Известные применения

Пример виртуального заместителя из раздела «Мотивация» заимствован из классов строительного блока текста, определенных в каркасе ET++.

В системе NEXTSTEP [Add94] заместители (экземпляры класса `NXProxy`) используются как локальные представители объектов, которые могут быть распределенными. Сервер создает заместителей для удаленных объектов, когда клиент их запрашивает. Заместитель кодирует полученное сообщение вместе со всеми аргументами, после чего отправляет его удаленному субъекту. Аналогично субъект кодирует возвращенные результаты и посылает их обратно объекту `NXProxy`.

В работе McCullough [McC87] обсуждается применение заместителей в Smalltalk для доступа к удаленным объектам. Джефри Пэско (Geoffrey Pascoe) [Pas86] описывает, как обеспечить побочные эффекты при вызове методов и реализовать контроль доступа с помощью «инкапсуляторов».

Родственные паттерны

Паттерн адаптер предоставляет другой интерфейс к адаптируемому объекту. Напротив, заместитель в точности повторяет интерфейс своего субъекта. Однако, если заместитель используется для ограничения доступа, он может отказаться выполнять операцию, которую субъект выполнил бы, поэтому на самом деле интерфейс заместителя может быть и подмножеством интерфейса субъекта.

Несколько замечаний относительно декоратора. Хотя его реализация и похожа на реализацию заместителя, но назначение совершенно иное. Декоратор добавляет объекту новые обязанности, а заместитель контролирует доступ к объекту.

Степень схожести реализации заместителей и декораторов может быть различной. Защищающий заместитель мог бы быть реализован в точности как декоратор. С другой стороны, удаленный заместитель не содержит прямых ссылок на реальный субъект, а лишь косвенную ссылку, что-то вроде «идентификатор хоста и локальный адрес на этом хосте». Вначале виртуальный заместитель имеет только косвенную ссылку (скажем, имя файла), но в конечном итоге получает и использует прямую ссылку.

Обсуждение структурных паттернов

Возможно, вы обратили внимание на то, что структурные паттерны похожи между собой, особенно когда речь идет об их участниках и взаимодействиях. Вероятное объяснение такому явлению: все структурные паттерны основаны на небольшом множестве языковых механизмов структурирования кода и объектов (одиночном и множественном наследовании для паттернов уровня класса и композиции для паттернов уровня объектов). Но имеющееся сходство может быть обманчиво, ибо с помощью разных паттернов можно решать совершенно разные задачи. В этом разделе сопоставлены группы структурных паттернов, и вы сможете яснее почувствовать их сравнительные достоинства и недостатки.

Адаптер и мост

У паттернов адаптер и мост есть несколько общих атрибутов. Тот и другой повышают гибкость, вводя дополнительный уровень косвенности при обращении

к другому объекту. Оба перенаправляют запросы другому объекту, используя иной интерфейс.

Основное различие между адаптером и мостом в их назначении. Цель первого – устранить несовместимость между двумя существующими интерфейсами. При разработке адаптера не учитывается, как эти интерфейсы реализованы и то, как они могут независимо развиваться в будущем. Он должен лишь обеспечить совместную работу двух независимо разработанных классов, так чтобы ни один из них не пришлось переделывать. С другой стороны, мост связывает абстракцию с ее, возможно, многочисленными реализациями. Данный паттерн предоставляет клиентам стабильный интерфейс, позволяя в то же время изменять классы, которые его реализуют. Мост также подстраивается под новые реализации, появляющиеся в процессе развития системы.

В связи с описанными различиями адаптер и мост часто используются в разные моменты жизненного цикла системы. Когда выясняется, что два несовместимых класса должны работать вместе, следует обратиться к адаптеру. Тем самым удастся избежать дублирования кода. Заранее такую ситуацию предвидеть нельзя. Наоборот, пользователь моста с самого начала понимает, что у абстракции может быть несколько реализаций и развитие того и другого будет идти независимо. Адаптер обеспечивает работу *после* того, как нечто спроектировано; мост – *до* того. Это доказывает, что адаптер и мост предназначены для решения именно своих задач.

Фасад можно представлять себе как адаптер к набору других объектов. Но при такой интерпретации легко не заметить такой нюанс: фасад определяет *новый* интерфейс, тогда как адаптер повторно использует уже имеющийся. Подчеркнем, что адаптер заставляет работать вместе два *существующих* интерфейса, а не определяет новый.

Компоновщик, декоратор и заместитель

У компоновщика и декоратора аналогичные структурные диаграммы, свидетельствующие о том, что оба паттерна основаны на рекурсивной композиции и предназначены для организации заранее неопределенного числа объектов. При обнаружении данного сходства может возникнуть искушение посчитать объект-декоратор вырожденным случаем компоновщика, но при этом будет искажен сам смысл паттерна декоратор. Сходство и заканчивается на рекурсивной композиции, и снова из-за различия задач, решаемых с помощью паттернов.

Назначение декоратора – добавить новые обязанности объекта без порождения подклассов. Этот паттерн позволяет избежать комбинаторного роста числа подклассов, если проектировщик пытается статически определить все возможные комбинации. У компоновщика другие задачи. Он должен так структурировать классы, чтобы различные взаимосвязанные объекты удавалось трактовать единообразно, а несколько объектов рассматривать как один. Акцент здесь делается не на оформлении, а на представлении.

Указанные цели различны, но дополняют друг друга. Поэтому компоновщик и декоратор часто используются совместно. Оба паттерна позволяют спроектировать систему так, что приложения можно будет создавать, просто соединяя