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 (см. рис. 2.7). В классе `MonoGlyph` хранится ссылка на компонент, которому он и переадресует все запросы. При этом `MonoGlyph` по определению становится абсолютно прозрачным для клиентов. Вот как моноглиф реализует операцию `Draw`:

```
void MonoGlyph::Draw (Window* w) {
    _component->Draw(w);
}
```

Подклассы `MonoGlyph` замещают по меньшей мере одну из таких операций переадресации. Например, `Border::Draw` сначала вызывает операцию родительского класса `MonoGlyph::Draw`, чтобы компонент выполнил свою часть работы, то есть нарисовал все, кроме рамки. Затем `Border::Draw` рисует рамку, вызывая свою собственную закрытую операцию `DrawBorder`, детали которой мы опустим:

```
void Border::Draw (Window* w) {
    MonoGlyph::Draw(w);
    DrawBorder(w);
}
```

Обратите внимание, что `Border::Draw`, по сути дела, *расширяет* операцию родительского класса, чтобы нарисовать рамку. Это не то же самое, что простая *замена* операции: в таком случае `MonoGlyph::Draw` не вызывалась бы.

На рис. 2.7 показан другой подкласс класса `MonoGlyph`. `Scroller` – это `MonoGlyph`, который рисует свои компоненты на экране в зависимости от

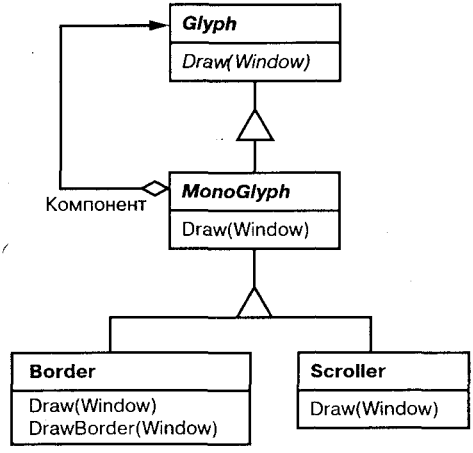

Рис. 2.7. Отношения класса MonoGlyph с другими классами

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 Lexi. Мы помещаем имеющийся экземпляр класса `Composition` в экземпляр класса `Scroller`, чтобы добавить интерфейс прокрутки, а результат композиции еще раз погружаем в экземпляр класса `Border`. Получившийся объект показан на рис. 2.8.

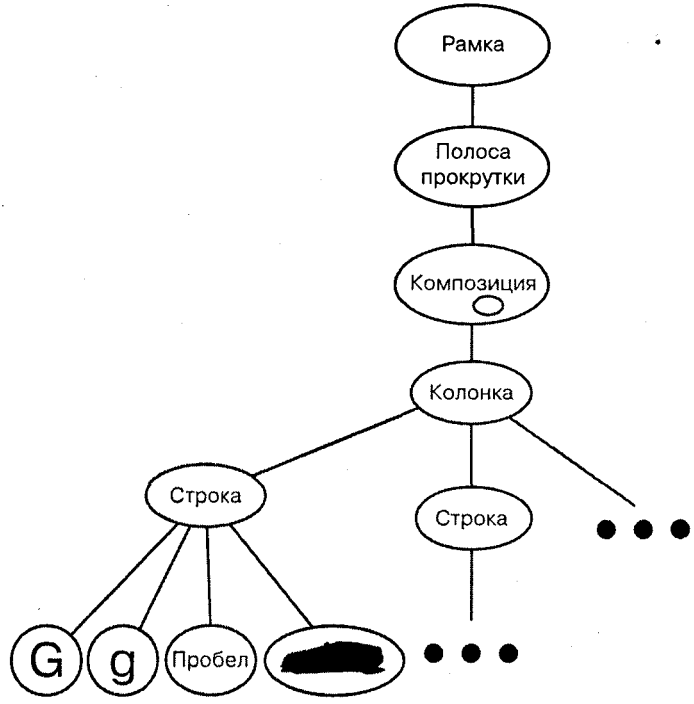

Рис. 2.8. Объектная структура после добавления элементов оформления

Обратите внимание, 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. Может быть, это то, что вам нужно, а может, и нет. Важно лишь, что прозрачное обрамление легко позволяет экспериментировать с разными вариантами, освобождая клиента от знания деталей кода, добавляющего декорации.

Отметим, что рамка допускает композицию только с одним глифом, не более того. Этим она отличается от рассмотренных выше композиций, где родительскому объекту позволялось иметь сколько угодно потомков. Здесь же заключение чего-то

в рамку предполагает, что это «что-то» имеется в единственном экземпляре. Мы могли бы приписать некоторую семантику декорации более одного объекта, но тогда пришлось бы вводить множество видов композиций с оформлением: оформление строки, колонки и т.д. Это не дает ничего нового, так как у нас уже есть классы для такого рода композиций. Поэтому для композиции лучше использовать уже существующие классы, а новые добавлять для оформления результата. Отделение декорации от других видов композиции одновременно упрощает классы, реализующие разные элементы оформления, и уменьшает их количество. Кроме того, мы избавлены от необходимости дублировать уже имеющуюся функциональность.

Паттерн декоратор

Паттерн декоратор 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 видели выше. 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 Lexi, чтобы расширить его область применения.

2.5. Поддержка нескольких стандартов внешнего облика

При проектировании системы приходится сталкиваться с серьезной проблемой: как добиться переносимости между различными программно-аппаратными платформами. Перенос Lexi на другую платформу не должен требовать капитального перепроектирования, иначе не стоит за него и браться. Он должен быть максимально прост.

Одним из препятствий для переноса является разнообразие стандартов внешнего облика, призванных унифицировать работу с приложениями на данной платформе. Эти стандарты определяют, как приложения должны выглядеть и реагировать на действия пользователя. Хотя существующие стандарты не так уж сильно отличаются друг от друга, ни один пользователь не спутает один стандарт с другим – приложения, написанные для Motif, выглядят не совсем так, как аналогичные приложения на других платформах, и наоборот. Программа, работающая более чем на одной платформе, должна всюду соответствовать принятой стилистике пользовательского интерфейса.

Наша проектная цель – сделать так, чтобы Lexi поддерживал разные стандарты внешнего облика и чтобы легко можно было добавить поддержку нового стандарта, как только таковой появится (а это неизбежно произойдет). Хотелось бы также, чтобы наш дизайн решал и другую задачу: изменение внешнего облика Lexi во время выполнения.