# Часть 3. Структурные паттерны

Любая грамотно спроектированная программная система является иерархической. В корне приложения, например, в методе Main, создается набор высокоуровневых компонентов, каждый из которых опирается на компоненты более низкого уровня. Эти компоненты, в свою очередь, также могут быть разбиты на более простые составляющие.

Существует набор типовых решений, которые помогают бороться со сложностью, создавать и развивать современные системы. Так, например, довольно часто возникает потребность в полиморфном использовании классов, которые выполняют схожую задачу, но не обладают единым интерфейсом. Такая ситуация возможна, когда классы разрабатывались в разное время, разными людьми, а иногда, и разными организациями. Связать такие классы вместе позволит паттерн *Адаптер*. Адаптер дает возможность "подстроить" разные реализации к одному интерфейсу, и использовать их полиморфным образом, даже тогда, когда часть существующего кода находится вне вашего контроля.

Еще одним распространенным структурным паттерном является *Фасад*. Фасад представляет высокоуровневый интерфейс к сторонней библиотеке или модулю, что упрощает код клиентов и делает их менее зависимыми от стороннего кода. Это позволяет существенно упростить логику приложения, и делает ее менее зависимой от ошибок и изменений сторонних библиотек. Фасад упрощает миграцию на новую версию библиотеки, поскольку проверять придется не весь код приложения, а лишь запустить тесты фасада и убедиться, что поведение осталось неизменным.

Паттерн *Декоратор* позволяет "нанизывать" дополнительные аспекты поведения один на другой, без создания чрезмерного числа производных классов. Декоратор прекрасно справляется с задачами кэширования, логирования или c ограничением числа вызовов. При этом за каждый аспект поведения будет отвечать выделенный класс, что упростит понимание и развитие системы.

Многим приложениям приходится работать с иерархическими данными. При этом очень часто возникает потребность скрыть иерархическую природу или упростить с ней работу таким образом, чтобы приложение работало с составными и одиночными объектами унифицированным образом. Паттерн *Компоновщик* предназначен для решения именно этой задачи: он позволяет объединить простые элементы в составные объекты и дает возможность клиентам работать с такими структурами единообразным способом.

Современные системы невозможно представить без удаленного взаимодействия. И в этой области паттерн *Прокси* уже давно стал классическим. Никто уже не может себе представить, что для взаимодействия с удаленным сервером, код приложения будет создавать TCP соединение, сериализировать аргументы и анализировать результаты (\*). Все современные библиотеки, как WCF или .NET Remoting, используют прокси-классы, которые сглаживают грань между внутрипроцессным и межпроцессным взаимодействием.

(\*) Сноска: это не значит, что приложение не может использовать самописные протоколы сетевого взаимодействия. Но даже если вам не подошел WCF и вы решили работать с TCP соединениями напрямую, то все равно логично спрятать низкоуровневые подробности в своем самодельном прокси-классе, и не «размазывать» эту логику по коду приложения.