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

объекты между собой, без определения новых классов. Появится некий абстрактный класс, одни подклассы которого – компоновщики, другие – декораторы, а третьи – реализации фундаментальных строительных блоков системы. В таком случае у компоновщиков и декораторов будет общий интерфейс. Для декоратора компоновщик является конкретным компонентом. А для компоновщика декоратор – это листовый узел. Разумеется, их необязательно использовать вместе, и, как мы видели, цели данных паттернов различны.

Заместитель – еще один паттерн, структура которого напоминает декоратор. Оба они описывают, как можно предоставить косвенный доступ к объекту, и в реализации объектов-декораторов и заместителей хранится ссылка на другой объект, которому переадресуются запросы. Но и здесь цели различаются.

Как и декоратор, заместитель предоставляет клиенту интерфейс, совпадающий с интерфейсом замещаемого объекта. Но в отличие от декоратора заместителю не нужно динамически добавлять и отбирать свойства, он не предназначен для рекурсивной композиции. Заместитель должен предоставить стандартную замену субъекту, когда прямой доступ к нему неудобен или нежелателен, например потому, что он находится на удаленной машине, хранится на диске или доступен лишь ограниченному кругу клиентов.

В паттерне заместитель субъект определяет основную функциональность, а заместитель разрешает или запрещает доступ к ней. В декораторе компонент обладает лишь частью функциональности, а остальное привносят один или несколько декораторов. Декоратор позволяет справиться с ситуацией, когда полную функциональность объекта нельзя определить на этапе компиляции или это по тем или иным причинам неудобно. Такая неопределенность делает рекурсивную композицию неотъемлемой частью декоратора. Для заместителя дело обстоит не так, ибо ему важно лишь одно отношение – между собой и своим субъектом, а данное отношение можно выразить статически.

Указанные различия существенны, поскольку в них абстрагированы решения конкретных проблем, снова и снова возникающих при объектно-ориентированном проектировании. Но это не означает, что сочетание разных паттернов невозможно. Можно представить себе заместителя-декоратора, который добавляет новую функциональность заместителю, или декоратора-заместителя, который оформляет удаленный объект. Такие гибриды теоретически *могут* быть полезны (у нас, правда, не нашлось реального примера), а вот паттерны, из которых они составлены, полезны наверняка.