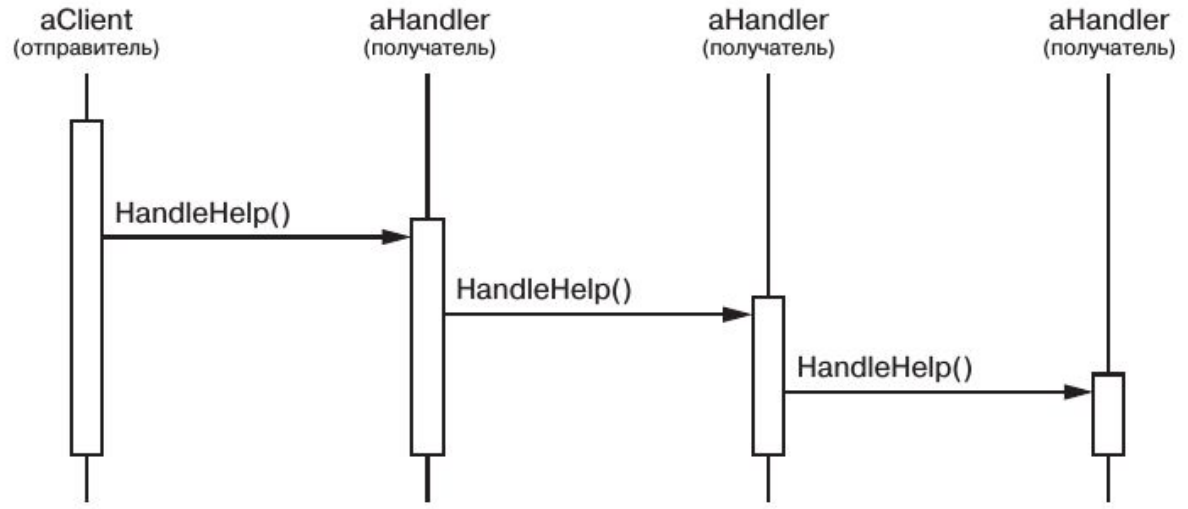
Паттерн Посредник разделяет объекты, заставляя их ссылаться друг на друга косвенно, через объект-посредник.



Объект-посредник распределяет запросы между объектами-коллегами и централизует обмен информацией между ними. Таким образом, коллеги могут общаться между собой только с помощью интерфейса посредника. Поскольку этот интерфейс фиксирован, посредник может реализовать собственную схему диспетчеризации для большей гибкости. Разрешается кодировать запросы и упаковывать аргументы так, что коллеги смогут запрашивать выполнение операций из заранее неизвестного множества.

Паттерн Посредник часто способствует уменьшению числа подклассов в системе, поскольку централизует весь обмен информацией в одном классе, вместо того чтобы распределять его по подклассам.

Наконец, паттерн Цепочка обязанностей отделяет отправителя от получателя за счет передачи запроса по цепочке потенциальных получателей.



Поскольку интерфейс между отправителями и получателями фиксирован, то Цепочка обязанностей также может нуждаться в специализированной схеме диспетчеризации. Поэтому она обладает теми же недостатками с точки зрения безопасности типов, что и посредник. Цепочка обязанностей – это хороший способ разделить отправителя и получателя в случае, если она уже является частью структуры системы, а один объект из группы может принять на себя обязанность обработать запрос. Данный паттерн повышает гибкость и за счет того, что цепочку можно легко изменить или расширить.

Резюме

Как правило, паттерны поведения дополняют и усиливают друг друга. Например, класс в Цепочке обязанностей, скорее всего, будет содержать хотя бы один Шаблонный метод. Он может пользоваться примитивными операциями, чтобы определить, должен ли объект обработать запрос сам, а также в случае необходимости выбрать объект, которому следует переадресовать запрос. Цепочка может применять паттерн Команда для представления запросов в виде объектов.

Паттерны поведения хорошо сочетаются и с другими паттернами. Например, система, в которой применяется паттерн Компоновщик, время от времени использует Посетитель для выполнения операций над компонентами, а также задействует

Цепочку обязанностей, чтобы обеспечить компонентам доступ к глобальным свойствам через их родителя. Бывает, что в системе применяется и паттерн Декоратор для переопределения некоторых свойств частей композиции. А паттерн Наблюдатель может связать структуры разных объектов, тогда как паттерн Состояние позволит компонентам варьировать свое поведение при изменении состояния. Сама композиция, например, может быть создана с применением Строителя.

Хорошо продуманные объектно-ориентированные системы внешне похожи на собрание многочисленных паттернов, но вовсе не потому, что их проектировщики мыслили именно такими категориями. Композиция на уровне паттернов, а не классов или объектов, позволяет добиться той же синергии, но с меньшими усилиями.