Глифы

Абстрактный класс Glyph (глиф) определяется для всех объектов, которые могут присутствовать в структуре документа[1]. Его подклассы определяют как примитивные графические элементы (скажем, символы и изображения), так и структурные элементы (строки и колонки). На рис. 2.4 изображена достаточно обширная часть иерархии класса Glyph, а в таблице 2.1 более подробно представлен базовый интерфейс этого класса в нотации C++[2].

Таблица 2.1. Базовый интерфейс класса Glyph

Обязанность	Операции
внешнее представление	`virtual void Draw(Window*)`
	`virtual void Bounds(Rect&)`
обнаружение точки воздействия	`virtual bool Intersects(const Point&)`
структура	`virtual void Insert(Glyph*, int)`
	`virtual void Remove(Glyph*)`
	`virtual Glyph* Child(int)`
	`virtual Glyph* Parent()`

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. В классе Window определены графические операции для прорисовки в окне на экране текста и основных геометрических фигур. Например, в подклассе Rectangle операция Draw могла определяться так:

```
void Rectangle::Draw (Window* w) {
    w->DrawRect(_x0, _y0, _x1, _y1);
}
```

где _x0, _y0, _x1 и _y1 – это данные-члены класса Rectangle, определяющие два противоположных угла прямоугольника, а DrawRect – операция из класса Window, рисующая на экране прямоугольник.

---

[1] Впервые термин «глиф» в этом контексте употребил Пол Кальдер [CL90]. В большинстве современных редакторов документов отдельные символы не представляются объектами, скорее всего, из соображений эффективности. Кальдер продемонстрировал практическую пригодность этого подхода в своей диссертации [Cal93]. Наши глифы проще предложенных им, поскольку мы для простоты ограничились строгими иерархиями. Глифы Кальдера могут использоваться совместно для уменьшения потребления памяти и, стало быть, образуют направленные ациклические графы. Для достижения того же эффекта мы можем воспользоваться паттерном Приспособленец, но оставим это в качестве упражнения читателю.

[2] Представленный здесь интерфейс намеренно сделан минимальным, чтобы не загромождать обсуждение техническими деталями. Полный интерфейс должен бы включать операции для работы с графическими атрибутами: цветами, шрифтами и преобразованиями координат, а также операции для более развитого управления потомками.

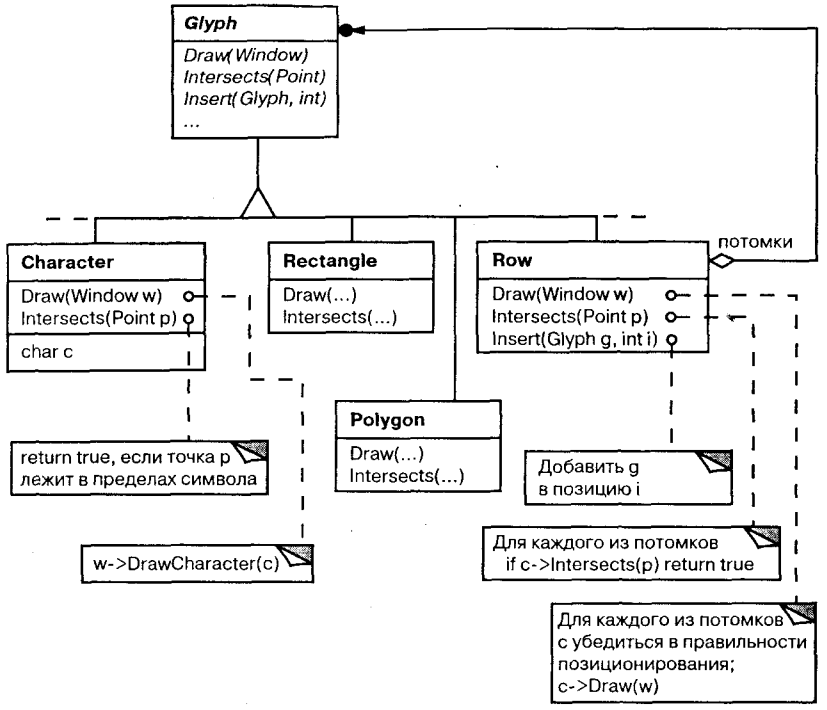

Рис. 2.4. Частичная иерархия класса Glyph

Глифу-родителю часто бывает нужно «знать», сколько места на экране занимает глиф-потомок, чтобы расположить его и остальные глифы в строке без перекрытий (как показано на рис. 2.2). 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, Lexi 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.[1] Операция `Remove` удаляет заданный глиф, если он действительно является потомком.

[1] Возможно, целочисленный индекс – не лучший способ описания потомков глифа. Это зависит от структуры данных, используемой внутри глифа. Если потомки хранятся в связанном списке, то более эффективно было бы передавать указатель на элемент списка. Мы еще увидим более удачное решение проблемы индексации в разделе 2.8, когда будем обсуждать анализ документа.

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. В Lexi 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. Сейчас самое время обратиться к разделу об этом паттерне и изучить его, имея в виду только что рассмотренный сценарий.

2.3. Форматирование

Мы решили, как *представлять* физическую 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 Lexi. 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.[1] В процессе форматирования это учитывается.

Кстати говоря, мы сузим значение термина «форматирование», понимая под этим лишь разбиение на строки. Будем считать слова «форматирование» и «разбиение на строки взаимозаменяемыми. Обсуждаемая техника в равной мере применима и к разбиению строк на колонки, и к разбиению колонок на страницы.

Таблица 2.2. Базовый интерфейс класса Compositor

Обязанность	Операции
что форматировать	`void SetComposition(Composition*)`
когда форматировать	`virtual void Compose()`

[1] У пользователя найдется что сказать и по поводу *логической* структуры документа: предложений, абзацев, разделов, глав и т.д. *Физическая* структура в общем-то менее интересна. Большинству людей не важно, где в абзаце произошел разрыв строки, если в целом все отформатировано правильно. То же самое относится и к форматированию колонок и страниц. Таким образом, пользователи задают только высокоуровневые ограничения на физическую структуру, а Lexi выполняет их.