стандарта (например, MotifScrollBar и MacScrollBar) от абстрактного класса (допустим, ScrollBar). Предположим, однако, что у нас уже есть несколько иерархий классов, полученных от разных поставщиков, – по одной для каждого стандарта. Маловероятно, что данные иерархии будут совместимы между собой. Поэтому отсутствуют общие абстрактные изготавливаемые классы для каждого вида виджетов (ScrollBar, Button, Menu и т.д.). А без них фабрика классов работать не может. Необходимо, чтобы иерархии виджетов имели единый набор абстрактных интерфейсов. Только тогда удастся правильно объявить операции Create... в интерфейсе абстрактной фабрики.

Для виджетов мы решили эту проблему, разработав собственные абстрактные и конкретные изготавливаемые классы. Теперь сталкиваемся с аналогичной трудностью при попытке заставить Lexi работать во всех существующих оконных средах. Именно разные среды имеют несовместимые интерфейсы программирования. Но на этот раз все сложнее, поскольку мы не можем себе позволить реализовать собственную нестандартную оконную систему.

Однако спасительный выход все же есть. Как и стандарты на внешний облик, интерфейсы оконных систем не так уж радикально отличаются друг от друга, ибо все они предназначены примерно для одних и тех же целей. Нам нужен унифицированный набор оконных абстракций, которым можно закрыть любую конкретную реализацию оконной системы.

Инкапсуляция зависимостей от реализации

В разделе 2.2 мы ввели класс Window для отображения на экране глифа или структуры, состоящей из глифов. Ничего не говорилось о том, с какой оконной системой работает этот объект, поскольку в действительности он вообще не связан ни с одной системой. Класс Window инкапсулирует понятие окна в любой оконной системе:

- операции для отрисовки базовых геометрических фигур;
- возможность свернуть и развернуть окно;

Таблица 2.3. Интерфейс класса Window

Обязанность	Операции

управление окнами	`virtual void Redraw()`
	`virtual void Raise()`
	`virtual void Lower()`
	`virtual void Iconify()`
	`virtual void Deiconify()`
	`...`

графика	`virtual void DrawLine(...)`
	`virtual void DrawRect(...)`
	`virtual void DrawPolygon(...)`
	`virtual void DrawText(...)`
	`...`

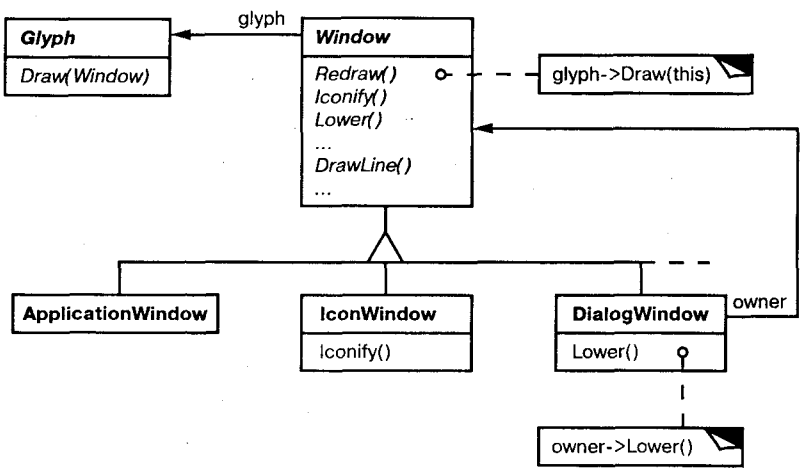
- возможность изменить собственные размеры;
- возможность при необходимости отобразить свое содержимое, например при развертывании или открытии ранее перекрытой части окна.

Класс `Window` должен покрывать функциональность окон, которая имеется в различных оконных системах. Рассмотрим два крайних подхода:

- *пересечение функциональности*. Интерфейс класса `Window` предоставляет только операции, общие для *всех* оконных систем. Однако в результате мы получаем интерфейс не богаче, чем в самой слабой из рассматриваемых систем. Мы не можем воспользоваться более развитыми средствами, даже если их поддерживает большинство оконных систем (но не все);
- *объединение функциональности*. Создается интерфейс, который включает возможности *всех* существующих систем. Здесь нас подстерегает опасность получить чрезмерно громоздкий и внутренне не согласованный интерфейс. Кроме того, нам придется изменять его (а вместе с ним и Lexi) всякий раз, как только производитель переработает интерфейс своей оконной системы.

Ни то, ни другое решение «в чистом виде» не годятся, поэтому мы выберем компромиссное. Класс `Window` будет предоставлять удобный интерфейс, поддерживающий наиболее популярные возможности оконных систем. Поскольку редактор Lexi будет работать с классом `Window` напрямую, этот класс должен поддерживать и сущности, о которых Lexi известно, то есть глифы. Это означает, что интерфейс класса `Window` должен включать базовый набор графических операций, позволяющий глифам отображать себя в окне. В табл. 2.3 перечислены некоторые операции из интерфейса класса `Window`.

`Window` – это абстрактный класс. Его конкретные подклассы поддерживают различные виды окон, с которыми имеет дело пользователь. Например, окна приложений, сообщений, значки – это все окна, но свойства у них разные. Для учета таких различий мы можем определить подклассы `ApplicationWindow`, `IconWindow` и `DialogWindow`. Возникающая иерархия позволяет таким приложениям, как



Lexi, создать унифицированную, интуитивно понятную абстракцию окна, не зависящую от оконной системы конкретного поставщика.

Итак, мы определили оконный интерфейс, с которым будет работать Lexi. Но где же в нем место для реальной платформенно-зависимой оконной системы? Если мы не собираемся реализовывать собственную оконную систему, то в каком-то месте наша абстракция окна должна быть выражена в терминах целевой системы. Но где именно?

Можно было бы реализовать несколько вариантов класса `Window` и его подклассов – по одному для каждой оконной среды. Выбор нужного варианта производится при сборке Lexi для данной платформы. Но представьте себе, с чем вы столкнетесь при сопровождении, если придется отслеживать множество разных классов с одним и тем же именем `Window`, но реализованных для разных оконных систем. Вместо этого мы могли бы создать зависящие от реализации подклассы каждого класса в иерархии `Window`, но закончилось бы это тем же самым комбинаторным ростом числа классов, о котором уже говорилось при попытке добавить элементы оформления. Кроме того, оба решения недостаточно гибки, чтобы можно было перейти на другую оконную систему уже после компиляции программы. Поэтому придется поддерживать несколько разных исполняемых файлов.

Ни тот, ни другой вариант не выглядят привлекательно, но что еще можно сделать? То же самое, что мы сделали для форматирования и декорирования, – *инкапсулировать изменяющуюся сущность*. В этом случае изменчивой частью является реализация оконной системы. Если инкапсулировать функциональность оконной системы в объекте, то удастся реализовать свой класс `Window` и его подклассы в терминах интерфейса этого объекта. Более того, если такой интерфейс сможет поддержать все интересующие нас оконные системы, то не придется изменять ни `Window`, ни его подклассы при переходе на другую систему. Мы сконфигурируем оконные объекты в соответствии с требованиями нужной оконной системы, просто передав им подходящий объект, инкапсулирующий оконную систему. Это можно сделать даже во время выполнения.

Классы Window и WindowImp

Мы определим отдельную иерархию классов `WindowImp`, в которой скроем знание о различных реализациях оконных систем. `WindowImp` – это абстрактный класс для объектов, инкапсулирующих системно-зависимый код. Чтобы заставить Lexi работать в конкретной оконной системе, каждый оконный объект будем конфигурировать экземпляром того подкласса `WindowImp`, который предназначен для этой системы. На диаграмме ниже представлены отношения между иерархиями `Window` и `WindowImp`:

Скрыв реализацию в классах `WindowImp`, мы сумели избежать «засорения» классов `Window` зависимостями от оконной системы. В результате иерархия `Window` получается сравнительно компактной и стабильной. В то же время мы можем расширить иерархию реализаций, если будет нужно поддержать новую оконную систему.