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.

Нам не хватает механизма параметризации пунктов меню запросами, которые они должны выполнять. Таким способом удалось бы избежать разрастания числа подклассов и обеспечить большую гибкость во время выполнения. Параметризовать `MenuItem` можно вызываемой функцией, но это решение неполно по трем причинам:

- в нем не учитывается проблема отмены/повтора;
- с функцией трудно ассоциировать состояние. Например, функция, изменяющая шрифт, должна «знать», какой именно это шрифт;
- функции с трудом поддаются расширению, а использовать их части тоже затруднительно.

Поэтому лучше параметризовать пункты меню *объектом*, а не функцией. Тогда мы сможем прибегнуть к механизму наследования для расширения и повторного использования реализации запроса. Кроме того, у нас появляется место для сохранения состояния и возможность реализовать отмену и повтор. Вот еще один пример инкапсуляции изменяющейся сущности, в данном случае – запроса. Каждый запрос мы инкапсулируем в объект-команду.

Класс Command и его подклассы

Сначала определим абстрактный класс `Command`, который будет предоставлять интерфейс для выдачи запроса. Базовый интерфейс включает всего одну абстрактную операцию `Execute`. Подклассы `Command` по-разному реализуют эту операцию для выполнения запросов. Некоторые подклассы могут частично или полностью делегировать работу другим объектам, а остальные выполняют запрос сами (см. рис. 2.11). Однако для запрашивающего объект `Command` – это всего лишь объект `Command`, все они рассматриваются одинаково.

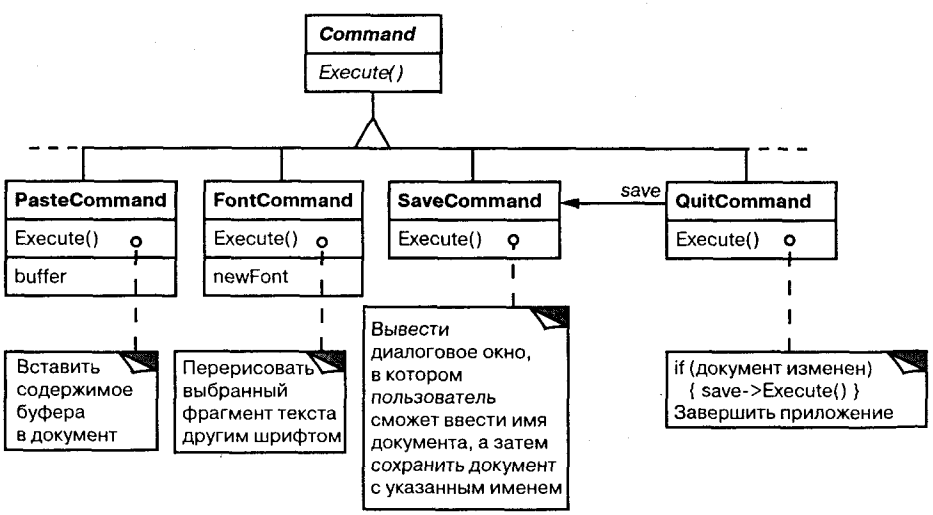

Рис. 2.11. Часть иерархии класса Command