класса Window. Прикладной программист не обращается к интерфейсу WindowImp непосредственно, он имеет дело только с объектами класса Window. Поэтому интерфейс WindowImp необязательно должен соответствовать представлению программиста о мире, как то было в случае с иерархией и интерфейсом класса Window. Интерфейс WindowImp может более точно отражать сущности, которые в действительности предоставляют оконные системы, со всеми их особенностями. Он может быть ближе к идее пересечения или объединения функциональности – в зависимости от требований к целевой оконной системе.

Важно понимать, что интерфейс класса Window призван обслуживать интересы прикладного программиста, тогда как интерфейс класса WindowImp в большей степени ориентирован на оконные системы. Разделение функциональности окон между иерархиями Window и WindowImp позволяет нам независимо реализовывать и специализировать их интерфейсы. Объекты из этих иерархий взаимодействуют, позволяя Lexi работать без изменений в нескольких оконных системах.

Отношение иерархий Window и WindowImp являет собой пример паттерна мост. Идея его создания заключалась в том, чтобы предоставить возможность совместной работы отдельным иерархиям классов, даже в случае их раздельного эволюционирования. Критерии разработки, которыми мы руководствовались, заставили нас создать две различные иерархии классов: одну, поддерживающую логическое понятие окон, и другую для хранения промежуточных вариантов окон. Паттерн мост позволяет нам сохранять и совершенствовать наши логические абстракции управления окнами без необходимости привлечения программно-зависимого кода и наоборот.

2.7. Операции пользователя

Часть функциональности Lexi доступна через WYSIWYG-представление документа. Вы вводите и удаляете текст, перемещаете точку вставки и выбираете участки текста, просто указывая и щелкая мышью или нажимая клавиши. Другая часть функциональности доступна через выпадающие меню, кнопки и клавиши-ускорители. К этой категории относятся такие операции:

- создание нового документа;
- открытие, сохранение и печать существующего документа;
- вырезание выбранной части документа и вставка ее в другое место;
- изменение шрифта и стиля выбранного текста;
- изменение форматирования текста, например, установка режима выравнивания;
- завершение приложения и др.

Lexi предоставляет для этих операций различные пользовательские интерфейсы. Но мы не хотим ассоциировать конкретную операцию с определенным пользовательским интерфейсом, поскольку для выполнения одной и той же операции желательно иметь несколько интерфейсов (например, листать страницы можно с помощью кнопки или выбора из меню). Кроме того, в будущем может понадобиться изменить интерфейс.

Далее. Операции реализованы в разных классах. Нам как разработчикам хотелось бы получать доступ к функциональности классов, не создавая зависимостей между классами, отвечающими за реализацию и за пользовательский интерфейс. В противном случае получится сильно связанный код, который будет трудно понять, модернизировать и сопровождать.

Ситуация осложняется еще и тем, что Lexi должен поддерживать операции отмены и повтора[1] большинства, но *не всех* функций. Точнее, желательно уметь отменять операции модификации документа (скажем, удаление), которые из-за оплошности пользователя могут привести к уничтожению большого объема данных. Но не следует пытаться отменить такую операцию, как сохранение чертежа или завершение приложения. Мы также не хотели бы налагать произвольные ограничения на число уровней отмены и повтора.

Ясно, что поддержка пользовательских операций распределена по всему приложению. Задача в том, чтобы найти простой и расширяемый механизм, удовлетворяющий всем вышеизложенным требованиям.

Инкапсуляция запроса

С точки зрения проектировщика выпадающее меню – это просто еще один вид вложенных глифов. От других глифов, имеющих потомков, его отличает то, что большинство содержащихся в меню глифов каким-то образом реагирует на отпускание кнопки мыши.

Предположим, что такие реагирующие глифы являются экземплярами подкласса `MenuItem` класса `Glyph` и что свою работу они выполняют в ответ на запрос клиента[2]. Для выполнения запроса может потребоваться вызвать одну операцию одного объекта или много операций разных объектов. Возможны и промежуточные варианты.

Мы могли бы определить подкласс класса `MenuItem` для каждой операции пользователя, а затем жестко закодировать каждый подкласс для выполнения соответствующего запроса. Но вряд ли это правильно: нам не нужен подкласс `MenuItem` для каждого запроса, точно так же, как не нужен отдельный подкласс для каждой строки в выпадающем меню. Помимо всего прочего, такой подход тесно привязывает запрос к конкретному элементу пользовательского интерфейса, поэтому выполнить тот же запрос через другой интерфейс будет нелегко.

Предположим, что на последнюю страницу документа вы можете перейти, выбрав соответствующий пункт из меню или щелкнув по значку с изображением страницы в нижней части окна Lexi (для коротких документов это удобно). Если мы ассоциируем запрос с подклассом `MenuItem` с помощью наследования, то должны сделать то же самое и для значка страницы, и для любого другого виджета, который способен послать подобный запрос. В результате число классов будет примерно равно произведению числа типов виджетов на число запросов.

---

[1] Под повтором (redo) понимается выполнение только что отмененной операции.

[2] Концептуально клиентом является пользователь Lexi, но на самом деле это просто какой-то другой объект (например, диспетчер событий), который управляет обработкой ввода пользователя.