Абстрагирование создания объекта

Все, что мы видим и с чем можем взаимодействовать в пользовательском интерфейсе Lexi, – это визуальные глифы, скомпонованные в другие, уже невидимые глифы вроде строки (`Row`) и колонки (`Column`). Невидимые глифы содержат видимые – скажем, кнопку (`Button`) или символ (`Character`) – и правильно располагают их на экране. В руководствах по стилистическому оформлению много говорится о внешнем облике и поведении так называемых «виджетов» (widgets); это просто другое название таких видимых глифов, как кнопки, полосы прокрутки и меню, выполняющих в пользовательском интерфейсе функции элементов управления. Для представления данных виджеты могут пользоваться более простыми глифами: символами, окружностями, прямоугольниками и многоугольниками.

Мы будем предполагать, что имеется два набора классов глифов-виджетов, с помощью которых реализуются стандарты внешнего облика:

- набор абстрактных подклассов класса `Glyph` для каждой категории виджетов. Например, абстрактный класс `ScrollBar` будет дополнять интерфейс глифа с целью получения операций прокрутки общего вида, а `Button` – это абстрактный класс, добавляющий операции с кнопками;
- набор конкретных подклассов для каждого абстрактного подкласса, в которых реализованы стандарты внешнего облика. Так, у `ScrollBar` могут быть подклассы `MotifScrollBar` и `PMScrollBar`, реализующие полосы прокрутки в стиле Motif и Presentation Manager соответственно.

Lexi должен различать глифы-виджеты для разных стилей внешнего оформления. Например, когда необходимо поместить в интерфейс кнопку, редактор должен инстанцировать подкласс класса `Glyph` для нужного стиля кнопки (`MotifButton`, `PMButton`, `MacButton` и т.д.).

Ясно, что в реализации Lexi это нельзя сделать непосредственно, например, вызвав конструктор, если речь идет о языке C++. При этом была бы жестко закодирована кнопка одного конкретного стиля, значит, выбрать нужный стиль во время выполнения оказалось бы невозможно. Кроме того, мы были бы вынуждены отслеживать и изменять каждый такой вызов конструктора при переносе Lexi на другую платформу. А ведь кнопки – это лишь один элемент пользовательского интерфейса Lexi. Загромождение кода вызовами конструкторов для разных классов внешнего облика вызывает существенные неудобства при сопровождении. Стоит что-нибудь пропустить – и в приложении, ориентированном на платформу Mac, появится меню в стиле Motif.

Lexi необходим какой-то способ определить нужный стандарт внешнего облика для создания подходящих виджетов. При этом надо не только постараться избежать явных вызовов конструкторов, но и уметь без труда заменять весь набор виджетов. Этого можно добиться путем *абстрагирования процесса создания объекта*. Далее на примере мы продемонстрируем, что имеется в виду.

Фабрики и изготовленные классы

В Motif для создания экземпляра глифа полосы прокрутки обычно было достаточно написать следующий код на C++:

```
ScrollBar* sb = new MotifScrollBar;
```

Но такого кода надо избегать, если мы хотим минимизировать зависимость Lexi от стандарта внешнего облика. Предположим, однако, что sb инициализируется так:

```
ScrollBar* sb = guiFactory->CreateScrollBar();
```

где guiFactory – CreateScrollBar объект класса MotifFactory. Операция возвращает новый экземпляр подходящего подкласса ScrollBar, который соответствует желательному варианту внешнего облика, в нашем случае – Motif. С точки зрения клиентов результат тот же самый, что и при прямом обращении к конструктору MotifScrollBar. Но есть и существенное отличие: нигде в коде больше не упоминается имя Motif. Объект guiFactory абстрагирует процесс создания не только полос прокрутки для Motif, но и любых других. Более того, guiFactory не ограничен изготовлением только полос прокрутки. Его можно применять для производства любых виджетов, включая кнопки, поля ввода, меню и т.д.

Все это стало возможным, поскольку MotifFactory является подклассом GUIFactory – абстрактного класса, который определяет общий интерфейс для создания глифов-виджетов. В нем есть такие операции, как CreateScrollBar и CreateButton, для инстанцирования различных видов виджетов. Подклассы GUIFactory реализуют эти операции, возвращая глифы вроде MotifScrollBar и PMButton, которые имеют нужный внешний облик и поведение. На рис. 2.9 показана иерархия классов для объектов guiFactory.

Мы говорим, что фабрики *изготавливают* объекты. Продукты, изготовленные фабриками, связаны друг с другом; в нашем случае все такие продукты – это виджеты, имеющие один и тот же внешний облик. На рис. 2.10 показаны некоторые классы, необходимые для того, чтобы фабрика могла изготавливать глифы-виджеты.

Экземпляр класса GUIFactory может быть создан любым способом. Переменная guiFactory может быть глобальной или статическим членом хорошо известного класса или даже локальной, если весь пользовательский интерфейс создается внутри одного класса или функции. Существует специальный паттерн проектирования **одиночка**, предназначенный для работы с такого рода объектами, существующими в единственном экземпляре. Важно, однако, чтобы фабрика guiFactory была инициализирована до того, как начнет использоваться для производства объектов, но после того, как стало известно, какой внешний облик нужен.

Когда вариант внешнего облика известен на этапе компиляции, то guiFactory можно инициализировать простым присваиванием в начале программы:

```
GUIFactory* guiFactory = new MotifFactory;
```

Если же пользователю разрешается задавать внешний облик с помощью строки-параметра при запуске, то код создания фабрики мог бы выглядеть так:

```
GUIFactory* guiFactory;
const char* styleName = getenv("LOOK_AND_FEEL");
    // получаем это от пользователя или среды при запуске
```

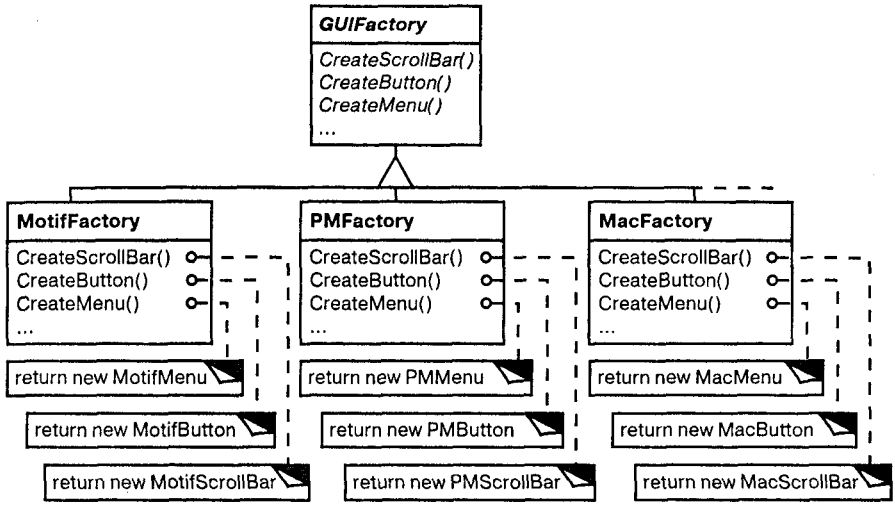

Рис. 2.9. Иерархия классов GUIFactory

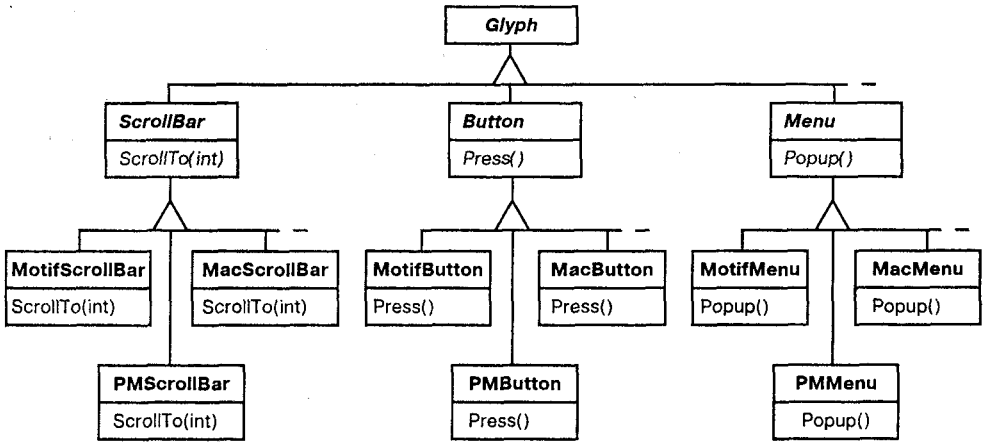

Рис. 2.10. Абстрактные классы-продукты и их конкретные подклассы

```
if (strcmp(styleName, "Motif") == 0) {
    guiFactory = new MotifFactory;

} else if (strcmp(styleName, "Presentation_Manager") == 0) {
    guiFactory = new PMFactory;

} else {
    guiFactory = new DefaultGUIFactory;
}
```

Есть и другие способы выбора фабрики во время выполнения. Например, можно было бы вести реестр, в котором символьные строки отображаются на объекты фабрик. Это позволяет зарегистрировать экземпляр новой фабрики, не меняя существующий код, как то требуется при предыдущем подходе. И нет нужды связывать с приложением код фабрик для всех платформ. Это существенно, поскольку связать код для `MotifFactory` с приложением, работающим на платформе, где Motif не поддерживается, может оказаться невозможным.

Важно, впрочем, лишь то, что после конфигурации приложения для работы с конкретной фабрикой объектов, мы получаем нужный внешний облик. Если впоследствии мы изменим решение, то сможем инициализировать `guiFactory` по-другому, чтобы изменить внешний облик, а затем динамически перестроим интерфейс.

Паттерн абстрактная фабрика

Фабрики и их продукция – вот ключевые участники паттерна абстрактная фабрика. Этот паттерн может создавать семейства объектов, не инстанцируя классы явно. Применять его лучше всего, когда число и общий вид изготавливаемых объектов остаются постоянными, но между конкретными семействами продуктов имеются различия. Выбор того или иного семейства осуществляется путем инстанцирования конкретной фабрики, после чего она используется для создания всех объектов. Подставив вместо одной фабрики другую, мы можем заменить все семейство объектов целиком. В паттерне абстрактная фабрика акцент делается на создании *семейств* объектов, и это отличает его от других порождающих паттернов, создающих только один какой-то вид объектов.

2.6. Поддержка нескольких оконных систем

Как должно выглядеть приложение – это лишь один из многих вопросов, встающих при переносе приложения на новую платформу. Еще одна проблема из той же серии – оконная среда, в которой работает Lexi. Данная среда создает иллюзию наличия нескольких перекрывающихся окон на одном растровом дисплее. Она распределяет между окнами площадь экрана и направляет им события клавиатуры и мыши. Сегодня существует несколько широко распространенных и во многом не совместимых между собой оконных систем (например, Macintosh, Presentation Manager, Windows, X). Мы хотели бы, чтобы Lexi работал в любой оконной среде по тем же причинам, по которым мы поддерживаем несколько стандартов внешнего облика.

Можно ли воспользоваться абстрактной фабрикой?

На первый взгляд представляется, что перед нами еще одна возможность применить паттерн абстрактная фабрика. Но ограничения, связанные с переносом на другие оконные системы, существенно отличаются от тех, что накладывают независимость от внешнего облика.

Применяя паттерн абстрактная фабрика, мы предполагали, что удастся определить конкретный класс глифов-виджетов для каждого стандарта внешнего облика. Это означало, что можно будет произвести конкретный класс для данного