### Инкапсуляция алгоритма форматирования

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 Lexi – это 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.

Учитывая все вышесказанное, мы должны постараться спроектировать Lexi 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

Мы определим класс `Compositor` (композитор) для объектов, которые могут инкапсулировать алгоритм форматирования. Интерфейс (см. табл. 2.2) позволяет объекту этого класса узнать, *какие* глифы надо форматировать и *когда*. Форматируемые композитором глифы являются потомками специального подкласса класса `Glyph`, который называется `Composition` (композиция). 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. На рис. 2.5 изображены отношения между классами `Composition` и `Compositor`.

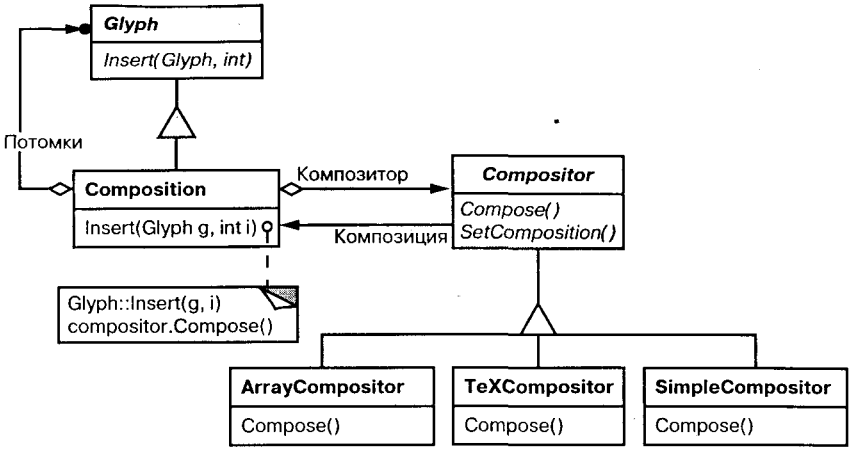


*Рис. 2.5. Отношения классов Composition и Compositor*

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 операцию `Compose` своего объекта `Compositor`. Композитор обходит всех потомков композиции и вставляет новые глифы `Row` и `Column` в соответствии со своим алгоритмом разбиения на строки.[1] На рис. 2.6 показана получающаяся объектная структура. Глифы, созданные и вставленные в эту структуру композитором, закрашены на рисунке серым цветом.

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 [Knu84], 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 одну-единственную операцию `SetCompositor` к базовому интерфейсу класса `Composition`.

---

[1] Композитор должен получить коды символов глифов `Character`, чтобы вычислить места разбиения на строки. В разделе 2.8 мы увидим, как можно получить информацию полиморфно, не добавляя специфичной для символов операции к интерфейсу класса `Glyph`.

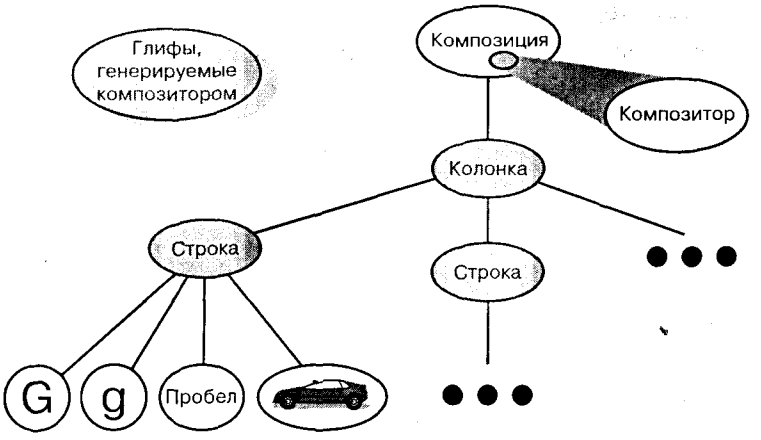


*Рис. 2.6. Объектная структура, отражающая алгоритм разбиения на строки, выбираемый композитором*

### 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.

## 2.4. Оформление пользовательского интерфейса

Рассмотрим два усовершенствования пользовательского интерфейса Lexi. 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.