Компоновщик (Composite)

Его задача – агрегировать объекты, обладающие общим интерфейсом, и обеспечить выполнение операций этого интерфейса над всеми агрегируемыми объектами.
Дизайн шаблона Компоновщика показан на рис. 6. То, что и компоновщик, и отдельный объект реализуют один и тот же интерфейс, удобно с точки зрения клиента, который не видит разницы между ними в плане функциональности. Обычно компоновщика снабжают методами добавления, удаления, редактирования и поиска отдельных агрегируемых компонент.

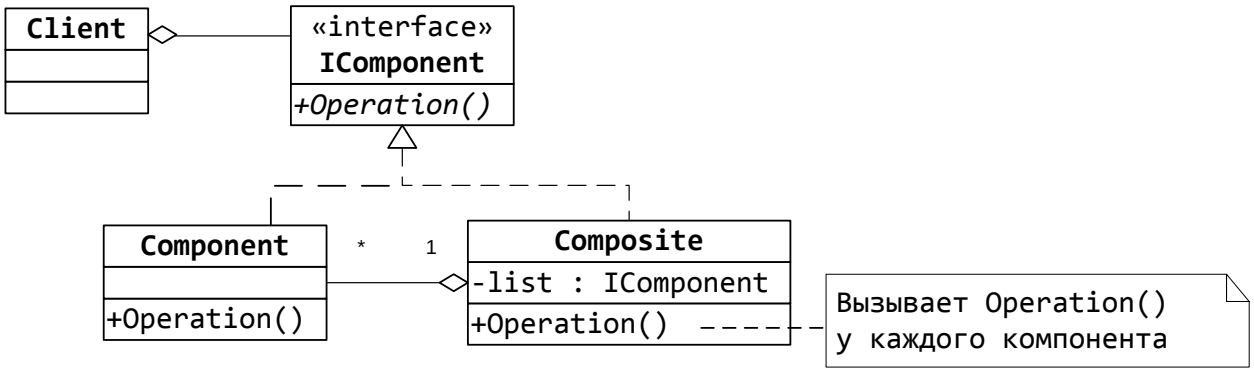

Рис. 6. Дизайн шаблона Компоновщик.

Приспособленец (Flyweight)

Шаблон Приспособленец предлагает эффективный способ разделить общую информацию, находящуюся в небольших объектах. Суть шаблона в том, чтобы разделить состояние некоторого объекта на состояния трех типов. *Внутреннее состояние* (intrinsic state) принадлежит самому объекту. Тип `Flyweight` реализует интерфейс `IFlyweight`, который определяет операции, в которых заинтересована остальная часть системы. Клиент владеет *общим* (неразделяемым) *состоянием* (unshared state), а также коллекцией приспособленцев, которых производит класс-фабрика (`FlyweightFactory`). Наконец, *внешнее состояние* (extrinsic state) не появляется в системе как таковое. Если оно понадобится, то будет вычислено уже во время выполнения программы для каждого внутреннего состояния.

Адаптер (Adapter)

Шаблон Адаптер предназначен для обеспечения совместной работы классов, которые изначально не были предназначены для совместного использования. Такие ситуации часто возникают, когда идет работа с существующими библиотеками кода. Нередко их интерфейс не отвечает требованиям клиента, но изменить библиотеку возможности нет. Возникает задача адаптации библиотеки для клиента.

Дизайн шаблона Адаптер показан на рис. 8.

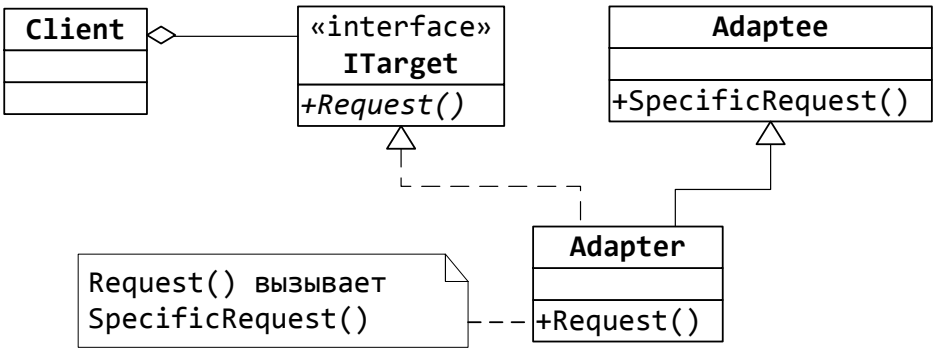

Рис. 8. Дизайн шаблона Адаптер.

В адаптере четко видно преимущество программирования согласно интерфейсам. `Client` работает в соответствии с требованиями своей предметной области, эти требования отражены в целевом интерфейсе `ITarget` (интерфейс, в котором заинтересован клиент). Адаптируемый класс `Adaptee` обладает требуемой функциональностью, но неподходящим интерфейсом. `Adapter` реализует интерфейс `ITarget` и перенаправляет вызовы от `Client` к `Adaptee`, изменяя при необходимости параметры и возвращаемые значения.

Особенность адаптеров в том, что они могут добавлять дополнительное поведение к тому поведению, что специфицируется в `ITarget` и в `Adaptee`. Другими словами, адаптеры могут быть *прозрачными* для клиента и *непрозрачными*. В примере кода показан последний случай, где `Adapter` добавляет `"Rough:"`. Эта добавка показывает, что вызов `Request()` был адаптирован (изменен) перед тем, как был вызван метод `SpecificRequest()`.

Фасад (Façade)

Назначение шаблона Фасад состоит в предоставлении различных высокоуровневых представлений подсистем, детали реализации которых скрыты от клиента. В общем случае набор операций, желаемый для клиента, может формироваться в виде набора из разных частей подсистемы. Сокрытие деталей – это ключевая концепция программирования.

Из возможных вариантов отметим *прозрачные фасады* (в этом случае компоненты подсистемы могут быть доступны и через фасад, и в обход его) и *статические фасады* (фасад является статическим классом – скрываемые объекты не агрегируются, а создаются в методах фасада по необходимости).

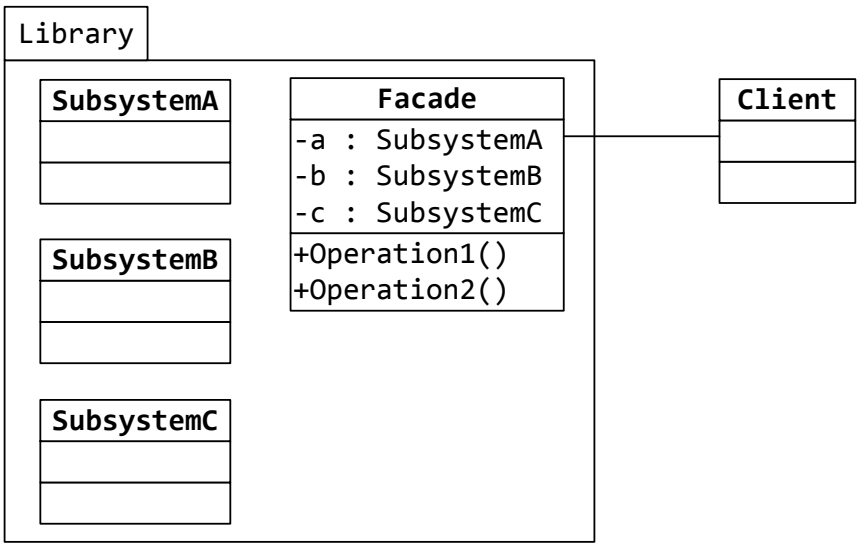

Рис. 9. Дизайн шаблона проектирования Фасад.