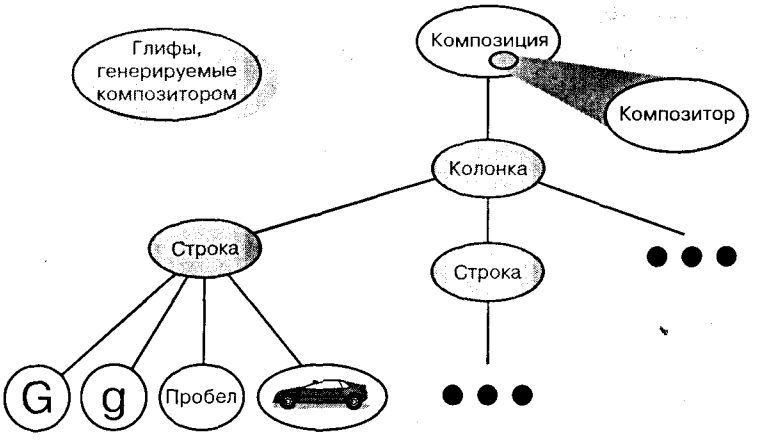

Рис. 2.6. Объектная структура, отражающая алгоритм разбиения на строки, выбираемый композитором

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.

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 ниже.