БИБЛИОТЕКА ПРОГРАММИСТА

Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес

Приемы объектно-ориентированного проектирования

Паттерны проектирования

- Принципы применения паттернов проектирования
- Классификация паттернов
- Различные подходы к выбору паттернов
- Каталог паттернов с их детальным описанием

Addison-Wesley
ДМК
ПИТЕР

ББК 32.973.2-018
УДК 681.3.068
П75

Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.

П75 Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2015. — 368 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-00389-6

В предлагаемой книге описываются простые и изящные решения типичных задач, возникающих в объектно-ориентированном проектировании. Паттерны появились, потому что многие разработчики искали пути повышения гибкости и степени повторного использования своих программ. Найденные решения воплощены в краткой и легко применимой на практике форме. Авторы излагают принципы использования паттернов проектирования и приводят их каталог. Таким образом книга одновременно решает две задачи. Во-первых, здесь демонстрируется роль паттернов в создании архитектуры сложных систем. Во-вторых, применяя содержащиеся в справочнике паттерны, проектировщик сможет с легкостью разрабатывать собственные приложения.

Издание предназначено как для профессиональных разработчиков, так и для программистов, осваивающих объектно-ориентированное проектирование.

6+ (Для детей старше 6 лет. В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018
УДК 681.3.068

Права на издание получены по соглашению с Addison-Wesley Longman.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 0-201-63361-2 (англ.)
ISBN 978-5-496-00389-6

Original English language Edition Copyright © 1995 by Addison Wesley Longman, Inc.
© Перевод на русский язык, А. Слинкин, 2015
© Серия, оформление, ООО Издательство «Питер», 2015
Оригинал-макет подготовлен издательством «ДМК Пресс»

Содержание

Предисловие 10
Глава 1. Введение в паттерны проектирования 15
1.1. Что такое паттерн проектирования 16
1.2. Паттерны проектирования в схеме MVC в языке Smalltalk 18
1.3. Описание паттернов проектирования 20
1.4. Каталог паттернов проектирования 22
1.5. Организация каталога 24
1.6. Как решать задачи проектирования с помощью паттернов 25
Поиск подходящих объектов 25
Определение степени детализации объекта 27
Специфицирование интерфейсов объекта 27
Специфицирование реализации объектов 29
Механизмы повторного использования 32
Сравнение структур времени выполнения и времени компиляции 37
Проектирование с учетом будущих изменений 38
1.7. Как выбирать паттерн проектирования 43
1.8. Как пользоваться паттерном проектирования 44
Глава 2. Проектирование редактора документов 46
2.1. Задачи проектирования 46
2.2. Структура документа 48
Рекурсивная композиция 49
Глифы 51
Паттерн компоновщик 53
2.3. Форматирование 53
Инкапсуляция алгоритма форматирования 54
Классы Compositor и Composition 54
Стратегия 56
2.4. Оформление пользовательского интерфейса 56
Прозрачное обрамление 57
Моноглиф 58
Паттерн декоратор 60
2.5. Поддержка нескольких стандартов внешнего облика 60
Абстрагирование создания объекта 61
Фабрики и изготовленные классы 61
Паттерн абстрактная фабрика 64

2.6. Поддержка нескольких оконных систем ... 64
Можно ли воспользоваться абстрактной фабрикой? ... 64
Инкапсуляция зависимостей от реализации ... 65
Классы Window и WindowImp ... 67
Подклассы WindowImp ... 68
Конфигурирование класса Window с помощью WindowImp ... 70
Паттерн мост ... 70
2.7. Операции пользователя ... 71
Инкапсуляция запроса ... 72
Класс Command и его подклассы ... 73
Отмена операций ... 74
История команд ... 75
Паттерн команда ... 76
2.8. Проверка правописания и расстановка переносов ... 76
Доступ к распределенной информации ... 77
Инкапсуляция доступа и порядка обхода ... 77
Класс Iterator и его подклассы ... 78
Паттерн итератор ... 81
Обход и действия, выполняемые при обходе ... 81
Класс Visitor и его подклассы ... 86
Паттерн посетитель ... 87
2.9. Резюме ... 88
Глава 3. Порождающие паттерны ... 89
Паттерн Abstract Factory ... 93
Паттерн Builder ... 102
Паттерн Factory Method ... 111
Паттерн Prototype ... 121
Паттерн Singleton ... 130
Обсуждение порождающих паттернов ... 138
Глава 4. Структурные паттерны ... 140
Паттерн Adapter ... 141
Паттерн Bridge ... 152
Паттерн Composite ... 162
Паттерн Decorator ... 173
Паттерн Facade ... 183
Паттерн Flyweight ... 191
Паттерн Proxy ... 203
Обсуждение структурных паттернов ... 213
Адаптер и мост ... 213
Компоновщик, декоратор и заместитель ... 214
Глава 5. Паттерны поведения ... 216
Паттерн Chain of Responsibility ... 217
Паттерн Command ... 227

Паттерн Interpreter ... 236
Паттерн Iterator ... 249
Паттерн Mediator ... 263
Паттерн Memento ... 272
Паттерн Observer ... 280
Паттерн State ... 291
Паттерн Strategy ... 300
Паттерн Template Method ... 309
Паттерн Visitor ... 314
Обсуждение паттернов поведения ... 328
Инкапсуляция вариаций ... 328
Объекты как аргументы ... 328
Должен ли обмен информацией быть инкапсулированным или распределенным ... 329
Разделение получателей и отправителей ... 330
Резюме ... 332
Глава 6. Заключение ... 333
6.1. Чего ожидать от паттернов проектирования ... 333
Единый словарь проектирования ... 333
Помощь при документировании и изучении ... 334
Дополнение существующих методов ... 334
Цель реорганизации ... 335
6.2. Краткая история ... 336
6.3. Проектировщики паттернов ... 337
Языки паттернов Александра ... 338
Паттерны в программном обеспечении ... 339
6.4. Приглашение ... 339
6.5. На прощание ... 340
Приложение A. Глоссарий ... 341
Приложение B. Объяснение нотации ... 344
B.1. Диаграмма классов ... 344
B.2. Диаграмма объектов ... 345
B.3. Диаграмма взаимодействий ... 346
Приложение C. Базовые классы ... 348
C.1. List ... 348
C.2. Iterator ... 350
C.3. ListIterator ... 350
C.4. Point ... 351
C.5. Rect ... 351
Библиография ... 353
Алфавитный указатель ... 359

Глава 2. Проектирование редактора документов

В данной главе рассматривается применение паттернов на примере проектирования визуального редактора документов Lexi[1], построенного по принципу «что видишь, то и получаешь» (WYSIWYG). Мы увидим, как с помощью паттернов можно решать проблемы проектирования, характерные для Lexi и аналогичных приложений. Здесь описывается опыт работы с восемью паттернами.

На рис. 2.1 изображен пользовательский интерфейс редактора Lexi. WYSIWYG-представление документа занимает большую прямоугольную область в центре. В документе могут произвольно сочетаться текст и графика, отформатированные разными способами. Вокруг документа – привычные выпадающие меню и полосы прокрутки, а также значки с номерами для перехода на нужную страницу документа.

2.1. Задачи проектирования

Рассмотрим семь задач, характерных для дизайна Lexi:

- *структура документа*. Выбор внутреннего представления документа отражается практически на всех аспектах дизайна. Для редактирования, форматирования, отображения и анализа текста необходимо уметь обходить это представление. Способ организации информации играет решающую роль при дальнейшем проектировании;
- *форматирование*. Как в Lexi организованы текст и графика в виде строк и колонок? Какие объекты отвечают за реализацию стратегий форматирования? Взаимодействие данных стратегий с внутренним представлением документа;
- *создание привлекательного интерфейса пользователя*. В состав пользовательского интерфейса Lexi входят полосы прокрутки, рамки и оттененные выпадающие меню. Вполне вероятно, что количество и состав элементов интерфейса будут изменяться по мере его развития. Поэтому важно иметь возможность легко добавлять и удалять элементы оформления, не затрагивая приложение;
- *поддержка стандартов внешнего облика программы*. Lexi должен без серьезной модификации адаптироваться к стандартам внешнего облика программ, например, таким как Motif или Presentation Manager (PM);

---

[1] Дизайн Lexi основан на программе Doc – текстового редактора, разработанного Кальдером [CL92].

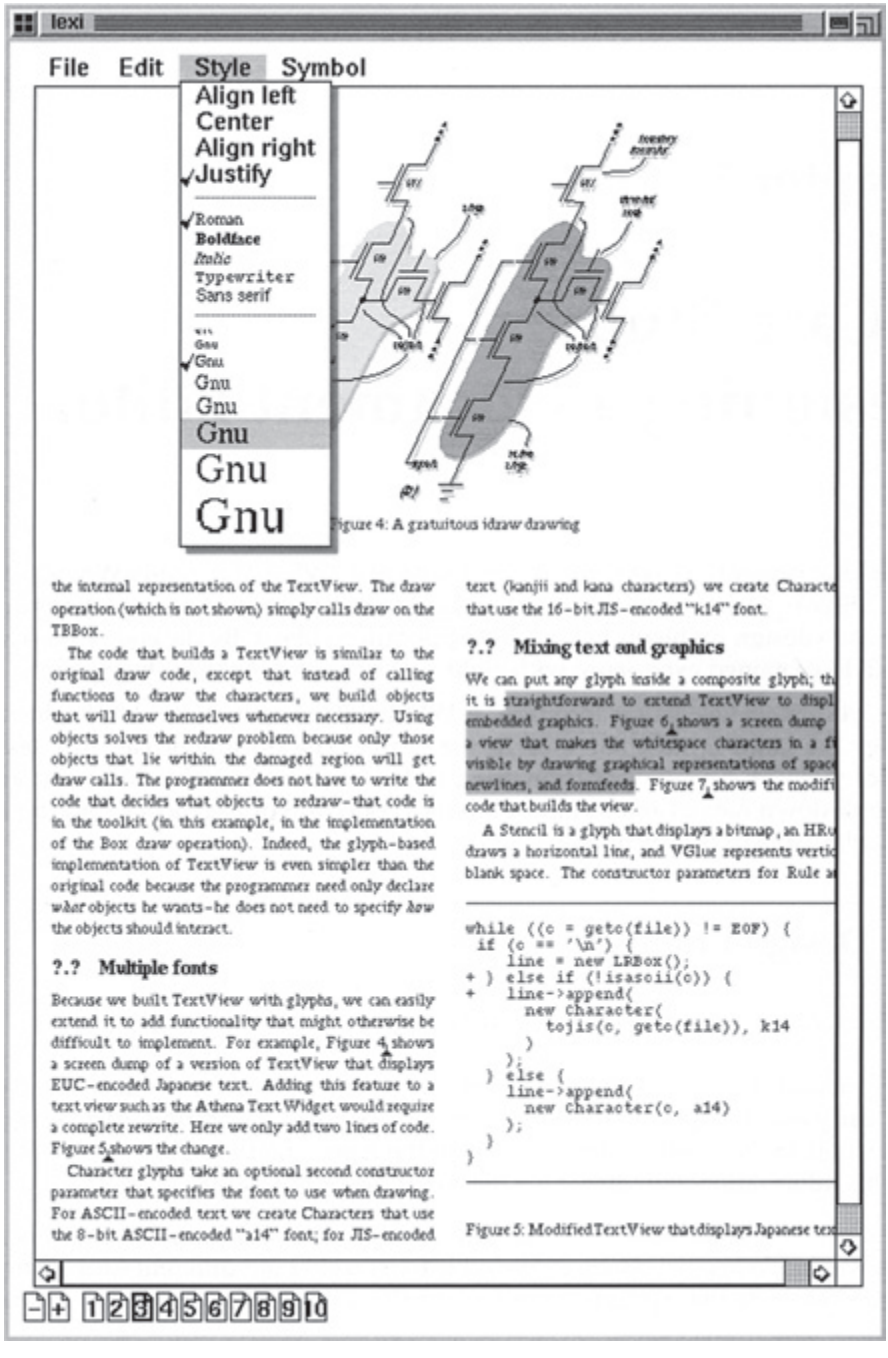

Рис. 2.1. Пользовательский интерфейс Lexi

- *поддержка оконных систем*. В оконных системах стандарты внешнего облика обычно различаются. По возможности дизайн Lexi должен быть независимым от оконной системы;
- *операции пользователя*. Пользователи управляют работой Lexi с помощью элементов интерфейса, в том числе кнопок и выпадающих меню. Функции, которые вызываются из интерфейса, разбросаны по всей программе. Разработать единообразный механизм для доступа к таким «рассеянным» функциям и для отмены уже выполненных операций довольно трудно;
- *проверка правописания и расстановка переносов*. Поддержка в Lexi таких аналитических операций, как проверка правописания и определение мест переноса. Как минимизировать число классов, которые придется модифицировать при добавлении новой аналитической операции?

Ниже обсуждаются указанные проблемы проектирования. Для каждой из них определены некоторые цели и ограничения на способы их достижения. Прежде чем предлагать решение, мы подробно остановимся на целях и ограничениях. На примере проблемы и ее решения демонстрируется применение одного или нескольких паттернов проектирования. Обсуждение каждой проблемы завершается краткой характеристикой паттерна.

2.2. Структура документа

Документ – это всего лишь организованное некоторым способом множество базовых графических элементов: символов, линий, многоугольников и других геометрических фигур. Все они несут в себе полную информацию о содержании документа. И все же автор часто представляет себе эти элементы не в графическом виде, а в терминах физической структуры документа – строк, колонок, рисунков, таблиц и других подструктур.[1] Эти подструктуры, в свою очередь, составлены из более мелких и т.д.

Пользовательский интерфейс Lexi должен позволять пользователям непосредственно манипулировать такими подструктурами. Например, пользователю следует дать возможность обращаться с диаграммой как с неделимой единицей, а не как с набором отдельных графических примитивов, предоставить средства ссылаться на таблицу как на единое целое, а не как на неструктурированное хранилище текста и графики. Это делает интерфейс простым и интуитивно понятным. Чтобы придать реализации Lexi аналогичные свойства, мы выберем такое внутреннее представление, которое в точности соответствует физической структуре документа.

В частности, внутреннее представление должно поддерживать:

- отслеживание физической структуры документа, то есть разбиение текста и графики на строки, колонки, таблицы и т.д.;

---

[1] Авторы часто рассматривают документы и в терминах их *логической* структуры: предложений, абзацев, разделов, подразделов и глав. Чтобы не слишком усложнять пример, мы не будем явно хранить во внутреннем представлении информацию о логической структуре. Но то проектное решение, которое мы опишем, вполне пригодно для представления и такой информации.

- генерирование визуального представления документа;
- отображение позиций экрана на элементы внутреннего представления. Это позволит определить, что имел в виду пользователь, когда указал на что-то в визуальном представлении.

Помимо данных целей имеются и ограничения. Во-первых, текст и графику следует трактовать единообразно. Интерфейс приложения должен позволять свободно помещать текст внутрь графики и наоборот. Не следует считать графику частным случаем текста или текст – частным случаем графики, поскольку это в конечном итоге привело бы к появлению избыточных механизмов форматирования и манипулирования. Одного набора механизмов должно хватить и для текста, и для графики.

Во-вторых, в нашей реализации не может быть различий во внутреннем представлении отдельного элемента и группы элементов. При одинаковой работе Lexi с простыми и сложными элементами можно будет создавать документы со структурой любой сложности. Например, десятым элементом на пересечении пятой строки и второй колонки мог бы быть как один символ, так и сложно устроенная диаграмма со многими внутренними компонентами. Но, коль скоро мы уверены, что этот элемент имеет возможность изображать себя на экране и сообщать свои размеры, его внутренняя сложность не имеет никакого отношения к тому, как и в каком месте страницы он появляется.

Однако второе ограничение противоречит необходимости анализировать текст на предмет выявления орфографических ошибок и расстановки переносов. Во многих случаях нам безразлично, является ли элемент строки простым или сложным объектом. Но иногда вид анализа зависит от анализируемого объекта. Так, вряд ли имеет смысл проверять орфографию многоугольника или пытаться переносить его с одной строки на другую. При проектировании внутреннего представлении надо учитывать эти и другие потенциально конфликтующие ограничения.

Рекурсивная композиция

На практике для представления иерархически структурированной информации часто применяется прием, называемый *рекурсивной композицией*. Он позволяет строить все более сложные элементы из простых. Рекурсивная композиция дает нам способ составить документ из простых графических элементов. Сначала мы можем линейно расположить множество символов и графики слева направо для формирования одной строки документа. Затем несколько строк можно объединить в колонку, несколько колонок – в страницу и т.д. (см. рис. 2.2.).

Данную физическую структуру можно представить, введя отдельный объект для каждого существенного элемента. К таковым относятся не только видимые элементы вроде символов и графики, но и структурные элементы – строки и колонки. В результате получается структура объекта, изображенная на рис. 2.3.

Представляя объектом каждый символ и графический элемент документа, мы обеспечиваем гибкость на самых нижних уровнях дизайна Lexi. С точки зрения отображения, форматирования и вкладывания друг в друга единообразно трактуются текст и графика. Мы сможем расширить Lexi для поддержки новых наборов

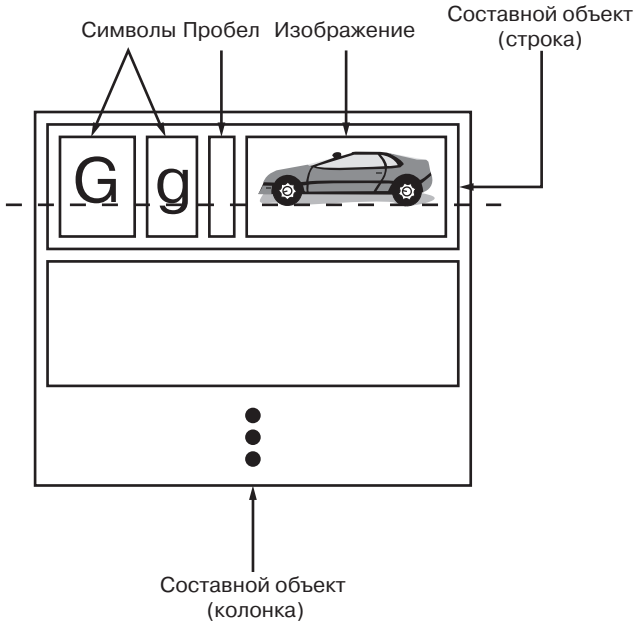

Рис. 2.2. Рекурсивная композиция текста и графики

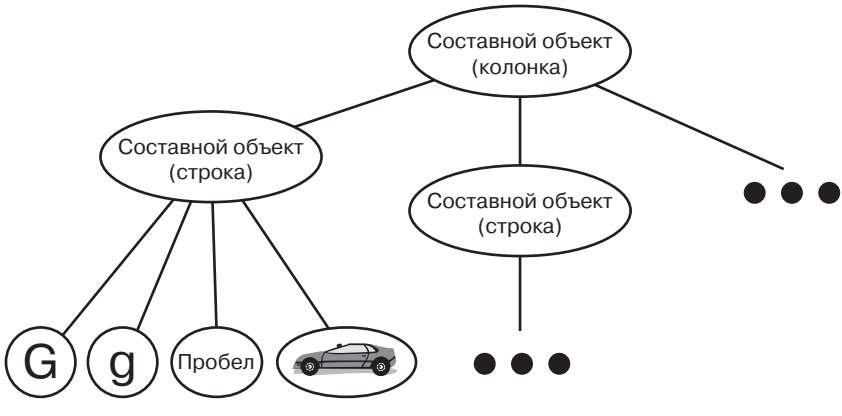

Рис. 2.3. Структура объекта для рекурсивной композиции текста и графики

символов, не затрагивая никаких других функций. Объектная структура Lexi точно отражает физическую структуру документа.

У описанного подхода есть два важных следствия. Первое очевидно: для объектов нужны соответствующие классы. Второе, менее очевидное, состоит в том, что у этих классов должны быть совместимые интерфейсы, поскольку мы хотим унифицировать работу с ними. Для обеспечения совместимости интерфейсов в таком языке, как C++, применяется наследование.

Глифы

Абстрактный класс `Glyph` (глиф) определяется для всех объектов, которые могут присутствовать в структуре документа[1]. Его подклассы определяют как примитивные графические элементы (скажем, символы и изображения), так и структурные элементы (строки и колонки). На рис. 2.4 изображена достаточно обширная часть иерархии класса `Glyph`, а в таблице 2.1 более подробно представлен базовый интерфейс этого класса в нотации C++[2].

Таблица 2.1. Базовый интерфейс класса Glyph

Обязанность	Операции
внешнее представление	`virtual void Draw(Window*)`
	`virtual void Bounds(Rect&)`
обнаружение точки воздействия	`virtual bool Intersects(const Point&)`
структура	`virtual void Insert(Glyph*, int)`
	`virtual void Remove(Glyph*)`
	`virtual Glyph* Child(int)`
	`virtual Glyph* Parent()`

У глифов есть три основные функции. Они имеют информацию о своих предках и потомках, а также о том, как нарисовать себя на экране и сколько места они занимают.

Подклассы класса `Glyph` переопределяют операцию `Draw`, выполняющую перерисовку себя в окне. При вызове `Draw` ей передается ссылка на объект `Window`. В классе `Window` определены графические операции для прорисовки в окне на экране текста и основных геометрических фигур. Например, в подклассе `Rectangle` операция `Draw` могла определяться так:

```
void Rectangle::Draw (Window* w) {
    w->DrawRect(_x0, _y0, _x1, _y1);
}
```

где `_x0`, `_y0`, `_x1` и `_y1` – это данные-члены класса `Rectangle`, определяющие два противоположных угла прямоугольника, а `DrawRect` – операция из класса `Window`, рисующая на экране прямоугольник.

---

[1] Впервые термин «глиф» в этом контексте употребил Пол Кальдер [CL90]. В большинстве современных редакторов документов отдельные символы не представляются объектами, скорее всего, из соображений эффективности. Кальдер продемонстрировал практическую пригодность этого подхода в своей диссертации [Cal93]. Наши глифы проще предложенных им, поскольку мы для простоты ограничились строгими иерархиями. Глифы Кальдера могут использоваться совместно для уменьшения потребления памяти и, стало быть, образуют направленные ациклические графы. Для достижения того же эффекта мы можем воспользоваться паттерном Приспособленец, но оставим это в качестве упражнения читателю.

[2] Представленный здесь интерфейс намеренно сделан минимальным, чтобы не загромождать обсуждение техническими деталями. Полный интерфейс должен бы включать операции для работы с графическими атрибутами: цветами, шрифтами и преобразованиями координат, а также операции для более развитого управления потомками.

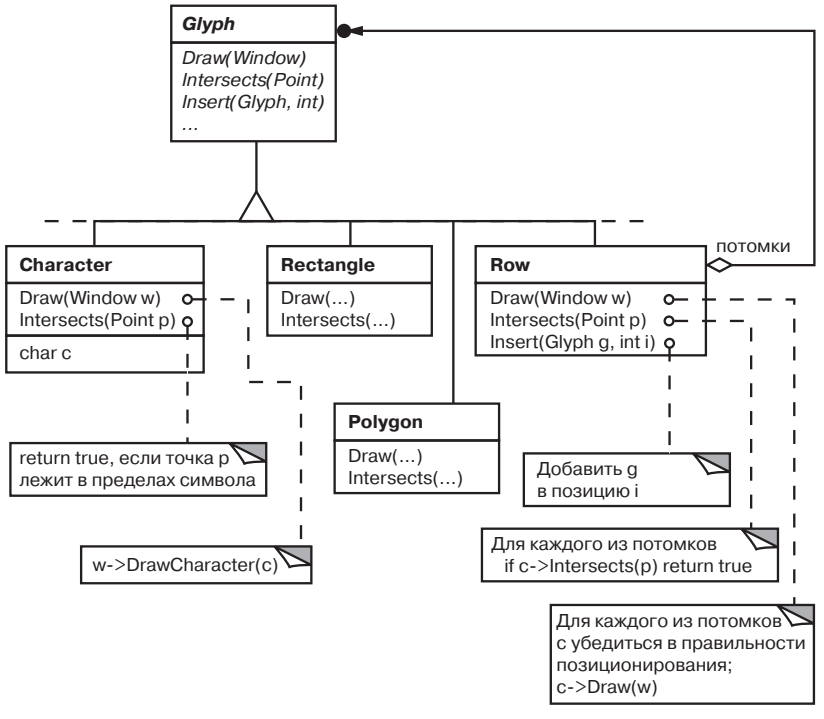

Рис. 2.4. Частичная иерархия класса Glyph

Глифу-родителю часто бывает нужно «знать», сколько места на экране занимает глиф-потомок, чтобы расположить его и остальные глифы в строке без перекрытий (как показано на рис. 2.2). Операция `Bounds` возвращает прямоугольную область, занимаемую глифом, точнее, противоположные углы наименьшего прямоугольника, содержащего глиф. В подклассах класса `Glyph` эта операция переопределена в соответствии с природой конкретного элемента.

Операция `Intersects` возвращает признак, показывающий, лежит ли заданная точка в пределах глифа. Всякий раз, когда пользователь щелкает мышью где-то в документе, Lexi вызывает эту операцию, чтобы определить, какой глиф или глифовая структура оказалась под курсором мыши. Класс `Rectangle` переопределяет эту операцию для вычисления пересечения точки с прямоугольником.

Поскольку у глифов могут быть потомки, то нам необходим единый интерфейс для добавления, удаления и обхода потомков. Например, потомки класса `Row` – это глифы, расположенные в данной строке. Операция `Insert` вставляет глиф в позицию, заданную целочисленным индексом.[1] Операция `Remove` удаляет заданный глиф, если он действительно является потомком.

---

[1] Возможно, целочисленный индекс – не лучший способ описания потомков глифа. Это зависит от структуры данных, используемой внутри глифа. Если потомки хранятся в связанном списке, то более эффективно было бы передавать указатель на элемент списка. Мы еще увидим более удачное решение проблемы индексации в разделе 2.8, когда будем обсуждать анализ документа.

Операция `Child` возвращает потомка с заданным индексом (если таковой существует). Глифы типа `Row`, у которых действительно есть потомки, должны пользоваться операцией `Child`, а не обращаться к структуре данных потомка напрямую. В таком случае при изменении структуры данных, скажем, с массива на связанный список не придется модифицировать операции вроде `Draw`, которые обходят всех потомков. Аналогично операция `Parent` предоставляет стандартный интерфейс для доступа к родителю глифа, если таковой имеется. В Lexi глифы хранят ссылку на своего родителя, а `Parent` просто возвращает эту ссылку.

Паттерн компоновщик

Рекурсивная композиция пригодна не только для документов. Мы можем воспользоваться ей для представления любых потенциально сложных иерархических структур. Паттерн компоновщик инкапсулирует сущность рекурсивной композиции объектно-ориентированным способом. Сейчас самое время обратиться к разделу об этом паттерне и изучить его, имея в виду только что рассмотренный сценарий.

2.3. Форматирование

Мы решили, как *представлять* физическую структуру документа. Далее нужно разобраться с тем, как сконструировать *конкретную* физическую структуру, соответствующую правильно отформатированному документу. Представление и форматирование – это разные аспекты проектирования. Описание внутренней структуры не дает возможности добраться до определенной подструктуры. Ответственность за это лежит в основном на Lexi. Редактор разбивает текст на строки, строки – на колонки и т.д., учитывая при этом пожелания пользователя. Так, пользователь может изменить ширину полей, размер отступа и положение точек табуляции, установить одиночный или двойной междустрочный интервал, а также задать много других параметров форматирования.[1] В процессе форматирования это учитывается.

Кстати говоря, мы сузим значение термина «форматирование», понимая под этим лишь разбиение на строки. Будем считать слова «форматирование» и «разбиение на строки взаимозаменяемыми. Обсуждаемая техника в равной мере применима и к разбиению строк на колонки, и к разбиению колонок на страницы.

Таблица 2.2. Базовый интерфейс класса Compositor

Обязанность	Операции
что форматировать	`void SetComposition(Composition*)`
когда форматировать	`virtual void Compose()`

[1] У пользователя найдется что сказать и по поводу *логической* структуры документа: предложений, абзацев, разделов, глав и т.д. *Физическая* структура в общем-то менее интересна. Большинству людей не важно, где в абзаце произошел разрыв строки, если в целом все отформатировано правильно. То же самое относится и к форматированию колонок и страниц. Таким образом, пользователи задают только высокоуровневые ограничения на физическую структуру, а Lexi выполняет их.

Инкапсуляция алгоритма форматирования

С учетом всех ограничений и деталей процесс форматирования с трудом поддается автоматизации. К этой проблеме есть много подходов, и у разных алгоритмов форматирования имеются свои сильные и слабые стороны. Поскольку Lexi – это WYSIWYG-редактор, важно соблюдать компромисс между качеством и скоростью форматирования. В общем случае желательно, чтобы редактор реагировал достаточно быстро и при этом внешний вид документа оставался приемлемым. На достижение этого компромисса влияет много факторов, и не все из них удастся установить на этапе компиляции. Например, можно предположить, что пользователь готов смириться с замедленной реакцией в обмен на лучшее качество форматирования. При таком предположении нужно было бы применять совершенно другой алгоритм форматирования. Есть также компромисс между временем и памятью, скорее, имеющий отношение к реализации: время форматирования можно уменьшить, если хранить в памяти больше информации.

Поскольку алгоритмы форматирования обычно оказываются весьма сложными, желательно, чтобы они были достаточно замкнутыми, а еще лучше – полностью независимыми от структуры документа. Оптимальный вариант – добавление нового вида глифа вовсе не затрагивает алгоритм форматирования. С другой стороны, при добавлении нового алгоритма форматирования не должно возникать необходимости в модификации существующих глифов.

Учитывая все вышесказанное, мы должны постараться спроектировать Lexi так, чтобы алгоритм форматирования легко можно было заменить, по крайней мере, на этапе компиляции, если уж не во время выполнения. Допустимо изолировать алгоритм и одновременно сделать его легко замещаемым путем инкапсулирования в объекте. Точнее, мы определим отдельную иерархию классов для объектов, инкапсулирующих алгоритмы форматирования. Для корневого класса иерархии определим интерфейс, который поддерживает широкий спектр алгоритмов, а каждый подкласс будет реализовывать этот интерфейс в виде конкретного алгоритма форматирования. Тогда удастся ввести подкласс класса `Glyph`, который будет автоматически структурировать своих потомков с помощью переданного ему объекта-алгоритма.

Классы Compositor и Composition

Мы определим класс `Compositor` (композитор) для объектов, которые могут инкапсулировать алгоритм форматирования. Интерфейс (см. табл. 2.2) позволяет объекту этого класса узнать, *какие* глифы надо форматировать и *когда*. Форматируемые композитором глифы являются потомками специального подкласса класса `Glyph`, который называется `Composition` (композиция). Композиция при создании получает объект некоторого подкласса `Compositor` (специализированный для конкретного алгоритма разбиения на строки) и в нужные моменты предписывает композитору форматировать глифы, по мере того как пользователь изменяет документ. На рис. 2.5 изображены отношения между классами `Composition` и `Compositor`.

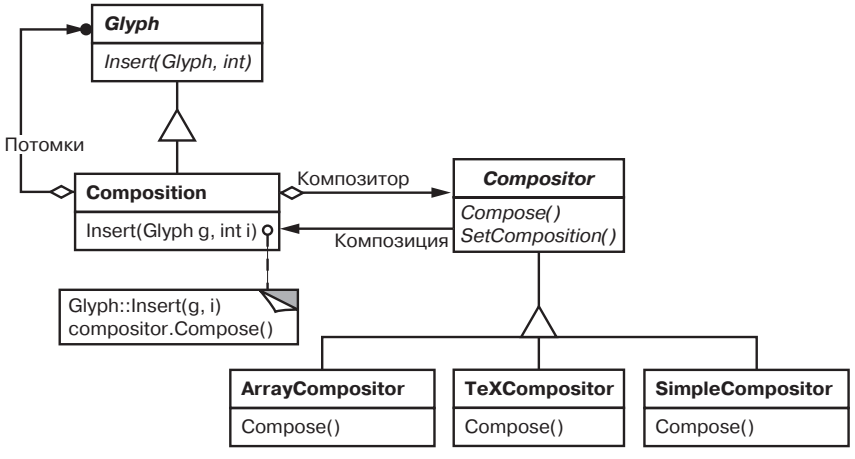

Рис. 2.5. Отношения классов Composition и Compositor

Неформатированный объект `Composition` содержит только видимые глифы, составляющие основное содержание документа. В нем нет глифов, определяющих физическую структуру документа, например `Row` и `Column`. В таком состоянии композиция находится сразу после создания и инициализации глифами, которые должна отформатировать. Во время форматирования композиция вызывает операцию `Compose` своего объекта `Compositor`. Композитор обходит всех потомков композиции и вставляет новые глифы `Row` и `Column` в соответствии со своим алгоритмом разбиения на строки.[1] На рис. 2.6 показана получающаяся объектная структура. Глифы, созданные и вставленные в эту структуру композитором, закрашены на рисунке серым цветом.

Каждый подкласс класса `Compositor` может реализовывать свой собственный алгоритм форматирования. Например, класс `SimpleCompositor` мог бы осуществлять быстрый проход, не обращая внимания на такую экзотику, как «цвет» документа. Под «хорошим цветом» понимается равномерное распределение текста и пустого пространства. Класс `TeXCompositor` мог бы реализовывать полный алгоритм TEX [Knu84], учитывающий наряду со многими другими вещами и цвет, но за счет увеличения времени форматирования.

Наличие классов `Compositor` и `Composition` обеспечивает четкое отделение кода, поддерживающего физическую структуру документа, от кода различных алгоритмов форматирования. Мы можем добавить новые подклассы к классу `Compositor`, не трогая классов глифов, и наоборот. Фактически допустимо подменить алгоритм разбиения на строки во время выполнения, добавив одну-единственную операцию `SetCompositor` к базовому интерфейсу класса `Composition`.

---

[1] Композитор должен получить коды символов глифов `Character`, чтобы вычислить места разбиения на строки. В разделе 2.8 мы увидим, как можно получить информацию полиморфно, не добавляя специфичной для символов операции к интерфейсу класса `Glyph`.

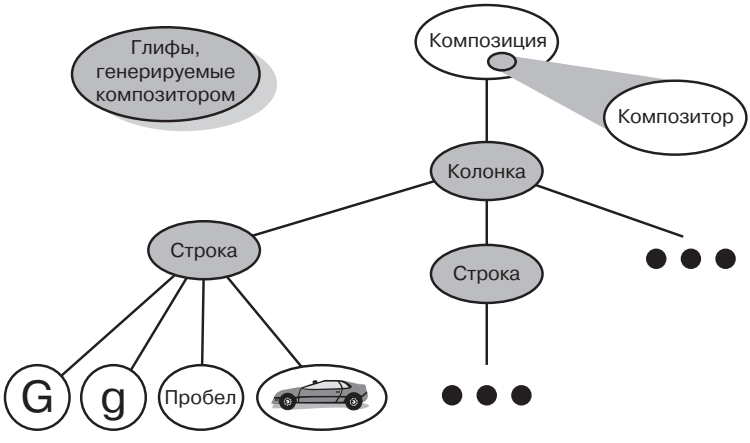

Рис. 2.6. Объектная структура, отражающая алгоритм разбиения на строки, выбираемый композитором

Стратегия

Инкапсуляция алгоритма в объект – это назначение паттерна стратегия. Основными участниками паттерна являются объекты-стратегии, инкапсулирующие различные алгоритмы, и контекст, в котором они работают. Композиторы представляют варианты стратегий; они инкапсулируют алгоритмы форматирования. Композиция – это контекст для стратегии композитора.

Ключ к применению паттерна стратегия – спроектировать интерфейсы стратегии и контекста, достаточно общие для поддержки широкого диапазона алгоритмов. Не должно возникать необходимости изменять интерфейс стратегии или контекста для поддержки нового алгоритма. В нашем примере поддержка доступа к потомкам, их вставки и удаления, предоставляемая базовыми интерфейсами класса `Glyph`, достаточно общая, чтобы подклассы класса `Compositor` могли изменять физическую структуру документа независимо от того, с помощью каких алгоритмов это делается. Аналогично интерфейс класса `Compositor` дает композициям все, что им необходимо для инициализации форматирования.

2.4. Оформление пользовательского интерфейса

Рассмотрим два усовершенствования пользовательского интерфейса Lexi. Первое добавляет рамку вокруг области редактирования текста, чтобы четко обозначить страницу текста, второе – полосы прокрутки, позволяющие пользователю просматривать разные части страницы. Чтобы упростить добавление и удаление таких элементов оформления (особенно во время выполнения), мы не должны использовать наследование. Максимальной гибкости можно достичь, если другим объектам пользовательского интерфейса даже не будет известно о том, какие еще есть элементы оформления. Это позволит добавлять и удалять декорации, не изменяя других классов.

Прозрачное обрамление

В программировании оформление пользовательского интерфейса означает расширение существующего кода. Использование для этой цели наследования не дает возможности реорганизовать интерфейс во время выполнения. Не менее серьезной проблемой является комбинаторный рост числа классов в случае широкого использования наследования.

Можно было бы добавить рамку к классу `Composition`, породив от него новый подкласс `BorderedComposition`. Точно так же можно было бы добавить и интерфейс прокрутки, породив подкласс `ScrollableComposition`. Если же мы хотим иметь и рамку, и полосу прокрутки, следовало бы создать подкласс `BorderedScrollableComposition`, и так далее. Если довести эту идею до логического завершения, то пришлось бы создавать отдельный подкласс для каждой возможной комбинации элементов оформления. Это решение быстро перестает работать, когда количество разнообразных декораций растет.

С помощью композиции объектов можно найти куда более приемлемый и гибкий механизм расширения. Но из каких объектов составлять композицию? Поскольку известно, что мы оформляем существующий глиф, то и сам элемент оформления могли бы сделать объектом (скажем, экземпляром класса `Border`). Следовательно, композиция может быть составлена из глифа и рамки. Следующий шаг – решить, что является агрегатом, а что – компонентом. Допустимо считать, что рамка содержит глиф, и это имеет смысл, так как рамка окружает глиф на экране. Можно принять и противоположное решение – поместить рамку внутрь глифа, но тогда пришлось бы модифицировать соответствующий подкласс класса `Glyph`, чтобы он «знал» о наличии рамки. Первый вариант – включение глифа в рамку – позволяет поместить весь код для отображения рамки в классе `Border`, оставив остальные классы без изменения.

Как выглядит класс `Border`? Тот факт, что у рамки есть визуальное представление, наталкивает на мысль, что она должна быть глифом, то есть подклассом класса `Glyph`. Но есть и более настоятельные причины поступить именно таким образом: клиентов не должно волновать, есть у глифов рамки или нет. Все глифы должны трактоваться единообразно. Когда клиент сообщает простому глифу без рамки о необходимости нарисовать себя, тот делает это, не добавляя никаких элементов оформления. Если же этот глиф заключен в рамку, то клиент не должен обрабатывать рамку как-то специально; он просто предписывает составному глифу выполнить отображение точно так же, как и простому глифу в предыдущем случае. Отсюда следует, что интерфейс класса `Border` должен соответствовать интерфейсу класса `Glyph`. Чтобы гарантировать это, мы и делаем `Border` подклассом `Glyph`.

Все это подводит нас к идее *прозрачного обрамления* (transparent enclosure), где комбинируются понятия о композиции с одним потомком (однокомпонентные), и о совместимых интерфейсах. В общем случае клиенту неизвестно, имеет ли он дело с компонентом или его *обрамлением* (то есть родителем), особенно если обрамление просто делегирует все операции своему единственному компоненту. Но обрамление может также и расширять поведение компонента, выполняя дополнительные действия либо до, либо после делегирования (а возможно, и до, и после). Обрамление может также добавить компоненту состояние. Как именно, будет показано ниже.

Моноглиф

Концепцию прозрачного обрамления можно применить ко всем глифам, оформляющим другие глифы. Чтобы конкретизировать эту идею, определим подкласс класса `Glyph`, называемый `MonoGlyph`. Он будет выступать в роли абстрактного класса для глифов-декораций вроде рамки (см. рис. 2.7). В классе `MonoGlyph` хранится ссылка на компонент, которому он и переадресует все запросы. При этом `MonoGlyph` по определению становится абсолютно прозрачным для клиентов. Вот как моноглиф реализует операцию `Draw`:

```
void MonoGlyph::Draw (Window* w) {
    _component->Draw(w);
}
```

Подклассы `MonoGlyph` замещают по меньшей мере одну из таких операций переадресации. Например, `Border::Draw` сначала вызывает операцию родительского класса `MonoGlyph::Draw`, чтобы компонент выполнил свою часть работы, то есть нарисовал все, кроме рамки. Затем `Border::Draw` рисует рамку, вызывая свою собственную закрытую операцию `DrawBorder`, детали которой мы опустим:

```
void Border::Draw (Window* w) {
    MonoGlyph::Draw(w);
    DrawBorder(w);
}
```

Обратите внимание, что `Border::Draw`, по сути дела, *расширяет* операцию родительского класса, чтобы нарисовать рамку. Это не то же самое, что простая *замена* операции: в таком случае `MonoGlyph::Draw` не вызывалась бы.

На рис. 2.7 показан другой подкласс класса `MonoGlyph`. `Scroller` – это `MonoGlyph`, который рисует свои компоненты на экране в зависимости от

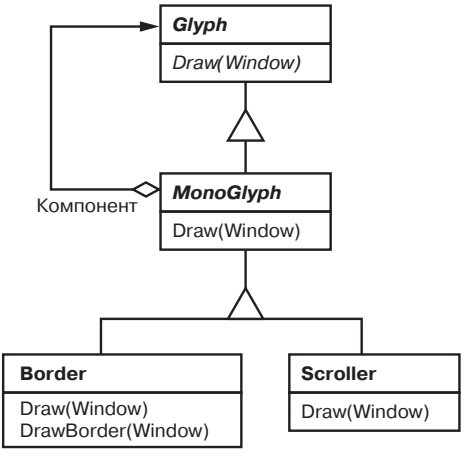

Рис. 2.7. Отношения класса MonoGlyph с другими классами

положения двух полос прокрутки, добавляющихся в качестве элементов оформления. Когда `Scroller` отображает свой компонент, графическая система обрезает его по границам окна. Отсеченные части компонента, оказавшиеся за пределами видимой части окна, не появляются на экране.

Теперь у нас есть все, что необходимо для добавления рамки и прокрутки к области редактирования текста в Lexi. Мы помещаем имеющийся экземпляр класса `Composition` в экземпляр класса `Scroller`, чтобы добавить интерфейс прокрутки, а результат композиции еще раз погружаем в экземпляр класса `Border`. Получившийся объект показан на рис. 2.8.

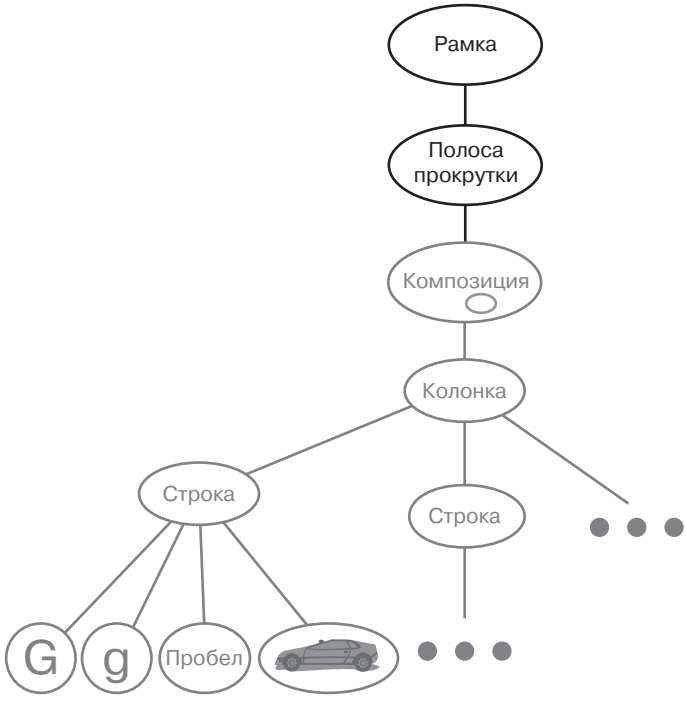

Рис. 2.8. Объектная структура после добавления элементов оформления

Обратите внимание, что мы могли изменить порядок композиции на обратный, сначала добавив рамку, а потом погрузив результат в `Scroller`. В таком случае рамка прокручивалась бы вместе с текстом. Может быть, это то, что вам нужно, а может, и нет. Важно лишь, что прозрачное обрамление легко позволяет экспериментировать с разными вариантами, освобождая клиента от знания деталей кода, добавляющего декорации.

Отметим, что рамка допускает композицию только с одним глифом, не более того. Этим она отличается от рассмотренных выше композиций, где родительскому объекту позволялось иметь сколько угодно потомков. Здесь же заключение чего-то

в рамку предполагает, что это «что-то» имеется в единственном экземпляре. Мы могли бы приписать некоторую семантику декорации более одного объекта, но тогда пришлось бы вводить множество видов композиций с оформлением: оформление строки, колонки и т.д. Это не дает ничего нового, так как у нас уже есть классы для такого рода композиций. Поэтому для композиции лучше использовать уже существующие классы, а новые добавлять для оформления результата. Отделение декорации от других видов композиции одновременно упрощает классы, реализующие разные элементы оформления, и уменьшает их количество. Кроме того, мы избавлены от необходимости дублировать уже имеющуюся функциональность.

Паттерн декоратор

Паттерн декоратор абстрагирует отношения между классами и объектами, необходимые для поддержки оформления с помощью техники прозрачного обрамления. Термин «оформление» на самом деле применяется в более широком смысле, чем мы видели выше. В паттерне декоратор под ним понимается нечто, что возлагает на объект новые обязанности. Можно, например, представить себе оформление абстрактного дерева синтаксического разбора семантическими действиями, конечного автомата – новыми состояниями или сети, состоящей из устойчивых объектов, – тэгами атрибутов. Декоратор обобщает подход, который мы использовали в Lexi, чтобы расширить его область применения.

2.5. Поддержка нескольких стандартов внешнего облика

При проектировании системы приходится сталкиваться с серьезной проблемой: как добиться переносимости между различными программно-аппаратными платформами. Перенос Lexi на другую платформу не должен требовать капитального перепроектирования, иначе не стоит за него и браться. Он должен быть максимально прост.

Одним из препятствий для переноса является разнообразие стандартов внешнего облика, призванных унифицировать работу с приложениями на данной платформе. Эти стандарты определяют, как приложения должны выглядеть и реагировать на действия пользователя. Хотя существующие стандарты не так уж сильно отличаются друг от друга, ни один пользователь не спутает один стандарт с другим – приложения, написанные для Motif, выглядят не совсем так, как аналогичные приложения на других платформах, и наоборот. Программа, работающая более чем на одной платформе, должна всюду соответствовать принятой стилистике пользовательского интерфейса.

Наша проектная цель – сделать так, чтобы Lexi поддерживал разные стандарты внешнего облика и чтобы легко можно было добавить поддержку нового стандарта, как только таковой появится (а это неизбежно произойдет). Хотелось бы также, чтобы наш дизайн решал и другую задачу: изменение внешнего облика Lexi во время выполнения.