По поводу полезности шаблонов/паттернов проектирования (design patterns) есть разные мнения. Пол Грэхэм, к примеру, считает, что сама идея шаблонов проектирования — антипаттерн, сигнал о том, что система не обладает достаточным уровнем абстракции, и необходима её тщательная переработка. Более традиционное мнение заключается в том, что всё же польза от них есть. Шаблон проектирования — это повторимая архитектурная конструкция, являющаяся решением некоторой типичной технической проблемы. По аналогии со стилями и архитектурными шаблонами обычно паттерны проектирования не являются законченным готовым кодом, это лишь некий пример или подход для решения задачи, который может быть применен в различных ситуациях и контекстах. По сути объектно-ориентированный шаблон показывает отношения и взаимодействия между классами или объектами практически независимо от того, что это за классы и как они будут потом использоваться.

Главная польза каждого отдельного шаблона состоит в том, что он описывает решение целого класса абстрактных проблем. Решение становится более гибким и переиспользуемым. Также тот факт, что каждый шаблон имеет свое имя, облегчает дискуссию между разработчиками, так как они могут ссылаться на известные шаблоны. Таким образом, за счёт шаблонов производится унификация терминологии, названий модулей и элементов проекта. К тому же, наличие паттернов чаще всего делает архитектуру более наглядной и простой в изучении.

Однако, все помним про то, что такое карго-культ программирование. Шаблоны не стоит рассматривать как лекарство от всех проблем, слепое применение шаблонов по книжке, без осмысления причин и предпосылок выделения каждого отдельного шаблона, замедляет профессиональный рост программиста, так как подменяет творческую работу механическим подставлением шаблонов. Хороший критерий нужной степени профессионализма — выделение шаблонов самостоятельно, на основании собственного опыта. К тому же слепое следование некоторому выбранному шаблону может привести к усложнению программы. Именно поэтому многие разработчики крайне скептически настроены к популярности шаблонов проектирования.

Проектирование текстового редактора

Чтобы не скатиться в пропасть карго-культа, лучше всего разбирать шаблоны на примерах. Причём идеально, если к этому моменту уже есть собственный опыт решения подобной задачи, и тогда информация о шаблоне лучше всего усваивается. Если собственного опыта нет, то основная рекомендация -- понять, в каких ситуациях какой шаблон применим, и какие проблемы он помогает решить. В таком случае когда придёт время, вы услышите знакомые слова и вспомните, что когда-то что-то такое читали/слышали.

Одна из самых известных книг по шаблонам проектирования -- книга Гаммы и компании. Они собрали в ней 23 шаблона и постарались дать им подробные описания по целому ряду критериев. Разумеется, шаблонов гораздо больше, но эти уже стали некоего рода классикой, и считается, что более-менее адекватно про них рассказать должен каждый грамотный специалист (даже тот, который шаблоны на дух не переносит). Использовать шаблоны в реальных проектах или нет -- дело каждого, но проанализировать решения, которые лежат в их основе, полезно всем.

Шаблоны принято делить на структурные, поведенческие и порождающие. Мы сегодня разберём группу структурных шаблонов, и начнём с примера, на котором можно будет подробно проследить рождение пары шаблонов в реальном проекте. Будем проектировать текстовый WYSIWYG-редактор. В документах могут произвольным образом текст и графика, отформатированные разными способами. Вокруг документа — привычные выпадающие меню, панели инструментов, полосы прокрутки и прочие привычные современному пользователю элементы интерфейса.

Сразу же на ум приходит ряд задач, характерных для устройства и работы подобного вида приложений:

- *структура документа*. Выбор внутреннего представления документа отражается практически на всех аспектах дизайна. Для редактирования, форматирования, отображения и анализа текста необходимо уметь обходить это представление. Способ организации информации играет решающую роль при дальнейшем проектировании;
- *форматирование*. Как в подобных редакторах будут организованы текст и графика в виде строк и колонок? Какие объекты отвечают за реализацию разных видов форматирования? Как будет организовано взаимодействие данных алгоритмов с внутренним представлением документа?
- *создание привлекательного интерфейса пользователя.* В состав пользовательского интерфейса редактора входят полосы прокрутки, рамки, выпадающие меню и многое другое. Вполне вероятно, что количество и состав элементов интерфейса будут изменяться по мере его развития или по желанию пользователя. Поэтому важно иметь возможность легко добавлять и удалять элементы оформления, не затрагивая приложение;
- *поддержка стандартов внешнего облика программы*. Редактор должен без серьезной модификации адаптироваться к стандартам или стилям внешнего облика программ;
- *поддержка оконных систем*. В различных оконных системах стандарты внешнего облика обычно различаются. По возможности дизайн текстового редактора должен быть независимым от оконной системы;
- *операции пользователя*. Пользователи управляют работой редактора с помощью элементов интерфейса, в том числе кнопок и выпадающих меню. Функции, которые вызываются из интерфейса, разбросаны по всей программе. Разработать единообразный механизм для доступа к таким «рассеянным» функциям и для отмены уже выполненных операций довольно трудно;
- *проверка правописания и расстановка переносов*. Поддержка в редакторе таких аналитических операций, как проверка правописания и определение мест переноса. Как минимизировать число классов, которые придется модифицировать при добавлении новой аналитической операции?

Далее рассмотрим некоторые эти задачи и возможные подходы к их решению.

Структура документа

Документ — это организованное некоторым способом множество базовых графических элементов: символов, линий, многоугольников и других геометрических

фигур. Все они несут в себе полную информацию о содержании документа. И все же автор часто представляет себе эти элементы не в графическом виде, а в терминах физической структуры документа — строк, колонок, рисунков, таблиц и других подструктур. Эти подструктуры, в свою очередь, составлены из более мелких и т.д. Пользовательский интерфейс должен позволять пользователям непосредственно манипулировать такими подструктурами. Внутреннее представление должно поддерживать:

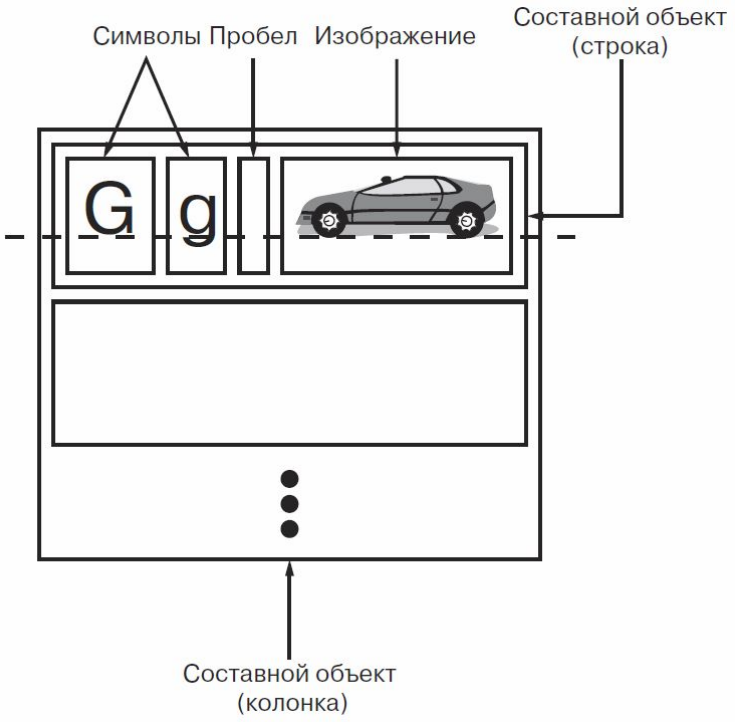
- отслеживание физической структуры документа, то есть разбиение текста и графики на строки, колонки, таблицы и т.д.;
- генерирование визуального представления документа;
- отображение позиций экрана на элементы внутреннего представления. Это позволит определить, что имел в виду пользователь, когда указал на что-то в визуальном представлении.

Помимо данных целей имеются и ограничения. Во-первых, текст и графику следует трактовать единообразно. Интерфейс приложения должен позволять свободно помещать текст внутрь графики и наоборот. Не следует считать графику частным случаем текста или текст — частным случаем графики, поскольку это в конечном итоге привело бы к появлению избыточных механизмов форматирования и манипулирования. Одного набора механизмов должно хватать и для текста, и для графики. Во-вторых, в нашей реализации не может быть различий во внутреннем представлении отдельного элемента и группы элементов. При одинаковой работе с простыми и сложными элементами можно будет создавать документы со структурой любой сложности. Например, десятым элементом на пересечении пятой строки и второй колонки мог бы быть как один символ, так и сложно устроенная диаграмма со многими внутренними компонентами. Но, коль скоро мы уверены, что этот элемент имеет возможность изображать себя на экране и сообщать свои размеры, его внутренняя сложность не имеет никакого отношения к тому, как и в каком месте страницы он появляется.

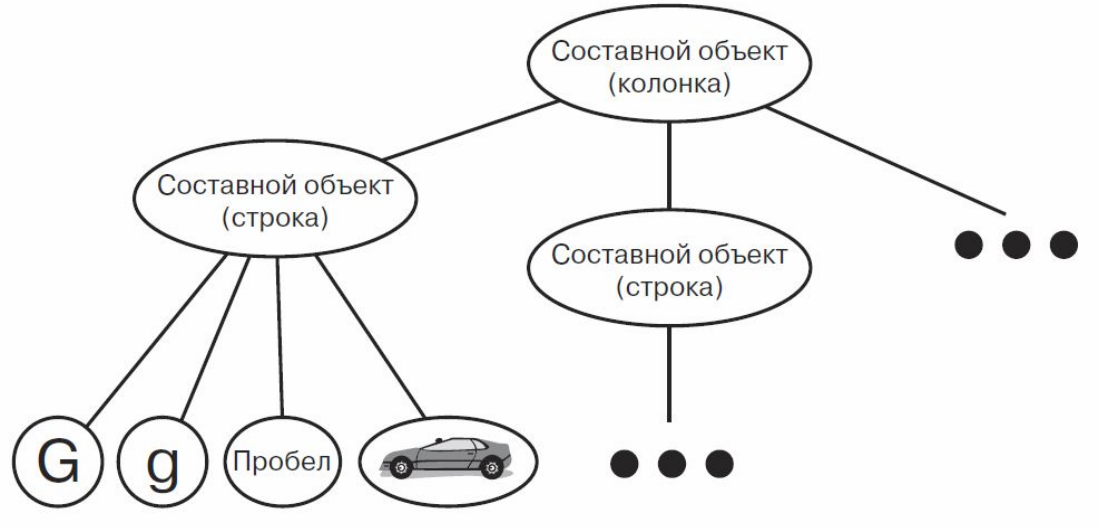
Однако, второе ограничение противоречит необходимости анализировать текст на предмет выявления орфографических ошибок и расстановки переносов. Во многих случаях нам безразлично, является ли элемент строки простым или сложным объектом. Но иногда вид анализа зависит от анализируемого объекта. Так, вряд ли имеет смысл проверять орфографию многоугольника или пытаться переносить его с одной строки на другую. При проектировании внутреннего представления надо учитывать эти и другие потенциально конфликтующие ограничения.

Рекурсивная композиция

На практике для представления иерархически структурированной информации часто применяется прием, называемый рекурсивной композицией. Он позволяет строить все более сложные элементы из простых. Рекурсивная композиция дает нам способ составить документ из простых графических элементов. Сначала мы можем линейно расположить множество символов и графики слева направо для формирования одной строки документа. Затем несколько строк можно объединить в колонку, несколько колонок — в страницу и т.д.



Данную физическую структуру можно представить, введя отдельный объект для каждого существенного элемента. К таковым относятся не только видимые элементы вроде символов и графики, но и структурные элементы — строки и колонки. В результате получается структура объекта, изображенная на следующем рисунке:



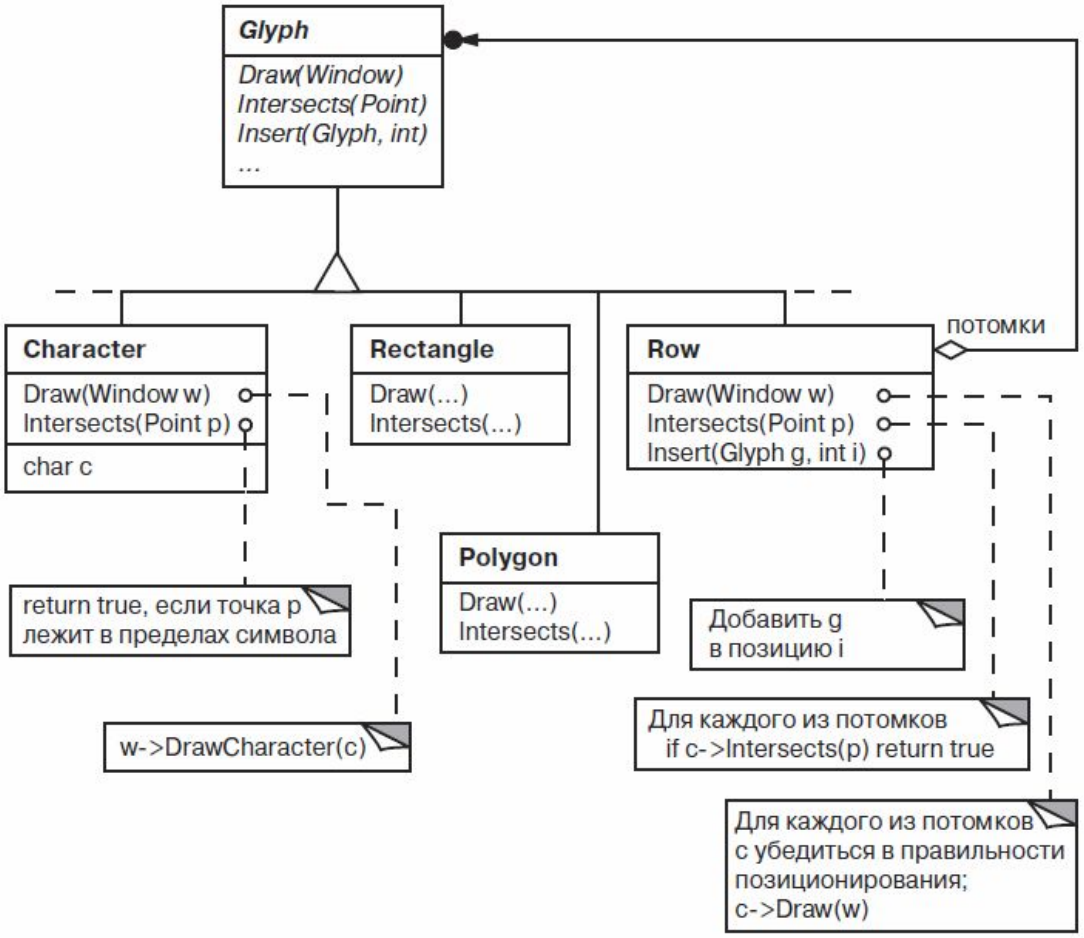
Представляя объектом каждый символ и графический элемент документа, мы обеспечиваем гибкость на самых нижних уровнях архитектуры. С точки зрения отображения, форматирования и вкладывания друг в друга единообразно трактуются текст и графика. Мы сможем расширить редактор для поддержки новых наборов символов, не затрагивая никаких других функций. Объектная структура точно отражает физическую структуру документа.

У описанного подхода есть два важных следствия. Первое очевидно: для объектов нужны соответствующие классы. Второе, менее очевидное, состоит в том, что у этих классов должны быть совместимые интерфейсы, поскольку мы хотим унифицировать

работу с ними. Для обеспечения совместимости интерфейсов в таких языках, как Java или C++, применяется наследование.

Глифы

Абстрактный класс Glyph (глиф) определяется для всех объектов, которые могут присутствовать в структуре документа. Его подклассы определяют как примитивные графические элементы (скажем, символы и изображения), так и структурные элементы (строки и колонки).



У глифов есть три основные функции. Они имеют информацию о своих предках и потомках, а также о том, как нарисовать себя на экране и сколько места они занимают. Подклассы класса Glyph переопределяют операцию Draw, выполняющую перерисовку себя в окне. При вызове Draw ей передается ссылка на объект Window.

Глифу-родителю часто бывает нужно «знать», сколько места на экране занимает глиф-потомок, чтобы расположить его и остальные глифы в строке без перекрытий. Операция Bounds возвращает прямоугольную область, занимаемую глифом, точнее, противоположные углы наименьшего прямоугольника, содержащего глиф. В подклассах класса Glyph эта операция переопределена в соответствии с природой конкретного элемента.

Операция Intersects возвращает признак, показывающий, лежит ли заданная точка в пределах глифа. Всякий раз, когда пользователь щелкает мышью где-то в документе, редактор вызывает эту операцию, чтобы определить, какой глиф или

глифовая структура оказалась под курсором мыши. Класс Rectangle переопределяет эту операцию для вычисления пересечения точки с прямоугольником. Поскольку у глифов могут быть потомки, то нам необходим единый интерфейс для добавления, удаления и обхода потомков. Например, потомки класса Row -- это глифы, расположенные в данной строке. Операция Insert вставляет глиф в позицию, заданную целочисленным индексом. Операция Remove удаляет заданный глиф, если он действительно является потомком.

Операция Child возвращает потомка с заданным индексом (если таковой существует). Глифы типа Row, у которых действительно есть потомки, должны пользоваться операцией Child, а не обращаться к структуре данных потомка напрямую. В таком случае при изменении структуры данных, скажем, с массива на связанный список не придется модифицировать операции вроде Draw, которые обходят всех потомков. Аналогично операция Parent предоставляет стандартный интерфейс для доступа к родителю глифа, если таковой имеется. Все глифы хранят ссылку на своего родителя, а Parent просто возвращает эту ссылку.

Паттерн Компоновщик

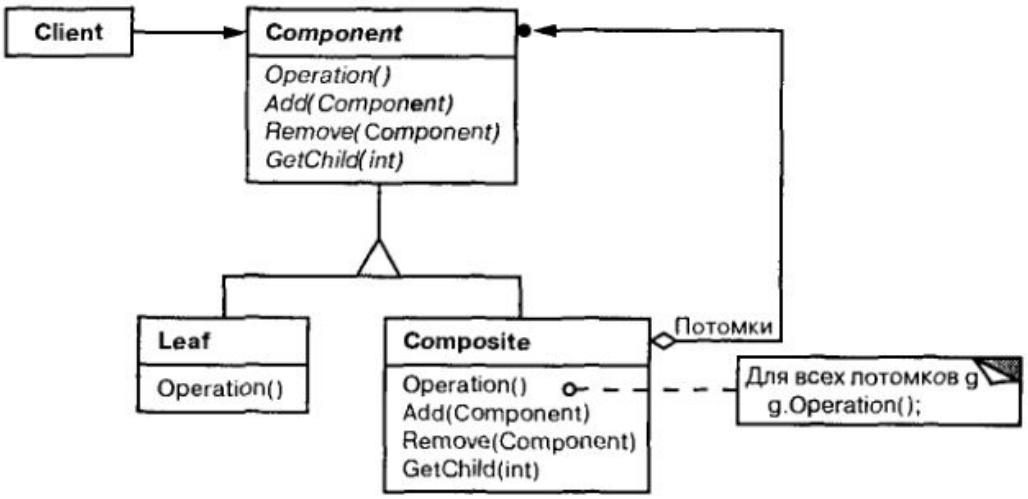
Рекурсивная композиция пригодна не только для документов. Мы можем воспользоваться ей для представления любых потенциально сложных иерархических структур. Паттерн компоновщик инкапсулирует сущность рекурсивной композиции объектно-ориентированным способом.

Ключом к паттерну компоновщик является абстрактный класс, который представляет одновременно и примитивы, и контейнеры (в нашем случае это Glyph). В нем объявлены операции, специфичные для каждого вида графического объекта (такие как Draw) и общие для всех составных объектов, например, операции для доступа и управления потомками. Подклассы Character, Rectangle и Polygon определяют примитивные графические объекты. В них операция Draw реализована соответственно для рисования прямых, прямоугольников и текста. Поскольку у примитивных объектов нет потомков, то ни один из этих подклассов не реализует операции, относящиеся к управлению потомками. Класс Row определяет агрегат, состоящий из объектов Glyph. Реализованная в нем операция Draw вызывает одноименную функцию для каждого потомка, а операции для работы с потомками уже не пусты. Поскольку интерфейс класса Row соответствует интерфейсу Glyph, то в состав объекта Row могут входить и другие такие же объекты.

Итак, паттерн компоновщик полезен, когда:

- нужно представить иерархию объектов вида часть-целое;
- хотим, чтобы клиенты единообразно трактовали составные и индивидуальные объекты.

Обобщенная схема:



Итого, паттерн Компоновщик:

- определяет иерархии классов, состоящие из примитивных и составных объектов. Из примитивных объектов можно составлять более сложные, которые, в свою очередь, участвуют в более сложных композициях и так далее. Любой клиент, ожидающий примитивного объекта, может работать и с составным;
- упрощает архитектуру клиента. Клиенты могут единообразно работать с индивидуальными объектами и с составными структурами. Обычно клиенту не известно, взаимодействует ли он с листовым или составным объектом. Это упрощает код клиента, поскольку нет необходимости писать функции, ветвящиеся в зависимости от того, с объектом какого класса они работают;
- облегчает добавление новых видов компонентов. Новые подклассы классов Composite или Leaf будут автоматически работать с уже существующими структурами и клиентским кодом. Изменять клиента при добавлении новых компонентов не нужно.

Оформление пользовательского интерфейса

Рассмотрим два усовершенствования пользовательского интерфейса нашего редактора. Первое добавляет рамку вокруг области редактирования текста, чтобы четко обозначить страницу текста, второе — полосы прокрутки, позволяющие пользователю просматривать разные части страницы. Чтобы упростить добавление и удаление таких элементов оформления (особенно во время выполнения), мы не должны использовать наследование. Максимальной гибкости можно достичь, если другим объектам пользовательского интерфейса даже не будет известно о том, какие еще есть элементы оформления. Это позволит добавлять и удалять декорации, не изменяя других классов. Опять же, желательно, чтобы полосы прокрутки могли появляться и исчезать во время выполнения, например, если текст перестаёт помещаться целиком в видимую область, а рамка может включаться/выключаться по опции.

Прозрачное обрамление

В программировании оформление пользовательского интерфейса означает расширение существующего кода. Использование для этой цели наследования не дает возможности реорганизовать интерфейс во время выполнения. Не менее серьезной проблемой является комбинаторный рост числа классов в случае широкого использования наследования.

Можно было бы добавить рамку к классу Composition, породив от него новый подкласс BorderedComposition. Точно так же можно было бы добавить и интерфейс прокрутки, породив подкласс ScrollableComposition. Если же мы хотим иметь и рамку, и полосу прокрутки, следовало бы создать подкласс BorderedScrollableComposition, и так далее. Если довести эту идею до логического завершения, то пришлось бы создавать отдельный подкласс для каждой возможной комбинации элементов оформления. Это решение быстро перестает работать, когда количество разнообразных декораций растет.

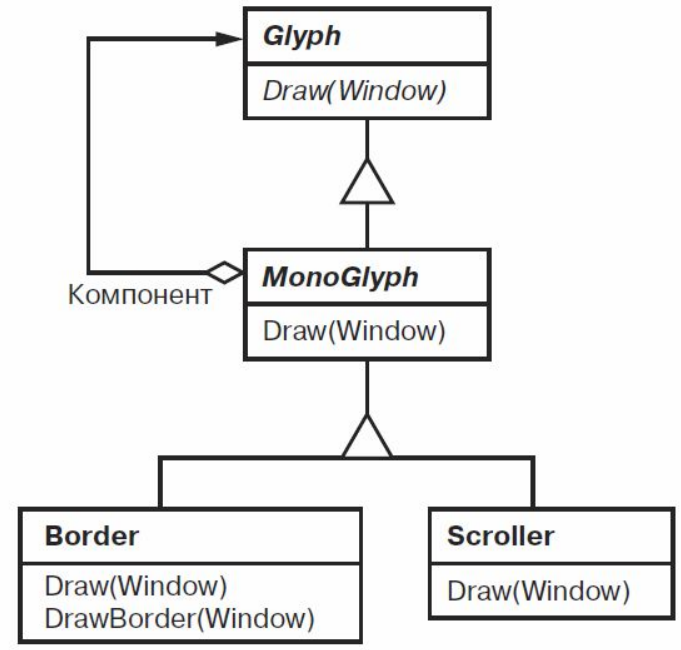
С помощью композиции объектов можно найти куда более приемлемый и гибкий механизм расширения. Но из каких объектов составлять композицию? Поскольку известно, что мы оформляем существующий глиф, то и сам элемент оформления могли бы сделать объектом (скажем, экземпляром класса Border). Следовательно, композиция может быть составлена из глифа и рамки. Следующий шаг — решить, что является агрегатом, а что — компонентом. Допустимо считать, что рамка содержит глиф, и это имеет смысл, так как рамка окружает глиф на экране. Можно принять и противоположное решение — поместить рамку внутрь глифа, но тогда пришлось бы модифицировать соответствующий подкласс класса Glyph, чтобы он «знал» о наличии рамки. Первый вариант — включение глифа в рамку — позволяет поместить весь код для отображения рамки в классе Border, оставив остальные классы без изменения.

Как выглядит класс Border? Тот факт, что у рамки есть визуальное представление, наталкивает на мысль, что она должна быть глифом, то есть подклассом класса Glyph. Но есть и более настоятельные причины поступить именно таким образом: клиентов не должно волновать, есть у глифов рамки или нет. Все глифы должны трактоваться единообразно. Когда клиент сообщает простому глифу без рамки о необходимости нарисовать себя, тот делает это, не добавляя никаких элементов оформления. Если же этот глиф заключен в рамку, то клиент не должен обрабатывать рамку как-то специально; он просто предписывает составному глифу выполнить отображение точно так же, как и простому глифу в предыдущем случае. Отсюда следует, что интерфейс класса Border должен соответствовать интерфейсу класса Glyph. Чтобы гарантировать это, мы и делаем Border подклассом Glyph.

Все это подводит нас к идее прозрачного обрамления (transparent enclosure), где комбинируются понятия о композиции с одним потомком (однокомпонентные), и о совместимых интерфейсах. В общем случае клиенту неизвестно, имеет ли он дело с компонентом или его обрамлением (то есть родителем), особенно если обрамление просто делегирует все операции своему единственному компоненту. Но обрамление может также и расширять поведение компонента, выполняя дополнительные действия либо до, либо после делегирования (а возможно, и до, и после). Обрамление может также добавить компоненту состояние. Как именно, будет показано чуть позже.

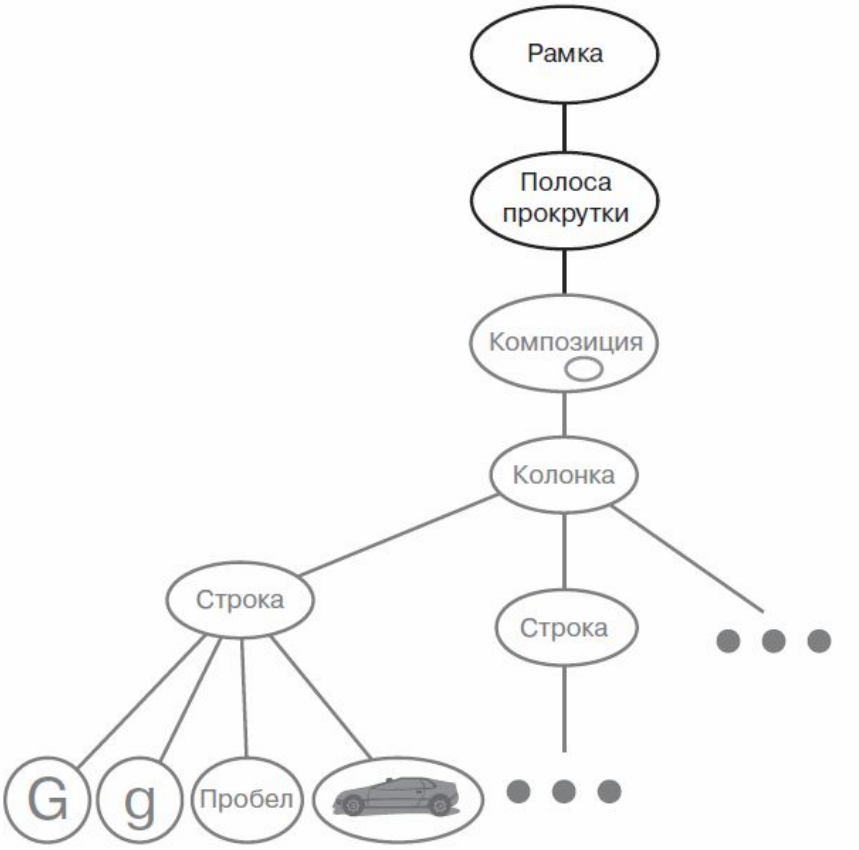
Моноглиф

Концепцию прозрачного обрамления можно применить ко всем глифам, оформляющим другие глифы. Чтобы конкретизировать эту идею, определим подкласс класса Glyph, называемый MonoGlyph. Он будет выступать в роли абстрактного класса для глифов-декораций вроде рамки.

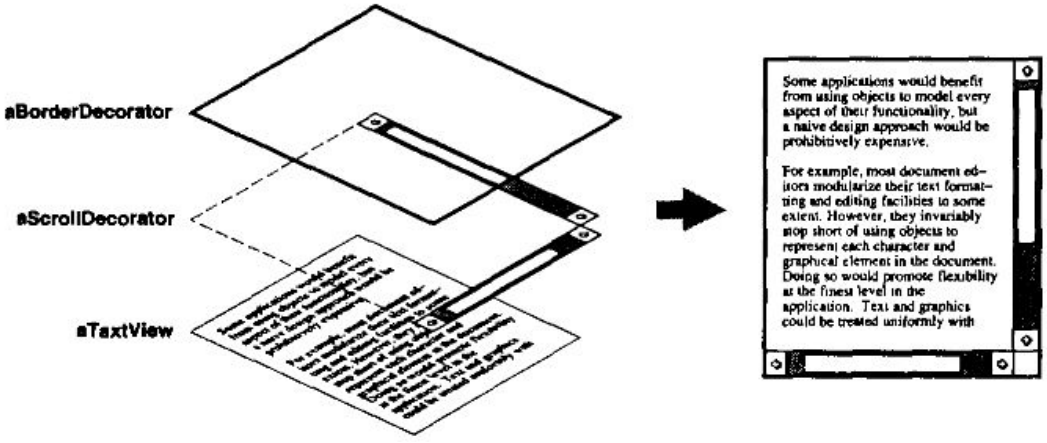


В классе MonoGlyph хранится ссылка на компонент, которому он и переадресует все запросы. При этом MonoGlyph по определению становится абсолютно прозрачным для клиентов. Подклассы MonoGlyph замещают по меньшей мере один из таких методов переадресации. Например, Border::draw() сначала вызывает метод родительского класса MonoGlyph::draw(), чтобы компонент выполнил свою часть работы, то есть нарисовал все, кроме рамки. Затем Border::Draw() рисует рамку, вызывая свой собственный закрытый метод drawBorder(), который делает что-то еще. Обратите внимание, что Border::draw(), по сути дела, расширяет метод родительского класса, чтобы нарисовать рамку. Это не то же самое, что простая замена метода: в таком случае MonoGlyph::draw() не вызывался бы. На рисунке выше показан другой подкласс класса MonoGlyph. Scroller — это MonoGlyph, который рисует свои компоненты на экране в зависимости от положения двух полос прокрутки, добавляющихся в качестве элементов оформления. Когда Scroller отображает свой компонент, графическая система обрезает его по границам окна. Отсеченные части компонента, оказавшиеся за пределами видимой части окна, не появляются на экране.

Теперь у нас есть все, что необходимо для добавления рамки и прокрутки к области редактирования текста в нашем редакторе. Мы помещаем имеющийся экземпляр класса Composition в экземпляр класса Scroller, чтобы добавить интерфейс прокрутки, а результат композиции еще раз погружаем в экземпляр класса Border.



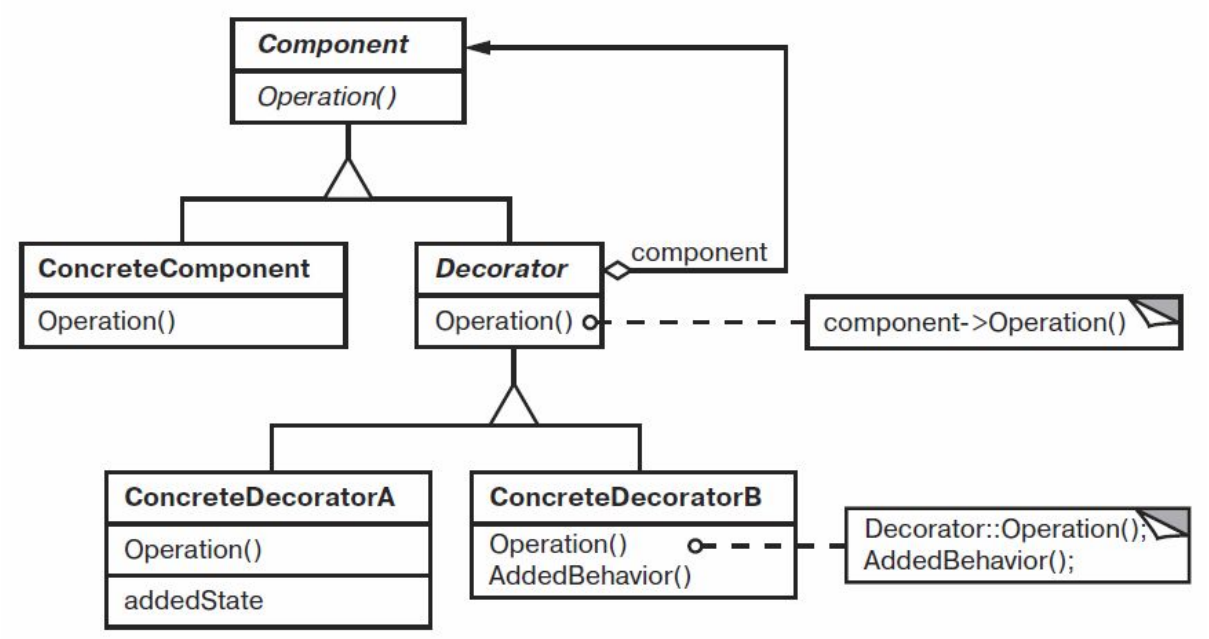
Стоит отметить, что мы могли изменить порядок композиции на обратный, сначала добавив рамку, а потом погрузив результат в Scroller. В таком случае рамка прокручивалась бы вместе с текстом. Важно, что прозрачное обрамление легко позволяет экспериментировать с разными вариантами, освобождая клиента от знания деталей кода, добавляющего декорации.



Паттерн Декоратор

Паттерн декоратор динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности. Абстрагирует отношения между классами и объектами, необходимые для поддержки оформления с помощью техники прозрачного обрамления. Термин «оформление» на самом деле применяется в более широком

смысле, чем мы рассмотрели сейчас. В паттерне декоратор под ним понимается нечто, что возлагает на объект новые обязанности. Можно, например, представить себе оформление абстрактного дерева синтаксического разбора семантическими действиями, конечного автомата — новыми состояниями или сети, состоящей из устойчивых объектов — тэгами атрибутов. Декоратор обобщает подход, который мы использовали в графическом редакторе, чтобы расширить его область применения.



Используйте паттерн Декоратор:

- для динамического, прозрачного для клиентов добавления обязанностей объектам;
- для реализации обязанностей, которые могут быть сняты с объекта;
- когда расширение путем порождения подклассов по каким-то причинам неудобно или невозможно. Иногда приходится реализовывать много независимых расширений, так что порождение подклассов для поддержки всех возможных комбинаций приведёт к комбинаторному росту их числа. В других случаях определение класса может быть скрыто или почему-либо еще недоступно, так что породить от него подкласс нельзя.

У паттерна Декоратор есть, по крайней мере, два плюса и два минуса:

- бо́льшая гибкость, нежели у статического наследования. Паттерн декоратор позволяет более гибко добавлять объекту новые обязанности, чем было бы возможно в случае статического (множественного) наследования. Декоратор может добавлять и удалять обязанности во время выполнения программы. При использовании же наследования требуется создавать новый класс для каждой дополнительной обязанности (например, BorderedScrollableTextView, BorderedTextView), что ведет к увеличению числа классов и, как следствие, к возрастанию сложности системы. Кроме того, применение нескольких декораторов к одному компоненту позволяет произвольным образом сочетать обязанности.

  Декораторы позволяют легко добавить одно и то же свойство дважды. Например, чтобы окружить объект TextView двойной рамкой, нужно просто

добавить два декоратора BorderDecorators. Двойное наследование классу Border в лучшем случае чревато ошибками;

- позволяет избежать перегруженных функциями классов на верхних уровнях иерархии. Декоратор разрешает добавлять новые обязанности по мере необходимости. Вместо того чтобы пытаться поддержать все мыслимые возможности в одном сложном, допускающем разностороннюю настройку классе, вы можете определить простой класс и постепенно наращивать его функциональность с помощью декораторов. В результате приложение уже не платит за неиспользуемые функции. Нетрудно также определять новые виды декораторов независимо от классов, которые они расширяют, даже если первоначально такие расширения не планировались. При расширении же сложного класса обычно приходится вникать в детали, не имеющие отношения к добавляемой функции;
- декоратор и его компонент не идентичны. Декоратор действует как прозрачное обрамление. Но декорированный компонент все же не идентичен исходному. При использовании декораторов это следует иметь в виду;
- множество мелких объектов. При использовании в проекте паттерна декоратор нередко получается система, составленная из большого числа мелких объектов, которые похожи друг на друга и различаются только способом взаимосвязи, а не классом и не значениями своих внутренних переменных. Хотя проектировщик, разбирающийся в устройстве такой системы, может легко настроить ее, но изучать и отлаживать ее очень тяжело.

Инкапсуляция алгоритма форматирования

Определившись, как представлять внутреннюю структуру документа, далее нужно разобраться с тем, как сконструировать конкретную физическую структуру, соответствующую правильно отформатированному документу. Представление и форматирование — это разные аспекты проектирования. Описание внутренней структуры не дает возможности добраться до определенной подструктуры. Ответственность за это лежит в основном на самом редакторе. Редактор разбивает текст на строки, строки — на колонки и т.д., учитывая при этом пожелания пользователя. Так, пользователь может изменить ширину полей, размер отступа и положение точек табуляции, установить одиночный или двойной междустрочный интервал, а также задать много других параметров форматирования. В процессе форматирования все это должно учитываться. Пользователь хочет задавать только высокоуровневые ограничения на физическую структуру, не вдаваясь в подробности внутреннего устройства.

Без ограничения общности сузим значение термина «форматирование», понимая под этим лишь разбиение на строки, и будем придумывать такой подход, чтобы он был в равной мере применим и к разбиению строк на колонки, и к разбиению колонок на страницы.

С учетом всех ограничений и деталей процесс форматирования с трудом поддается автоматизации. К этой проблеме есть много подходов, и у разных алгоритмов форматирования имеются свои сильные и слабые стороны. Поскольку мы пытаемся проектировать WYSIWYG-редактор, важно соблюдать компромисс между качеством и скоростью форматирования. В общем случае желательно, чтобы

редактор реагировал достаточно быстро и при этом внешний вид документа оставался приемлемым. На достижение этого компромисса влияет много факторов, и не все из них удастся установить на этапе компиляции. Например, можно предположить, что пользователь готов смириться с замедленной реакцией в обмен на лучшее качество форматирования. При таком предположении нужно было бы применять совершенно другой алгоритм форматирования. Есть также компромисс между временем и памятью, скорее, имеющий отношение к реализации: время форматирования можно уменьшить, если хранить в памяти больше информации.

Поскольку алгоритмы форматирования обычно оказываются весьма сложными, желательно, чтобы они были достаточно замкнутыми, а еще лучше — полностью независимыми от структуры документа. Оптимальный вариант — добавление нового вида глифа вовсе не затрагивает алгоритм форматирования. С другой стороны, при добавлении нового алгоритма форматирования не должно возникать необходимости в модификации существующих глифов.

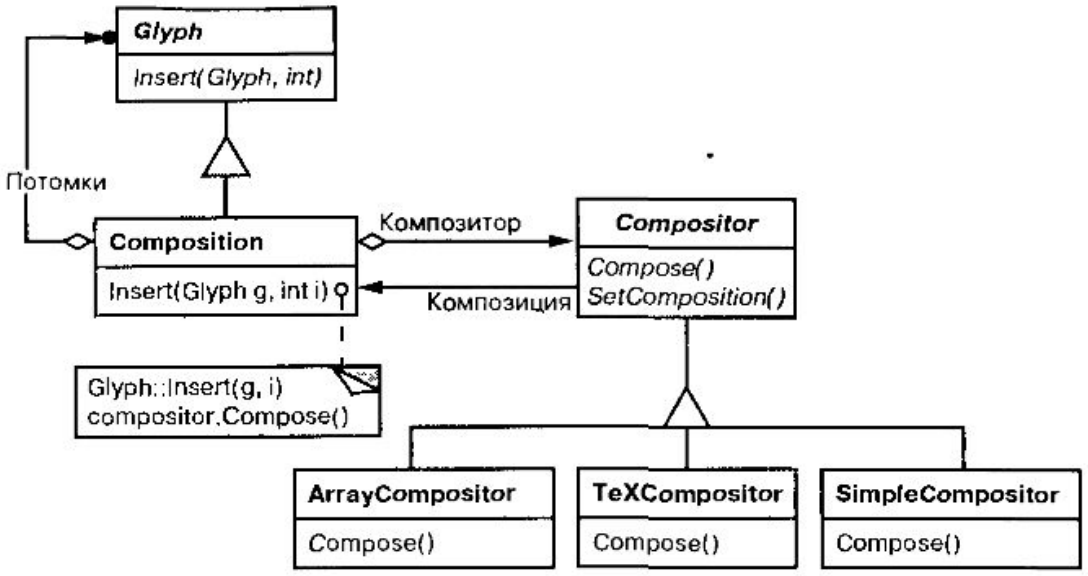
Разных алгоритмов форматирования бывает много, от простых до весьма навороченных алгоритмов, применяющихся, например, в texe. Там учитывается даже такая штука, как цвет документа — равномерность распределения текста и пустого пространства. Tex — не WYSIWYG, так что там форматирование идёт в процессе компиляции .tex-файла, которая сама по себе занимает довольно много времени, так что и форматирование может работать очень долго, пользователь всё равно останется доволен. При редактировании “на лету”, после каждой введённой буквы такое не сделать, да и не нужно, так что алгоритмы форматирования должно быть можно менять, может даже на лету, скажем, по опции в настройках.

Учитывая все вышесказанное, мы должны постараться спроектировать наш редактор так, чтобы алгоритм форматирования легко можно было заменить, по крайней мере, на этапе компиляции, если уж не во время выполнения. Допустимо изолировать алгоритм и одновременно сделать его легко замещаемым путем инкапсулирования в объекте. Точнее, мы определим отдельную иерархию классов для объектов, инкапсулирующих алгоритмы форматирования. Для корневого класса иерархии определим интерфейс, который поддерживает широкий спектр алгоритмов, а каждый подкласс будет реализовывать этот интерфейс в виде конкретного алгоритма форматирования. Тогда удастся ввести подкласс класса Glyph, который будет автоматически структурировать своих потомков с помощью переданного ему объекта-алгоритма.

Классы Compositor и Composition

Определим класс Compositor для объектов, которые могут инкапсулировать алгоритм форматирования. Базовый интерфейс этого класса будет содержать всего два метода — setComposition(), принимающий объект, который хотим форматировать, и метод compose(), который собственно инициирует форматирование.

Форматируемые композитором глифы являются потомками специального подкласса класса Glyph, который называется Composition. Композиция при создании получает объект некоторого подкласса Compositor (специализированный для конкретного алгоритма разбиения на строки) и в нужные моменты предписывает композитору форматировать глифы, по мере того как пользователь изменяет документ.



Неформатированный объект Composition содержит только видимые глифы, составляющие основное содержание документа. В нем нет глифов, определяющих физическую структуру документа, например Row и Column. В таком состоянии композиция находится сразу после создания и инициализации глифами, которые должна отформатировать. Во время форматирования композиция вызывает метод compose() своего объекта Compositor. Композитор обходит всех потомков композиции и вставляет новые глифы Row и Column в соответствии со своим алгоритмом разбиения на строки.

Каждый подкласс класса Compositor может реализовывать свой собственный алгоритм форматирования. Например, класс SimpleCompositor мог бы осуществлять быстрый проход, не обращая внимания на такую экзотику, как «цвет» документа. Под «хорошим цветом» понимается равномерное распределение текста и пустого пространства. Класс TeXCompositor мог бы реализовывать полный алгоритм TeX, учитывающий наряду со многими другими вещами и цвет, но за счет увеличения времени форматирования. Наличие классов Compositor и Composition обеспечивает четкое отделение кода, поддерживающего физическую структуру документа, от кода различных алгоритмов форматирования. Мы можем добавить новые подклассы к классу Compositor, не трогая классов глифов, и наоборот. Фактически допустимо подменить алгоритм разбиения на строки во время выполнения, добавив один-единственный метод setCompositor() к базовому интерфейсу класса Composition.

Паттерн Стратегия

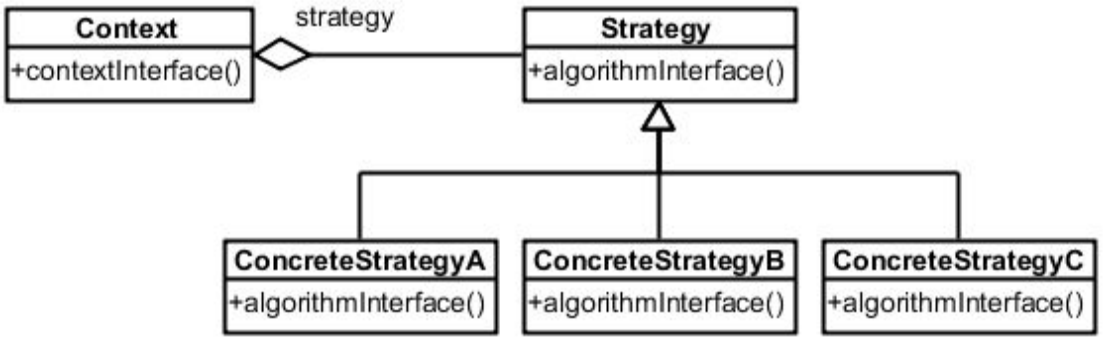
Инкапсуляция алгоритма в объект — это назначение паттерна стратегия. Основными участниками паттерна являются объекты-стратегии, инкапсулирующие различные алгоритмы, и контекст, в котором они работают. Композиторы представляют варианты стратегий; они инкапсулируют алгоритмы форматирования. Композиция — это контекст для стратегии композитора.

Ключ к применению паттерна стратегия — спроектировать интерфейсы стратегии и контекста, достаточно общие для поддержки широкого диапазона алгоритмов. Не

должно возникать необходимости изменять интерфейс стратегии или контекста для поддержки нового алгоритма. В нашем примере поддержка доступа к потомкам, их вставки и удаления, предоставляемая базовыми интерфейсами класса Glyph, достаточно общая, чтобы подклассы класса Compositor могли изменять физическую структуру документа независимо от того, с помощью каких алгоритмов это делается. Аналогично интерфейс класса Compositor дает композициям все, что им необходимо для инициализации форматирования.

Паттерн стратегия имеет смысл применять, когда:

- имеется много родственных классов, отличающихся только поведением. Стратегия позволяет сконфигурировать класс, задав одно из возможных поведений;
- нужно иметь несколько разных вариантов алгоритма. Например, можно определить два варианта алгоритма, один из которых требует больше времени, а другой — больше памяти;
- в алгоритме содержатся данные, о которых клиент не должен «знать»;
- в классе определено много поведений, что представлено разветвленными условными операторами. В этом случае проще перенести код из ветвей в отдельные классы стратегий.



Паттерн Адаптер

Иногда класс из инструментальной библиотеки, спроектированный для повторного использования, не удается использовать только потому, что его интерфейс не соответствует тому, который нужен конкретному приложению.

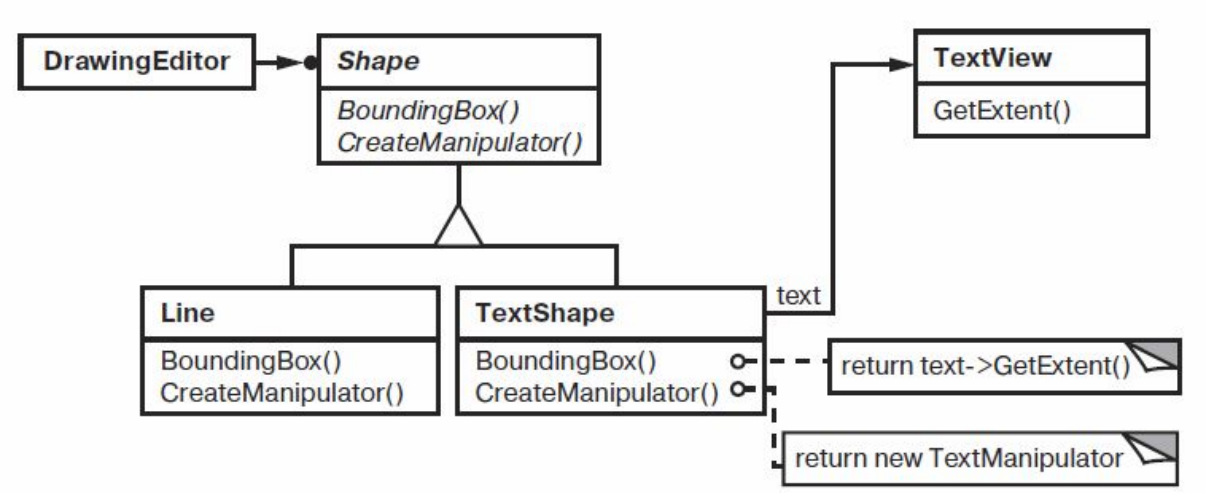
Рассмотрим, например, графический редактор, благодаря которому пользователи могут рисовать на экране графические элементы (линии, многоугольники, текст и т.д.) и организовывать их в виде картинок и диаграмм. Основной абстракцией графического редактора является графический объект, который имеет изменяемую форму и изображает сам себя. Интерфейс графических объектов определен абстрактным классом Shape. Редактор определяет подкласс класса Shape для каждого вида графических объектов: LineShape для прямых, PolygonShape для многоугольников и т.д.

Классы для элементарных геометрических фигур, например LineShape и PolygonShape, реализовать сравнительно просто, поскольку заложенные в них возможности рисования и редактирования крайне ограничены. Но подкласс TextShape, умеющий отображать и редактировать текст, уже значительно сложнее, поскольку даже для простейших операций редактирования текста нужно нетривиальным образом

обновлять экран и управлять буферами. В то же время, возможно, существует уже готовая библиотека для разработки пользовательских интерфейсов, которая предоставляет развитый класс TextView, позволяющий отображать и редактировать текст. В идеале мы хотели бы повторно использовать TextView для реализации TextShape, но библиотека разрабатывалась без учета классов Shape, поэтому заставить объекты TextView и Shape работать совместно не удается.

Так каким же образом существующие и независимо разработанные классы вроде TextView могут работать в приложении, которое спроектировано под другой, несовместимый интерфейс? Можно было бы так изменить интерфейс класса TextView, чтобы он соответствовал интерфейсу Shape, только для этого нужен исходный код. Но даже если он доступен, то вряд ли разумно изменять TextView; библиотека не должна приспосабливаться к интерфейсам каждого конкретного приложения.

Вместо этого мы могли бы определить класс TextShape так, что он будет адаптировать интерфейс TextView к интерфейсу Shape. Это допустимо сделать двумя способами: наследуя интерфейс от Shape, а реализацию от TextView; включив экземпляр TextView в TextShape и реализовав TextShape в терминах интерфейса TextView. Два данных подхода соответствуют вариантам паттерна адаптер в его классовой и объектной ипостасях. Класс TextShape мы будем называть адаптером.



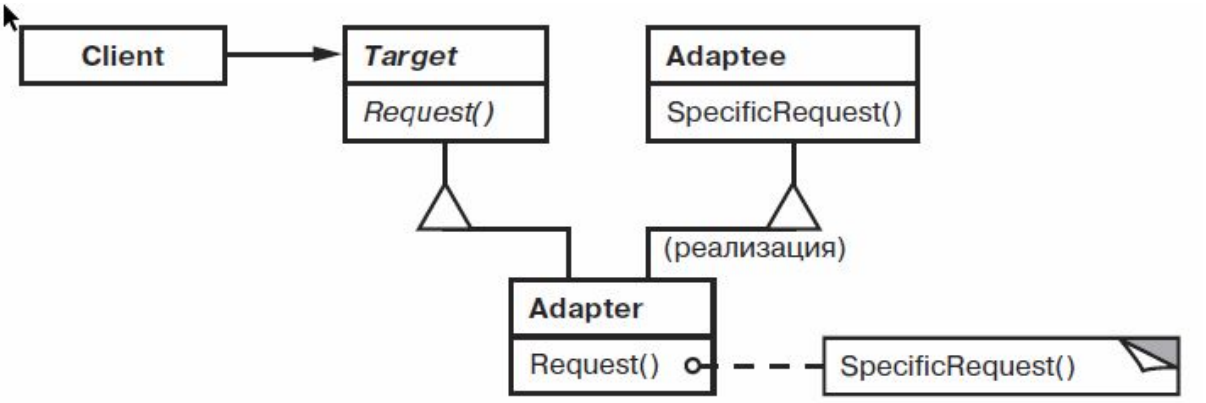
На этой диаграмме показан адаптер объекта. Видно, как запрос BoundingBox, объявленный в классе Shape, преобразуется в запрос GetExtent, определенный в классе TextView. Поскольку класс TextShape адаптирует TextView к интерфейсу Shape, графический редактор может воспользоваться классом TextView, хотя тот и имеет несовместимый интерфейс.

Часто адаптер отвечает за функциональность, которую не может предоставить адаптируемый класс. На диаграмме показано, как адаптер выполняет такого рода функции. У пользователя должна быть возможность перемещать любой объект класса Shape в другое место, но в классе TextView такая операция не предусмотрена. TextShape может добавить недостающую функциональность, самостоятельно реализовав операцию CreateManipulator класса Shape, которая возвращает экземпляр подходящего подкласса Manipulator.

Manipulator – это абстрактный класс объектов, которым известно, как анимировать Shape в ответ на такие действия пользователя, как перетаскивание фигуры в другое место. У класса Manipulator имеются подклассы для различных

фигур. Например, TextManipulator – подкласс для TextShape. Возвращая экземпляр TextManipulator, объект класса TextShape добавляет новую функциональность, которой в классе TextView нет, а классу Shape требуется.

Адаптер класса использует множественное наследование для адаптации одного интерфейса к другому.



Применяйте паттерн адаптер, когда:

- хотите использовать существующий класс, но его интерфейс не соответствует вашим потребностям;
- собираетесь создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы;
- (только для адаптера объектов) нужно использовать несколько существующих подклассов, но непрактично адаптировать их интерфейсы путем порождения новых подклассов от каждого. В этом случае адаптер объектов может приспосабливать интерфейс их общего родительского класса.

Результаты применения адаптеров объектов и классов различны. Адаптер класса:

- адаптирует Adaptee к Target, перепоручая действия конкретному классу Adaptee. Поэтому данный паттерн не будет работать, если мы захотим одновременно адаптировать класс и его подклассы;
- позволяет адаптеру Adapter заместить некоторые операции адаптируемого класса Adaptee, так как Adapter есть не что иное, как подкласс Adaptee;
- вводит только один новый объект. Чтобы добраться до адаптируемого класса, не нужно никакого дополнительного обращения по указателю.

Адаптер объектов:

- позволяет одному адаптеру Adapter работать со многим адаптируемыми объектами Adaptee, то есть с самим Adaptee и его подклассами (если таковые имеются). Адаптер может добавить новую функциональность сразу всем адаптируемым объектам;
- затрудняет замещение операций класса Adaptee. Для этого потребуется породить от Adaptee подкласс и заставить Adapter ссылаться на этот подкласс, а не на сам Adaptee.

Паттерн Прокси

Данный паттерн позволяет гибко управлять доступом к объекту. Например, отложить расходы на создание и инициализацию объекта до момента, когда объект действительно понадобится.

Рассмотрим редактор документов, который допускает встраивание в документ графических объектов. Затраты на создание некоторых таких объектов, например больших растровых изображений, могут быть весьма значительны. Но документ должен открываться быстро, поэтому следует избегать создания всех «тяжелых» объектов на стадии открытия (да и вообще это излишне, поскольку не все они будут видны одновременно).

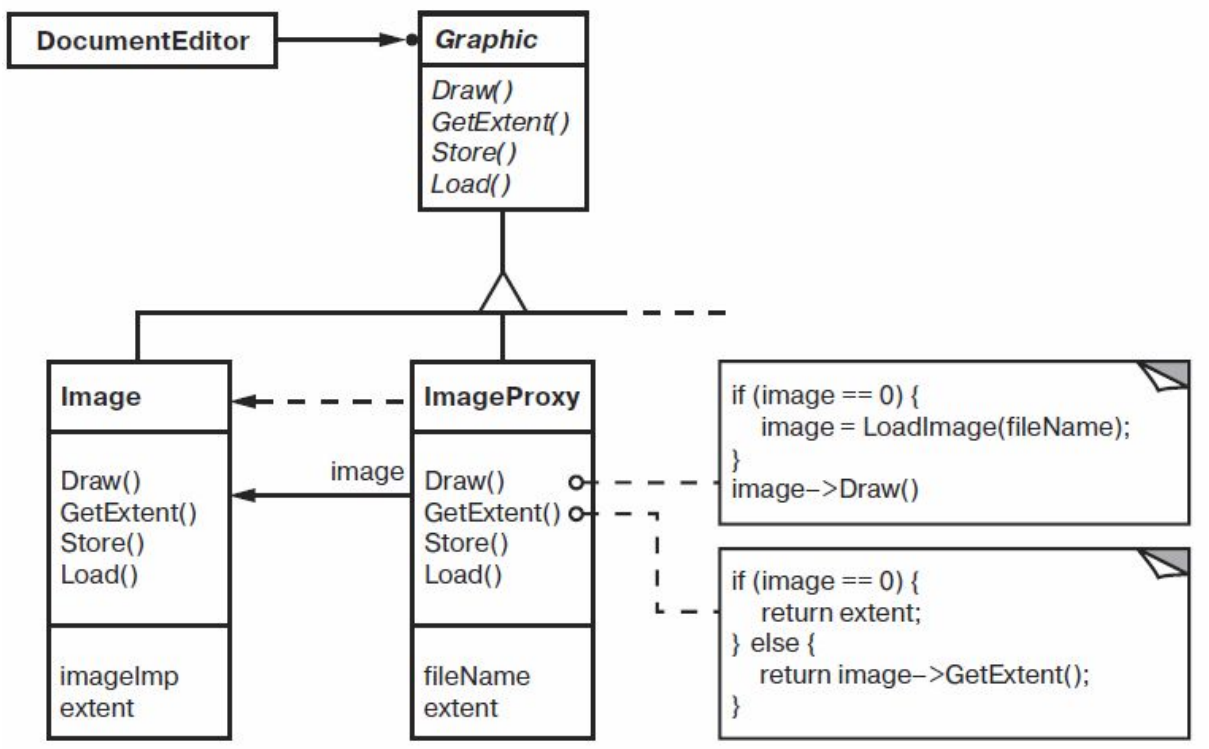
В связи с такими ограничениями кажется разумным создавать «тяжелые» объекты по требованию. Это означает «когда изображение становится видимым». Но что поместить в документ вместо изображения? И как, не усложняя реализации редактора, скрыть то, что изображение создается по требованию? Например, оптимизация не должна отражаться на коде, отвечающем за рисование и форматирование.

Решение состоит в том, чтобы использовать другой объект – заместитель изображения, который временно подставляется вместо реального изображения. Заместитель ведет себя точно так же, как само изображение, и выполняет при необходимости его инстанцирование.

Заместитель создает настоящее изображение, только если редактор документа вызовет операцию Draw. Все последующие запросы заместитель переадресует непосредственно изображению. Поэтому после создания изображения он должен сохранить ссылку на него.

Предположим, что изображения хранятся в отдельных файлах. В таком случае мы можем использовать имя файла как ссылку на реальный объект. Заместитель хранит также размер изображения, то есть длину и ширину. «Зная» ее, заместитель может отвечать на запросы форматера о своем размере, не инстанцируя изображение.

На следующей диаграмме классов этот пример показан более подробно.



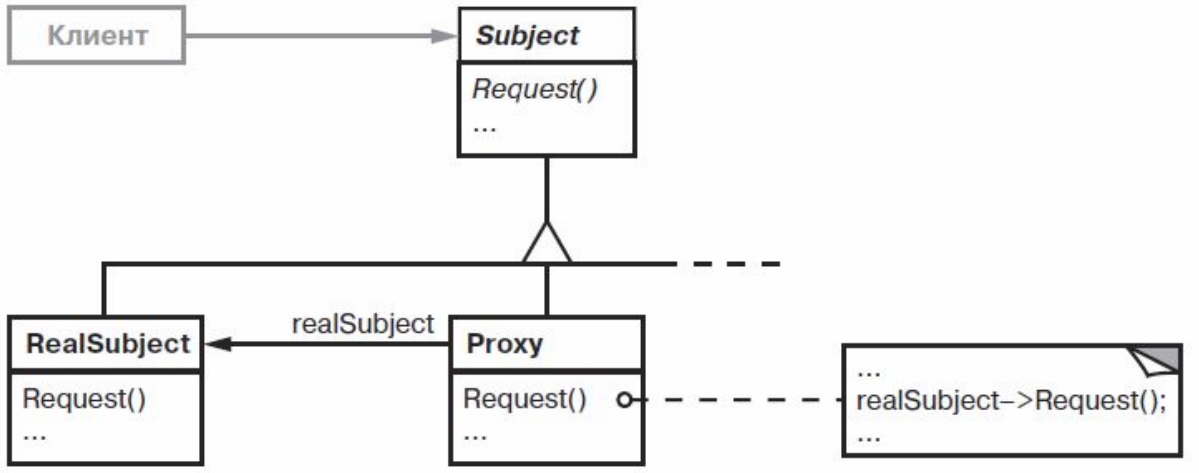
Редактор документов получает доступ к встроенным изображениям только через интерфейс, определенный в абстрактном классе Graphic. ImageProxy – это класс для представления изображений, создаваемых по требованию. В ImageProxy хранится имя файла, играющее роль ссылки на изображение, которое находится на диске. Имя файла передается конструктору класса ImageProxy.

В объекте ImageProxy находятся также ограничивающий прямоугольник изображения и ссылка на экземпляр реального объекта Image. Ссылка остается недействительной, пока заместитель не инстанцирует реальное изображение. Операцией Draw гарантируется, что изображение будет создано до того, как заместитель переадресует ему запрос. Операция GetExtent переадресует запрос изображению, только если оно уже инстанцировано; в противном случае ImageProxy возвращает размеры, которые хранит сам.

Паттерн заместитель применим во всех случаях, когда возникает необходимость сослаться на объект более изощренно, чем это возможно, если использовать простой указатель. Вот несколько типичных ситуаций, где заместитель оказывается полезным:

- удаленный заместитель предоставляет локального представителя вместо объекта, находящегося в другом адресном пространстве.
- виртуальный заместитель создает «тяжелые» объекты по требованию. При мером может служить класс ImageProxy, описанный выше;
- защищающий заместитель контролирует доступ к исходному объекту. Такие заместители полезны, когда для разных объектов определены различные права доступа.
- «умная» ссылка – это замена обычного указателя. Она позволяет выполнить дополнительные действия при доступе к объекту. К типичным применениям такой ссылки можно отнести:

- подсчет числа ссылок на реальный объект, с тем чтобы занимаемую им память можно было освободить автоматически, когда не останется ни одной ссылки (такие ссылки называют еще «умными» указателями);
- загрузку объекта в память при первом обращении к нему;
- проверку и установку блокировки на реальный объект при обращении к нему, чтобы никакой другой объект не смог в это время изменить его.
- копирование при записи (copy on write). Копирование большого и сложного объекта – очень дорогая операция. Если копия не модифицировалась, то нет смысла эту цену платить. Если отложить процесс копирования, применив заместитель, то можно быть уверенным, что эта операция произойдет только тогда, когда он действительно был изменен.

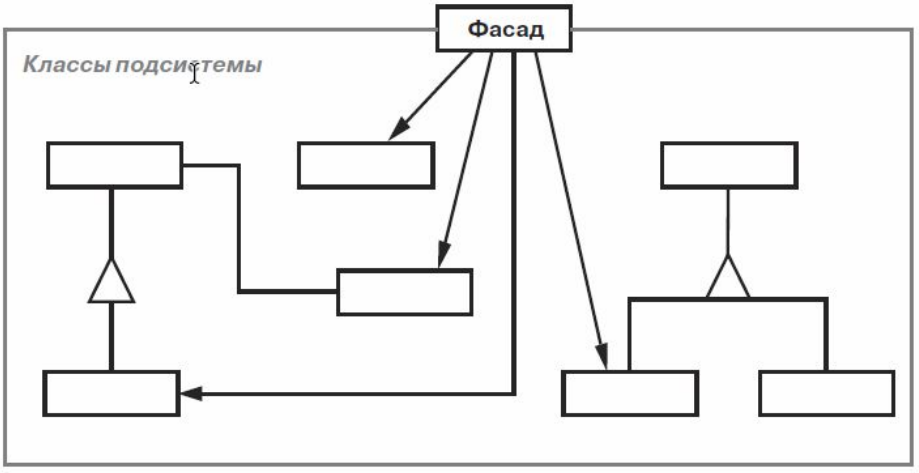


Паттерн Фасад

Разбиение на подсистемы облегчает проектирование сложной системы в целом. Общая цель всякого проектирования – свести к минимуму зависимость подсистем друг от друга и обмен информацией между ними. Один из способов решения этой задачи – введение объекта фасад, предоставляющий единый упрощенный интерфейс к более сложным системным средствам.

Рассмотрим, например, среду программирования, которая дает приложениям доступ к подсистеме компиляции. В этой подсистеме имеются такие классы, как Scanner (лексический анализатор), Parser (синтаксический анализатор), ProgramNode (узел программы), BytecodeStream (поток байтовых кодов) и ProgramNodeBuilder (строитель узла программы). Все вместе они составляют компилятор. Некоторым специализированным приложениям, возможно, понадобится прямой доступ к этим классам. Но для большинства клиентов компилятора такие детали, как синтаксический разбор и генерация кода, обычно не нужны; им просто требуется откомпилировать некоторую программу. Для таких клиентов применение мощного, но низкоуровневого интерфейса подсистемы компиляции только усложняет задачу.

Чтобы предоставить интерфейс более высокого уровня, изолирующий клиента от этих классов, в подсистему компиляции включен также класс Compiler (компилятор). Он определяет унифицированный интерфейс ко всем возможностям компилятора. Класс Compiler выступает в роли фасада: предлагает простой интерфейс к более сложной подсистеме. Он «склеивает» классы, реализующие функциональность компилятора, но не скрывает их полностью. Благодаря фасаду компилятора работа большинства программистов облегчается. При этом те, кому нужен доступ к средствам низкого уровня, не лишаются его.



Используйте паттерн фасад, когда:

- хотите предоставить простой интерфейс к сложной подсистеме. Часто подсистемы усложняются по мере развития. Применение большинства паттернов приводит к появлению меньших классов, но в большем количестве. Такую подсистему проще повторно использовать и настраивать под конкретные нужды, но вместе с тем применять подсистему без настройки становится труднее. Фасад предлагает некоторый вид системы по умолчанию, устраивающий большинство клиентов. И лишь те объекты, которым нужны более широкие возможности настройки, могут обратиться напрямую к тому, что находится за фасадом;
- между клиентами и классами реализации абстракции существует много зависимостей. Фасад позволит отделить подсистему как от клиентов, так и от других подсистем, что, в свою очередь, способствует повышению степени независимости и переносимости;
- вы хотите разложить подсистему на отдельные слои. Используйте фасад для определения точки входа на каждый уровень подсистемы. Если подсистемы

зависят друг от друга, то зависимость можно упростить, разрешив подсистемам обмениваться информацией только через фасады.

У паттерна фасад есть следующие преимущества:

- изолирует клиентов от компонентов подсистемы, уменьшая тем самым число объектов, с которыми клиентам приходится иметь дело, и упрощая работу с подсистемой;
- позволяет ослабить связанность между подсистемой и ее клиентами. Зачастую компоненты подсистемы сильно связаны. Слабая связанность позволяет видоизменять компоненты, не затрагивая при этом клиентов. Фасады помогают разложить систему на слои и структурировать зависимости между объектами, а также избежать сложных и циклических зависимостей. Это может оказаться важным, если клиент и подсистема реализуются независимо. Уменьшение числа зависимостей на стадии компиляции чрезвычайно важно в больших системах. Хочется, конечно, чтобы время, уходящее на перекомпиляцию после изменения классов подсистемы, было минимальным. Сокращение числа зависимостей за счет фасадов может уменьшить количество нуждающихся в повторной компиляции файлов после небольшой модификации какой-нибудь важной подсистемы. Фасад может также упростить процесс переноса системы на другие платформы, поскольку уменьшается вероятность того, что в результате изменения одной подсистемы понадобится изменять и все остальные;
- фасад не препятствует приложениям напрямую обращаться к классам подсистемы, если это необходимо. Таким образом, у вас есть выбор между простотой и общностью.