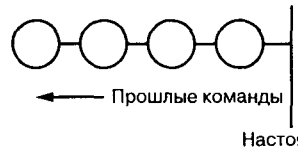
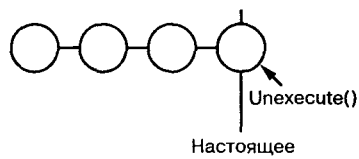
История команд

Последний шаг в направлении поддержки отмены и повтора с произвольным числом уровней – определение *истории команд*, то есть списка ранее выполненных или отмененных команд. Концептуально история команд выглядит так:

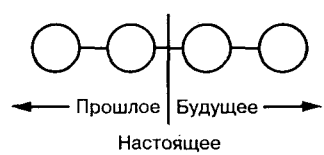


Каждый кружок представляет один объект `Command`. В данном случае пользователь выполнил четыре команды. Линия с пометкой «настоящее» показывает самую последнюю выполненную (или отмененную) команду.

Для того чтобы отменить последнюю команду, мы просто вызываем операцию `Unexecute` для самой последней команды:

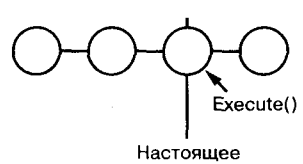


После отмены команды сдвигаем линию «настоящее» на одну команду влево. Если пользователь выполнит еще одну отмену, то произойдет откат на один шаг (см. рис. ниже).

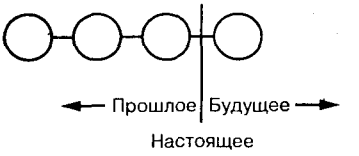


Видно, что за счет простого повторения процедуры мы получаем произвольное число уровней отмены, ограниченное лишь длиной списка истории команд.

Чтобы повторить только что отмененную команду, произведем обратные действия. Команды справа от линии «настоящее» – те, что могут быть повторены в будущем. Для повтора последней отмененной команды мы вызываем операцию `Execute` для последней команды справа от линии «настоящее»:



Затем мы сдвигаем линию «настоящее» так, чтобы следующий повтор вызвал операцию `Execute` для следующей команды «в области будущего».



Разумеется, если следующая операция – это не повтор, а отмена, то команда слева от линии «настоящее» будет отменена. Таким образом, пользователь может перемещаться в обоих направлениях, чтобы исправить ошибки.

Паттерн команда

Команды Lexi – это пример применения паттерна команда, описывающего, как инкапсулировать запрос. Этот паттерн предписывает единообразный интерфейс для выдачи запросов, с помощью которого можно сконфигурировать клиенты для обработки разных запросов. Интерфейс изолирует клиента от реализации запроса. Команда может полностью или частично делегировать реализацию запроса другим объектам либо выполнять данную операцию самостоятельно. Это идеальное решение для приложений типа Lexi, которые должны предоставлять централизованный доступ к функциональности, разбросанной по разным частям программы. Данный паттерн предлагает также механизмы отмены и повтора, надстроенные над базовым интерфейсом класса `Command`.

2.8. Проверка правописания и расстановка переносов

Наша последняя задача связана с анализом текста, точнее, с проверкой правописания и нахождением мест, где можно поставить перенос для улучшения форматирования.

Ограничения здесь аналогичны тем, о которых уже говорилось при обсуждении форматирования в разделе 2.3. Как и в случае с разбиением на строки, есть много способов реализовать поиск орфографических ошибок и вычисление точек переноса. Поэтому и здесь планировалась поддержка нескольких алгоритмов. Пользователь сможет выбрать тот алгоритм, который его больше устраивает по соотношению потребляемой памяти, скорости и качеству. Добавление новых алгоритмов тоже должно реализовываться просто.

Также мы хотим избежать жесткой привязки этой информации к структуре документа. В данном случае такая цель даже более важна, чем при форматировании, поскольку проверка правописания и расстановка переносов – это лишь два вида анализа текста, которые Lexi мог бы поддерживать. Со временем мы собираемся расширить аналитические возможности Lexi. Мы могли бы добавить поиск, подсчет слов, средства вычислений для суммирования значений в таблице, проверку грамматики и т.д. Но мы не хотим изменять класс `Glyph` и все его подклассы при каждом добавлении такого рода функциональности.

У этой задачи есть две стороны: доступ к анализируемой информации, которая разбросана по разным глифам в структуре документа и собственно выполнение анализа. Рассмотрим их по отдельности.