ФЕДЕРАЛЬНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Факультет математики, механики и компьютерных наук

Ю. А. Кирютенко

Объектно-ориентированное программирование
Архитектура MVC

(Учебное пособие)

Ростов-на-Дону
2008

Рецензенты: доцент кафедры информатики и вычислительного эксперимента ЮФУ, канд. ф.-м. наук Савельев В.А.

доцент кафедры фундаментальной и прикладной математики РГЭУ, канд. ф.-м. наук Богачев Т.В.

Ю. А. Кирютенко.
Объектно-ориентированное программирование. Архитектура MVC: учебное пособие. — Ростов-на-Дону, 2008. — 153 с.

Аннотация

Учебное пособие по архитектуре MVC, используемой в объектно-ориентированномой среде программирования VisualWorks для построения графического интерфейса пользователя. В книге рассматриваются особенности среды разработки приложений VisualWorks 7.4.1 и механизм зависимости, встроенный в архитектуру MVC. Основу книги составили материалы лекций, читавшихся автором на факультете математики, механики и компьютерных наук Южного федерального университета.

Для студентов и преподавателей вузов, программистов и всех желающих изучить язык Smalltalk, объектно-ориентированное программирование, их реализации и инструменты в среде VisualWorks.

Введение

Smalltalk (Смолток) — современный, простой и мощный, продуманный и универсальный язык программирования, в котором объектная идеология реализована с максимально возможной последовательностью и полнотой. Смолтоковские системы используются сегодня и как системы для обучения программированию, и как инструменты для разработки сложных приложений с развитым пользовательским интерфейсом, которые позволяют вести коллективную работу, быстро реализовывать и эффективно поддерживать сложные приложения с высокими требованиями к надежности и постоянной готовности к функционированию, эффективно сопровождать созданный программный продукт во время эксплуатации, производить регулярные модификации. Среди типовых задач, для которых сегодня используется Смолток, можно выделить следующие:

- автоматизация, робототехника;
- диспетчеризация, планирование;
- интерфейс пользователя;
- коммуникации, связь;
- медицина, экспертные системы;
- обработка коммерческой информации;
- системы управления;
- тренажеры, моделирование;
- обучение программированию.

Хотя многие зарубежные компании стремятся не указывать какие-либо данные об используемых ими системах разработки, опубликованные данные все же показывают, что одними из основных языков программирования являются C++ и Смолток,.

Во многих странах мира, начиная с 1983 года, Смолток активно используется, а монографии и учебники по языку издаются постоянно, в России он мало известен. За прошедшие годы на русском языке вышло всего несколько книг и статей, среди них [10, 11, 15, 12]. Данное учебное пособие по содержанию является продолжение ранее вышедшей книги [12], в которой рассматривались вопросы программирования на языке Смолток и учебника «Объектно-ориентированное программирование. Среда VisualWorks », созданного в 2007 году в рамках гранта ЮФУ, в котором рассматривались особенностям объектно-ориентированного про-

граммирования на языке Смолток в многопользовательской среде VisualWorks — наиболее часто используемой смолтоковской среде.

Общие положения

Учебное пособие предназначено для учебной дисциплины «Языки программирования» федерального компонента ЕН раздела «Компьютерные науки» для бакалаврской образовательной программы по направлению 010100 — «Математика». По содержанию оно представляет собой продолжение ранее вышедшей в издательстве «Вузовская книга» учебного пособия [12], в котором рассматриваются вопросы программирования на языке Смолток, и учебника “Объектно-ориентированное программирование: Среда VisualWorks” ([13]), выполненного в 2007 году в рамках гранта ЮФУ, в котором рассматриваются особенностям объектно-ориентированного программирования на языке Смолток в многопользовательской среде VisualWorks. Предполагается, что студент знаком с зыком Смолток и средой VisualWorks, либо по этим книгам, либо по другим источникам, например, по книгам [1]–[5] или по поставляемой с системой VisualWorks документации [6]–[9].

Основная цель учебного пособия — продолжить обучение студентов объектно-ориентированной технологии программирования и объяснить методику построения графического интерфейса пользователя в объектно-ориентированной среде программирования, показать на простых примерах, как это делать. Рассматриваются возможности по построению графического интерфейса пользователя (GUI), заключенные в среде VisualWorks. Основой служит архитектура MVC — Model-Viewer-Controller.

Структура учебного пособия. Учебное пособие состоит из двух модулей. Первый модуль включает шесть глав и посвящен описанию архитектуры MVC, основных инструментов и используемых объектов (виджеты, диалоговые окна, меню). Каждая глава сопровождается контрольными вопросами. Второй модуль также состоит из шести глав и посвящен примерам построения приложений с интерфейсом пользователя. В конце этой части приведены проектные задания для самостоятельной работы разной степени сложности. В учебном пособии создана система гиперссылок, предметный указатель и оглавление, также содержащие гиперссылки, что позволяет использовать учебное пособие как электронное.

В пособии описывается технология построения приложения с графическим интерфейсом пользователя (GUI) в среде VisualWorks, котрая основывается на структуре, содержащей три части: (1) графический интерфейс, (2) прикладную модель, (3) модель предметной области.

Графический интерфейс (GUI) — то, что пользователь видит и с чем взаимодействует.

Модель предметной области — набор классов, моделирующих в приложении объекты реального или воображаемого мира.

Прикладная модель — клей между GUI и моделью предметной области, которая интерпретирует события GUI, отслеживает изменения, вводимые пользователем, и сообщает о них объектам модели предметной области. Последние информируют о своём изменении все зависимые от них объекты (в том числе и компоненты GUI), используя механизм зависимости, для обновления интерфейса.

Поскольку все прикладные модели схожи в поведении, в VisualWorks определён класс ApplicationModel, который описывает общие функциональные возможности приложений, и классы прикладных моделей должны быть его подклассами. Существенной частью ApplicationModel являются реализуемые им методы обработки прерываний (hook), которые выполняются всегда, когда приложение открывается или закрывается. Переопределяя эти методы в классе прикладной модели, можно управлять запуском и завершением создаваемого приложения.

В VisualWorks работа компонентов GUI (виджетов) основана на отделении изображения (или представления — view) от взаимодействия пользователя с программой (через контроллеры — controller) и от объектов, ответственных за работу с данными (моделей — model). Это разделение и называется архитектурой «Model-View-Controller» — (MVC).

Модуль I

Механизмы и инструменты построения GUI

Цель модуля — описать основные структуры и механизмы, заложенные в архитектуру MVC, а также используемые при построении графического интерфейса пользователя (GUI) инструменты (UIPainter, MenuEditor, ResourseFinder, Image Editor, Hot Region Editor).

Глава 1

Механизм зависимости

Между объектами в объектно-ориентированной среде программирования существуют разные отношений. Например, наследование — отношение класса к классу, создание экземпляра — отношение класса к экземпляру, отношение включения (содержит или содержится) — отношение экземпляра к экземпляру.

Однако, могут существовать и другие важные отношения, которые возникают тогда, когда состояние или поведение одного объекта зависит от состояния другого объекта. Такие отношения называются отношениями зависимости и могут использоваться для того, чтобы связать два объекта так, что изменение состояния одного объекта автоматически приведет к изменению состояния или поведения другого.

Например, цена товара и её изменение может интересовать многих, хотя самой цене нет дела до их состояния (после её повышения!). Или емкость с жидкостью — объекты могут интересоваться температурой или количеством жидкости в емкости, но самой емкости это не интересно. Однако, и товар, и емкость должны позволять другим объектам как-то узнавать об изменениях в их состоянии.

Могут существовать разные методы решения последней задачи. Например, заинтересованные объекты могут запрашивать товар и емкость об их состоянии через определенные промежутки времени. Но такой способ имеет существенные недостатки — требуются компьютерные ресурсы, позволяющие постоянно запрашивать информацию, и время на её передачу заинтересованным объектам. Но главный недостаток состоит в том, что нужные объекты не будут проинформированы об изменении в режиме реального времени.

Другой подход реализован в Смолтоке в виде механизма зависимости. Механизм зависимости предоставляет возможность автоматически сообщать объектам о том, что некоторый объект, скажем, **A**, изменил своё состояние. Это достигается разрешением объектам регистрировать свой интерес в получении сообщений об изменениях объекта **A**, то есть в

регистрации себя как иждивенца (зависимого от) объекта **A**. Объект, от которого зависит иждивенец — объект **A**, называется родителем (предком) отношения зависимости. Когда родитель изменяет свое состояние, он сообщает сам себе об этом, тем самым запуская механизм уведомления всех иждивенцев о происшедшем изменении. Иждивенец, когда получает сообщение об изменении родителя, может сделать то, что считает нужным сделать в этой ситуации. Поэтому родителю не нужно программно следить за всеми объектами, которым требуется сообщать о своём изменении.

Механизм зависимости позволяет вовлечь в отношение зависимость любые объекты, в том числе и те о существовании которых не было известно во время разработки класса родителя.

1.1 Основной механизм зависимости

В Смолтоке реализовано несколько составляющих механизма зависимости, но все они полагаются на основной механизм родителя, посылающего при своем изменении самому себе сообщение об изменении из семейства changed, и иждивенца, которому посылается сообщение о модификации из семейства update.

Механизм зависимости реализуется классом Object, а значит любой объект может зарегистрировать себя в качестве иждивенца любого другого объекта. Методы семейства changed реализуются в Object так, что они ничего не делают.

```
Object >> changed
    self changed: nil

Object >> changed: anAspectSymbol
    self changed: anAspectSymbol with: nil

Object >> changed: anAspectSymbol with: aParameter
    self myDependents
            update: anAspectSymbol
            with: aParameter
            from: self
```

Обратим внимание на то, что в методе changed:with: класса Object используется сообщение myDependents, позволяющее получить набор иждивенцев для приемника этого сообщения. Его реализация в классе Object очень проста

```
Object >> myDependents
    "Возвратить иждивенцев приемника сообщения или
    nil."
    ^DependentsFields at: self ifAbsent: [nil]
```

Класс Object определяет переменную класса DependentsFields, которая является словарем с ключами-объектами, и значениями — набором его иждивенцев. Общий механизм доступа — сообщения вида

```
DependentsFields at: objectWithDependents ifAbsent: [nil]
DependentsFields at: objectWithDependents put: collectionOfDependents.
```

Сделать один объект зависимым от другого объекта позволяет сообщение addDependent:. Следующее выражение сделает объект anObject иждивенцем (зависимым от) объекта theObject:

```
theObject addDependent: anObject.
```

Разорвать отношение зависимости между объектами, то есть сделать один объект более независимым от другого позволяет сообщение removeDependent:. Следующее выражение делает объект anObject независимым от объекта theObject.

```
theObject removeDependent: anObject.
```

Из приведенного кода следует важный вывод: объект может определить набор своих иждивенцев, но он не содержит, а поэтому не знает, списка объектов, от которых он сам зависит. Этот вывод определяет, в каком направлении работает механизм зависимости.

Как видно из реализации changed-сообщений, если объект посылает себе сообщение changed, оно преобразуется в сообщение changed:with: с двумя параметрами равными nil. При посылке сообщения changed: или changed:with: передается более подробная информация об изменении.

Если надо сообщить иждивенцам, что именно изменилось, следует посылать сообщение changed: и передавать в качестве параметра связанный с изменением символ. Когда в результате иждивенец получит сообщение модификации, он посмотрит на символ и решит, интересно ли ему это изменение. В качестве аргумента changed: можно посылать что угодно, однако, обычно, посылается символ, являющийся селектором, используемым для получения нового значения, что при стандартном именовании методов доступа указывает на имя изменившейся переменной экземпляра родителя. Традиционно этот параметр называют aspect — аспект, часть, характеристика. Таким образом, он позволяет определить, какой аспект родителя изменился.

Если нужно уточнить происшедшее изменение, то следует посылать сообщение changed:with:, передавая в качестве параметров не только символ, но ещё старое или новое значение аспекта родителя, то есть сказать, **как** указанный аспект изменился. Есть два соглашений о том, что передавать в качестве аргумента with:. Можно передавать старое значение аспекта, так как иждивенцу больше не откуда его узнать, но он всегда может получить новое значение, запрашивая о нём родителя. Однако, можно передавать и новое значение аспекта, чтобы сократить количество пересылаемых сообщений. При этом предполагается, что если иждивенец нуждается в старом значении аспекта, он должен его хранить сам. Любой подход хорош, если он не меняется по ходу дела.

Итак, в конечном счете, родитель посылает всем иждивенцам сообщение модификации update:with:from:. Поэтому иждивенцы должны реализовывать одно из сообщений семейства update. Но что делать, если родитель не позаботился обо всех параметрах сообщения update:with:from:? Класс Object, аналогично тому, как он расширяет сообщение changed до changed:with:, сжимает сообщение update:with:from: до update, и все эти сообщения тоже ничего не делают.

```
Object >> update: anAspectSymbol with: aParameter from: aSender
    ^self update: anAspectSymbol with: aParameter

Object >> update: anAspectSymbol with: aParameter
    ^self update: anAspectSymbol

Object >> update: anAspectSymbol
    ^self
```

Первое сообщение самое мощное. Если в классе иждивенца реализовано оно, именно его будет получать каждый иждивенец всякий раз, когда родитель получит change-сообщение. Значения anAspectSymbol и aParameter будут теми, которые использовались в change-сообщении, или nil, если использовалось одно из более простых change-сообщений. Значение aSender — объект, которому было послано change-сообщение, оно включено сюда для того, чтобы можно было это узнать.

Если метод update:with:from: не был реализован в классе иждивенца, он наследует его реализацию по умолчанию из класса Object, которая просто посылает сообщение update:with:. Если это сообщение реализовано в классе иждивенца, то оно и будет выполняться. Здесь доступны те же параметры, за исключением aSender, так что будет не известно, какой объект инициировал посылку иждивенцу метода модификации.

Если метод update:with: не был реализован в классе иждивенца, ре-

ализация по умолчанию, наследуемая из класса Object, вызовет метод update:. Если он реализован в классе иждивенца, то у него есть доступ только к anAspectSymbol. Если и этот метод не будет реализован, выполнится метод update: реализации по умолчанию, и результат тоже будет благоприятным, поскольку ничего не произойдет.

1.2 Механизм зависимости класса Model

Одна из проблем механизма зависимости, реализованного в классе Object состоит в том, что если объект не удаляет себя, как иждивенца, посылая родителю сообщение removeDependent: self, набор иждивенцев никогда не исчезает из переменной класса DependentsFields. Так как зависимые объекты продолжают существовать в этом словаре как иждивенцы, они не будут собираться как мусор даже при том, что больше никому не нужны. Хотелось бы иметь механизм, который автоматически избавлялся от отношений зависимости и иждивенцев, когда их родитель уничтожается как мусор.

Решить эту задачи позволяет класс Model — подкласс класса Object. Он определяет единственную переменную экземпляра с именем dependents, в которой экземпляр класса Model хранить набор своих иждивенцев. Это означает, что если такой экземпляр уничтожается как мусор, набор его иждивенцев также исчезнет даже при том, что иждивенцы никогда не посылают сообщение removeDependent:. Однако, как хорошая практика программирования, всегда следует посылать сообщение removeDependent:, удаляя ненужные зависимости.

1.3 Сообщение expressInterestIn:for:sendBack:

Основной механизм зависимости имеет тот существенный недостаток, что **всем** иждивенцам посылаются **все** сообщения изменения, даже если иждивенец в них не заинтересован. Иждивенцы должны реализовать одно из сообщений семейства модификации и должны выяснить, заинтересованы ли они в том конкретном символе, который был им передан. Такой подход требует в коде метода модификации условных выражений.

Другой подход состоит в том, чтобы сообщить родителю отношения зависимости, что данный иждивенец заинтересован в конкретном его изменении и просит посылать сообщение модификации только в этом случае. То есть, иждивенец просит родителя, присылать ему только конкретный символ. Такое отношение зависимости возникает при посылке родителю вместо сообщения addDependent: сообщения

```
expressInterestIn: anAspectSymbol for: anObject sendBack: aSelector.
```

Этим сообщением иждивенец просит родителя послать объекту anObject (иждивенцу) сообщение, определяемое именем aSelector только в том случае, когда в родителе происходит изменение аспекта anAspectSymbol. Если сообщение, которое иждивенец попросит послать, не имеет параметров, объекту anObject будет послано только это сообщение. Если сообщение aSelector имеет один параметр (бинарное сообщение или сообщение из одного ключевого слова), объекту anObject будет послано в качестве параметра старое значение аспекта. Если такое сообщение состоит из двух ключевых слов, объекту anObject будет послано старое значение аспекта родителя и объект, аспект которого изменился (родитель).

Когда построенное отношение зависимости более не нужно, иждивенец должен сообщить родителю о нежелании получать информацию о его конкретном изменении. Это делается посредством посылки родителю сообщения retractInterestIn: anAspectSymbol for: anObject , в котором, как и раньше, anAspectSymbol — более не интересный аспект родителя, а anObject — иждивенец, отказывающийся от информации.

1.4 Сообщение onChangeSend:to:

Такое расширение отношений зависимости выглядит довольно сложно, и некоторые классы, экземпляры которых могут выступать как родители отношений зависимости, предлагают более простые сообщения. Например, абстрактный класс ValueModel, подкласс класса Model, определяет сообщение onChangeSend: aSelector to: anObject, которое требует не трех, а двух параметров: сообщения, которое должно посылаться, и объекта (иждивенца), которому его надо послать. Здесь не надо определять, в каком изменении заинтересован иждивенец, так как в ValueModel может измениться аспект #value, так что сообщение onChangeSend:to: автоматически устанавливает заинтересованность в символе #value:

```
ValueModel >> onChangeSend: aSymbol to: anObject
    self
        expressInterestIn: #value
        for: anObject
        sendBack: aSymbol
```

Экземпляр ValueModel — это обертка вокруг некоторого значения. Но изменять это значение непосредственно нельзя, нужно для этого обратиться к содержащему его экземпляру ValueModel. Поскольку экземпляр

ValueModel полностью контролирует изменения своего значения, он может уведомлять иждивенцев о его изменениях. Чтобы установить новое значение, надо послать ValueModel сообщение value: с новым значением в качестве параметра. Сообщение value: установит новое значение и уведомит о его изменении всех иждивенцев, выполняя выражение self changed: #value.

```
ValueModel >> value: newValue
    self setValue: newValue.
    self changed: #value
```

Таким образом, как только экземпляр ValueModel получит сообщение value:, он пошлёт назад всем своим иждивенцам, сообщение, которое было зарегистрировано сообщением onChangeSend:to:. Экземпляры подклассов класса ValueModel широко используются при построении интерфейса пользователя.

1.5 Особенности механизма зависимости

Есть два правила, которые обязательно надо помнить при использовании механизма зависимости.

Во-первых, родители отношений зависимости не должны посылать явных сообщений своим иждивенцам, поскольку механизм зависимости является односторонним и родители не должны ничего знать о своих иждивенцах. Они не должны даже знать, есть ли у них иждивенцы. Таким образом, код вида

```
self myDependents do:
    [ :each | (each isKindOf: SomeClass)
                ifTrue: [. . . ]]
```

нарушает первое правило механизма зависимости, что говорит о плохом проектировании приложения и некорректном использовании механизма зависимости.

Во-вторых, что более важное, следует всегда уничтожать зависимости, когда они больше не нужны. Программа не должна допускать утечки памяти. Утечка памяти происходит потому, что объекты вовремя не собираются как мусор, а объект не собирается как мусор, если на него есть хоть одна ссылка. Если родительский объект хранится в переменной класса или глобальной переменной, или на него ссылаются другие объекты, его иждивенцы никогда не будут собраны как мусор. Наиболее

трудно отследить утечку памяти, когда родитель — экземпляр подкласса Model, поскольку зависимости «не моделей» отслеживаются в одном месте, в переменной класса DependentsFields класса Object.

Чтобы избежать возможной утечки памяти, следует уничтожать зависимости, когда иждивенец больше не нужен родителю, или иждивенец больше не нуждается в родителе. Если зависимость была основана, используя сообщение addDependent:, её нужно удалить сообщением removeDependent:. Если она была основана сообщениями expressInterestIn:for:sendBack: или onChangeSend:to:, её нужно удалить сообщением retractInterestIn:for:. Разрушение зависимостей обычно проводится в методах release.

1.6 Контрольные вопросы

1. Какие классы реализуют механизм зависимости? В чем различие таких реализаций?
2. Какое сообщение делает один объект зависимым от другого?
3. Какое сообщение разрывает отношение зависимости между объектами?
4. Сколько существует changed-сообщений?
5. Сколько существует update-сообщений?
6. Как взаимодействуют changed-сообщения и update-сообщения?
7. Какое сообщение класса Model позволяет иждивенцу зарегистрировать свою заинтересованности в конкретном изменении?
8. Какое упрощение вносит сообщение onChangeSend:to: класса ValueModel в процесс организации зависимости между объектами?

Глава 2

Архитектура MVC

Механизм зависимости активно используется в задаче построения интерфейса пользователя (сокращенно, UI — User Interface). Чтобы его использовать разработаны специальные классы. Они реализованы в рамках архитектуры MVC — Model-View-Controller. Мощные инструменты построения интерфейса пользователя в среде VisualWorks скрывают детали архитектуры MVC, однако, чтобы эффективно их использовать, полезно иметь по крайней мере представление об этой архитектуре и о том, что и как создает VisualWorks. Полезны эти знания еще и для того, чтобы строить интерфейсы пользователя вне тех рамок, которые предлагают инструменты VisualWorks.

Кратко рассмотрим архитектуру MVC, то, как она работает и как ее использовать.

2.1 Основные понятия

Архитектура MVC определяет специфический способ построения приложений, включающих графические интерфейсы пользователя. Основная идея, заключенная в MVC, состоит в том, что интерфейс пользователя приложения должен отделяться от функциональных возможностей самого приложения. Отделение приложения от его интерфейса пользователя позволяет разрабатывать их отдельно. Но более важно то, что такое разделение позволяет легко соединять новый интерфейс пользователя с уже существующим приложением. Кроме того, компоненты существующего интерфейса пользователя могут использоваться с новыми приложениями. При таком разделении приложение можно использоваться и без его интерфейса пользователя, например, в другом приложении. Как легко заметить, все эти объяснения связаны с модульностью, возможностью многократного использования объекта и инкапсуляцией, которые поддерживает объектно-ориентированная методология программирования.

Наиболее важные объекты со стороны функциональных возможностей

приложения часто называют **моделями**. В MVC они скрываются под буквой M. Наиболее важные объекты со стороны интерфейса пользователя часто называют окнами просмотра и контроллерами. В MVC они скрываются под буквами V и C, соответственно. Смолтоковская библиотека классов содержит три соответствующих класса с именами Model (Модель), View (Окно просмотра) и Controller (Контроллер). Большинство объектов, которые ведут себя как модели, окна просмотра или контроллеры, как правило, наследуют из одного из указанных базовых классов. Хотя, это не всегда так, особенно для моделей. Наиболее важно в MVC не то, наследуют ли обязательно классы приложения из классов Model, View или Controller, а ведут ли себя экземпляры этих классов подобно модели, окну просмотра, или контроллеру. Посмотрим, как каждый из этих объектов проявляет себя, как они взаимодействуют между собой и с другими объектами.

Модели

Модели реализуют функциональные возможности приложений. Они ответственны за хранение данных, соответствующих приложению, и определяемые приложением операции на этих данных. Они могут быть очень простыми, например, модель может быть экземпляром класса String), или очень сложными, возможно целое приложение.

Важное значение имеет то, что модели содержат данные, и определенные типы операций на них, которые **не зависят** от интерфейса пользователя, что позволяет другим интерфейсам пользователя, или другим объектам использовать функциональные возможности модели.

Окна просмотра

Окна просмотра представляют пользователю информацию, хранящуюся в модели. Они ответственны за принятие данных, содержащихся в объектах модели и их отображении на экране в форме текста, графики, виджетов и так далее. Однако, окна просмотра не «понимают» (и не должны понимать) смысла данных.

Библиотека смолтоковских классов определяет несколько типов окон просмотра. Они позволяют отображать данные моделей различными способами, не изменяя при этом самой модели. Существуют окна просмотра почти для всего. Одно окно обычно содержит много других визуальных объектов (их называют виджетами), сотрудничающих между собой, чтобы создать единый интерфейс пользователя.

Контроллеры

Контроллеры ответственны за обработку ввода от пользователя. Они «слушают» клавиатуру и мышь, и интерпретируют ввод в терминах того, как должна управляться модель. Библиотека классов содержит много различных классов контроллеров. Каждый из контроллеров обычно имеет «узкую специализацию», позволяющую работать с одним или несколько типами окон просмотра. Везде, где есть окно просмотра, обычно есть и невидимый пользователю контроллер. Фактически, когда создается окно просмотра, без каких-либо дополнительных действий создается контроллер соответствующего класса и автоматически присоединяется к окну просмотра. Почти каждый класс окон просмотра знает, какой класс контроллера с ним связан.

2.2 Взаимодействие объектов MVC

Опишем основной механизм того, как в приложении совместно работают модели, окна просмотра и контроллеры, реализуя функциональные возможности приложения, представляя их пользователю и позволяя пользователю взаимодействовать с приложением.

Окно просмотра имеет переменную экземпляра с именем model, которая содержит модель, отображаемую окном просмотра, и переменную с именем controller, которая содержит контроллер, используемый для изменения модели, когда пользователь пользуется мышью или клавиатурой. Окно просмотра имеет и другие переменные, которые указывают на его контейнер (на то окно просмотра, частью которого оно является), и на его компоненты (окна, которые оно содержит).

Точно так же контроллер имеет переменные экземпляра model и view, которые указывают на связанные с ним модель и окно просмотра, соответственно.

Таким образом, окно просмотра и контроллер могут получить информацию друг о друге, и о модели, обращаясь к соответствующей переменной. Это означает, что окно просмотра способно спросить модель о данных, которые предполагается отобразить на экране. Аналогично, контроллер способен посылать модели сообщения, информирующие модель о том, какие надо выполнить операции, в соответствии с командами пользователя, переданными через мышь или клавиатуру. Например, когда контроллер получает сообщение о щелчке мыши, он может сам узнать координаты точки щелчка, но должен спросить у окна просмотра, какой объект там отображается, чтобы решить, какое действие выбрать.

Но почему не объединить окно просмотра и контроллер в один объект?

Во-первых, потому, что наличие двух отдельных объектов делает возможным их объединение различными способами. Можно использовать с одним и тем же контроллером другое окно просмотра, чтобы отобразить данные модели по-другому или можно использовать то же самое окно просмотра, но изменить контроллер, чтобы по-другому работать с мышью и клавиатурой.

Во-вторых, отделение окна просмотра от контроллера позволяет им наследовать из разных классов. Это означает, что в иерархии классов функциональные возможности окна просмотра и функциональные возможности контроллера могут быть структурированы по-разному. Такая методика создания функциональных возможностей из комбинаций экземпляров — фактически единственный способ преодолеть отсутствующее в Смолтоке множественное наследования, когда класс может наследовать не только из одного прямого суперкласса, а сразу из нескольких прямых суперклассов.

Модель не имеет переменной экземпляра, содержащей окно просмотра, или контроллер. Вместо этого, окна просмотра делают себя иждивенцами модели. Это означает, что модель использует встроенные change-сообщения, являющиеся частью механизма зависимости, для того, чтобы окно просмотра могло узнает о происшедших в модели изменениях и отразить результаты изменений на экране. Это означает, в частности, что к одной и той же модели, чтобы отображать ее и взаимодействовать с ней, можно одновременно подключать различные комбинации «окно просмотра / контроллер». Всякий раз, когда модель изменяется и посылает себе change-сообщения, и все они будут получать «сообщения о модификации» и будут знать, надо ли им в результате происшедших изменений перерисовывать самих себя.

Предельно упрощая ситуацию, рассмотрим пример, в котором все классы и методы вымышлены, но который иллюстрирует те основные механизмы, на которых строится взаимодействие модели, окна просмотра и контроллера.

Итак, рассмотрим приложение, состоящее из переключателя, который может быть или включен (on) или выключен (off). Модель такого приложения, назовем ее buttonModel, имеет одну булеву переменную, которая может принимать одно из двух значений, true или false. Пусть эта переменная имеет имя value. Окно просмотра, назовем его buttonView, ответственно за создание графического представления состояния переменной value из buttonModel, имеет на экране форму двумерного виджета кнопки. И, наконец, с окном связан контроллер, buttonController, ответственный за изменение состояния переменной value из buttonModel всякий раз, когда на виджете кнопки окна просмотра происходит щелчок пользова-

телем левой кнопкой мыши. Рассмотрим типичную последовательность событий, которая могла бы произойти в «жизни» этих объектов.

1. Окно, в которое встроена кнопка buttonView, открывается и buttonView должен в первый раз нарисовать виджет. Чтобы определить правильный вид кнопки (нажата она или нет), окно просмотра посылает модели buttonModel сообщение value. Возвращаемое им значение будет или true или false, и окно просмотра, используя эту информацию, нарисует виджет.

2. Приходит пользователь и нажимает левую кнопку мыши, располагая курсор в области экрана, управляемой buttonView. ButtonController видит это, и вступает с buttonView в диалог, чтобы определить был ли выполнен щелчок внутри виджета кнопки. Если это так, контроллер посылает сообщение модели, говоря ей, чтобы она изменила свое состояние (на true, если было false, и наоборот). Такое сообщение могло бы иметь вид

   ```
   model value: model value not.
   ```

   Оно спрашивает модель о ее значении (используя model value), инвертирует это значение (not), и посылает результат обратно модели (value:).

3. Модель изменилась, и как часть своего метода value: она посылает сама себе сообщение changed: #value. Это вызывает посылку сообщений о модификации всем иждивенцам модели, сообщая им, что объект, от которого они зависят, изменился.

4. Объект buttonView, являющийся одним из иждивенцев модели, получает модифицирующее сообщение в виде update: #value. Его реализация update: просто заново выполняет код, используемый для прорисовки виджета на том же месте. Он посылает модели сообщение value:, чтобы выяснить новое значение ее переменной, и затем рисует виджет соответствующим образом.

В этом примере отметим ряд важных моментов.

Первый: ни окно просмотра, ни контроллер не содержат состояние модели (true или false). Каждый раз, когда они нуждаются в нем, они спрашивают об этом у модели.

Второй: контроллер ничего не знает о визуальном размещении виджета. Когда требуется такая информация, он запрашивает ее у окна просмотра.

Третий: когда контроллер изменяет состояние модели, он ничего не сообщает непосредственно окну просмотра.

Четвертый: модель ничего не знает об окне просмотра. Когда ее состояние изменено, в данном случае контроллером, окно просмотра узнает об изменениях только благодаря механизму зависимости.

Пятый: когда модель сообщает окну просмотра, что она изменилась (использование зависимости), она не сообщает окну просмотра о своем новом состоянии, а только сообщает ему, какой *ее аспект* изменился (#value. Окно просмотра должно запросить у модели новое состояние этой переменной.

В заключение, обратим внимание на то, что изменить состояние buttonModel мог бы любой другой объект, а не buttonController, посылая модели сообщение value:. Это приведет в действие механизм зависимости, позволяя окну просмотра правильно отобразить новое состояние модели.

2.3 Расширение MVC в VisualWorks

Среда VisualWorks вносит дополнение в рассмотренную выше классическню архитектуру MVC. Основное изменение сводиться к разбиению модельной части MVC на две части. Первая часть обычно называется *моделью данных (или моделью предметной области)* и действует только как хранилище для данных прикладной программы, а вторая — *прикладной моделью (или моделью приложения)* и отражает специфическое воздействие на эти данные в данном приложении. Очевидно, что интерфейс пользователя и модель данных должны работать вместе и для организации такой совместной работы нужно поддерживать большой объем кода, не имеющего никакого отношения к реальным функциям приложения. Этот код и образует прикладную модель.

Такое разбиение модели удаляет специфическую для данного приложения обработку данных из модели данных, делая последнюю чаще повторно используемой, позволяет поместить некоторые функциональные возможности интерфейса пользователя в модель приложения и различным типам моделей наследовать из различных источников.

Когда создается в VisualWorks класс прикладной модели, он обычно становиться подкалассом в классе ApplicationModel, обеспечивающим фундамент для создания прикладных моделей.

Чтобы сформировать полное приложение, прикладная модель управляет логикой того, например, как окна просмотра, которые ничего не знают о конкретных моделях данных, и которые, в свою очередь, ничего не знают об интерфейсе пользователя, сотрудничают между собой. Прикладная модель — клей, который создает работающее приложение. Приложение VisualWorks определяется как подкласс в ApplicationModel, именно для

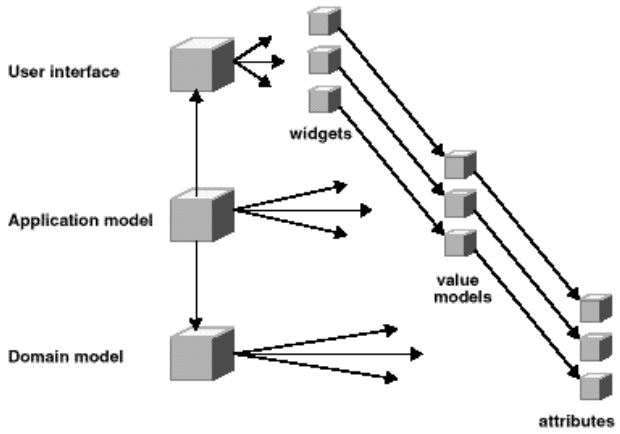


Рис. 2.1: Компоненты приложения

того, чтобы действовать в качестве такого посредника. Этот подкласс порождается (вручную или автоматически) из холста, представляющего интерфейс пользователя, когда интерфейс пользователя инсталлируется в систему. Окна, меню, некоторые графические объекты, и другие ресурсы определяются именно внутри прикладной модели.

Связи прикладной модели координируют работу объектов предметной области и объектов интерфейса пользователя, определяя взаимоотношения между ними. Каждый виджет интерфейса пользователя связывается с атрибутом (по другой терминологии — аспектом, доступной пользователю интерфейса структурной частью модели) или с операцией объекта из предметной области.

Действие пользователя на виджете, типа нажатия кнопки, ввода данных в поле ввода, выбора элемента их списка, либо изменяет атрибут объекта предметной области, либо запускает операцию, определенную в модели предметной области. Например, для виджета, устанавливающего атрибут (например, поля ввода), прикладная модель преобразует значение, полученное виджетом и посылает модели предметной области соответствующее сообщение, устанавливающее новое значение модели. Точно так же, если воздействующее на интерфейс пользователя значение изменяется в модели предметной области, прикладная модель принимает это изменение и посылает его UI.

Используемый прикладной моделью механизм связи интерфейса с моделью данных называются *адаптером*. Адаптер находится между конкретными объектами интерфейса и объектами предметной области, согласовывая (адаптируя) сообщения и значения. Адаптеры еще называют *моделями значения — value models*, поскольку они определяет соотношение между значениями атрибутов и виджетами, которые зависят

от этих значений. Таким образом, виджет интерфейса, модель значения и модель данных из предметной области составляют тестно связанную триаду, в которой виджет — иждивенец модели значения.

Есть разные виды моделей значений для различных видов значений атрибутов, классы которых являются подклассами класса ValueModel. Например, ValueHolder, который используется тогда, когда значение атрибута — простое значение данных, типа строки, которая хранится в модели приложения. AspectAdaptor используется тогда, когда значение данных внедрено в составной атрибут, который хранится в модели предметной области. PluggableAdaptor используется тогда, когда сообщения value и value: нужно перевести в действия, определяемые соответствующим блоком.

2.4 Контрольные вопросы

1. За что в приложении ответственны модели?
2. За что в приложении ответственны окна просмотра?
3. За что в приложении ответственны контроллеры?
4. Какие классы реализуют модели, окна просмотра и контроллеры?
5. Как связаны в приложении, использующем архитектуру MVC, окно просмотра и контроллер?
6. Как связаны в приложении, использующем архитектуру MVC, модель и окно просмотра?
7. Как связаны в приложении, использующем архитектуру MVC, модель и контроллер?
8. Какое изменение в архитектуру MVC вносит среда VisualWorks?
9. Что такое модель предметной области?
10. Что такое модель приложения? Какой класс в среде VisualWorks обеспечивает основу для создания модели приложения?
11. Какой объект находится между конкретными объектами интерфейса приложения и объектами предметной области? Какие зависимости устанавливаются в этой триаде?

Глава 3

Среда GUI системы VisualWorks

Построение графического интерфейса пользователя (GUI) в VisualWorks можно выполнить визуально, выбирая и размещая виджеты на холсте (canvas) (раскрашивая холст). Инструмент, который используется при этом, называется UIPainter (Живописец). Созданный в нём холст определяет окно приложения. Как результат сохранения созданного окна создается подкласс класса ApplicationModel — модель приложения, автоматически генерируемая инструментом UIPainter, включающая методы-заглушки для определенных разработчиком аспектов модели и действий, связанных с элементами графического интерфейса пользователя. В процессе разработки класса модели приложения программистом определяются методы-заглушки, а модель приложения связывается с моделью данных так, чтобы ввод в интерфейс передавался модели данных, а изменения в модели данных отображались в интерфейсе.

Кратко опишем основные инструменты, используемые при построении интерфейсов приложений.

3.1 Инструмент UIPainter

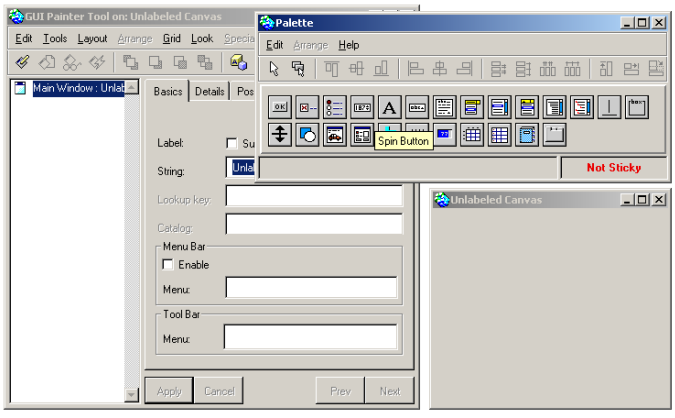


Рис. 3.1: Начальный вид инструмента UIPainter

Чтобы построить GUI приложения, следует открыть инструмент UIPa-

inter, нажимая на кнопку с холстом на мольберте или последовательно выбирая пункты меню Painter → New Canvas[1]. Откроются три окна (см. рисунок 3.1):

Palette **(Палитра)** — окно с виджетами, в котором представлены виджеты, помещаемые в интерфейс, и кнопки управления виджетами;

GUI Painter Tool on: — окно инструментов построения холста, которое используется для выполнения на нем основных операций, для иерархического представления виджетов на текущем холсте, для определения свойств выбранного виджета;

Canvas — холст, окно с заголовком Unlabeled Canvas, место построения GUI.

Холст изначально чист. Размеры холста можно изменять, тем самым определяя его размер в момент открытия приложения. Виджеты, размещаемые на холсте, имеют множество свойств, которые определяют их вид и взаимодействие с моделью данных, свойства самого виджета.

Как только холст раскрашен, его следует установить (инсталлировать) в приложение, нажимая первую кнопку в строке инструментов или выбирая пункты меню Edit → Install. Это приводит к записи спецификации определения холста в определяемый программистом класс модели приложения. Спецификация используется приложением для того, чтобы открыть окно. Позже можно отредактировать спецификацию, повторно открывая холст в окне Живописца. Всплывающее меню операций холста содержит команды для работы с холстом, которые доступны и из других частей Живописца, в частности, из его окна инструментов.

Палитра имеет по одной кнопке для каждого типа виджетов, которые можно располагать на холсте, включая простые элементы (например, метки и кнопки), и более сложные элементы, подобные иерархическим спискам и таблицам. Нужные части из палитры просто перемещаются на холст. При нажатии на одну из кнопок для виджетов, внизу палитры появится описание выбранного виджета. Чтобы, например, перенести на холст поле ввода (Input Field), надо выбрать его, переместите курсор на холст в нужное место и установить виджет, щелкая левой кнопкой мыши. Чтобы добавить несколько копий того же самого виджета следует воспользоваться кнопкой Sticky Select (второй слева в верхней панели инструментов). Тогда виджет устанавливается на холст, каждый раз, когда нажимается левая кнопка мыши. Чтобы возвратиться к установке одного виджета, следует выбрать кнопку Select One (первую слева в верхней панели инструментов).

[1]Пакет с инструментом должен быть загружен, см. [13, раздел 5.2]

Окно палитры имеет несколько кнопок для выполнения основных операций с виджетами, типа выравнивания и распределения виджетов. Эти операции также доступны через меню палитры и всплывающее меню операций самого холста.

Окно инструментов предназначено для управления виджетами при конфигурировании холста. Можно иметь несколько открытых холстов одновременно (несколько раз нажимая на кнопку с холстом на мольберте), но только одно окно инструментов и одну палитру, и они будут взаимодействовать с выбранным в данное время холстом.

Окно инструментов имеет меню и кнопки для конфигурирования холста и виджетов на нём. Слева расположена панель иерархического представления в виде древа компонентов холста. Сам холст располагается наверху списка, как Main Window (ОсновноеОкно). Ниже него располагаются, выровненные по уровню вложенности, остальные виджеты холста. Каждый виджет имеет имя по умолчанию, которое можно изменить, делая его более описательным и полезным. Щелчком на виджете холста или на имени виджета в иерархическом представлении можно выбрать виджет и сконфигурировать его.

Список виджетов в иерархическом представлении также отражает порядок их табуляции (если табуляция возможна). Порядок по умолчанию — от вершины к основанию. Перемещение виджета в списке на новое место изменяет его место в последовательности табуляции. Самый большой раздел окна инструментов занимают страницы наборов свойств выбранного на холсте виджета. Определенные страницы и их содержание изменяются от виджета к виджету. Но каждый виджет имеет страницы свойств Basics, Details, Color, Position. Дополнительно страницы добавляются по мере выбора виджета.

Некоторые общие свойства стандартных виджетов, изначально присутствующих в среде VisualWorks описаны в главе 4. Дополнительные свойства каждого виджета можно найти в [7, глава 9]. В среду VisualWorks можно подгружать наборы других виджетов.

3.2 Resource Finder

Холсты, меню, графические образы рассматриваются в VisualWorks как ресурсы и хранятся как спецификации в методах соответствующего подкласса класса Application Model. Чтобы найти созданный ресурс, следует в основном окне системы выбрать команду меню Painter → Resource Finder или в панели инструменов нажать на кнопку с холстом и биноклем и открыть окно инструмента Resource Finder (ПоисковикРесурсов

или БраузерРесурсов). В окне инструмента Resource Finder классы приложений перечисляются в левой панели. Можно изменять список перечисляемых классов, используя меню View окна Resource Finder.

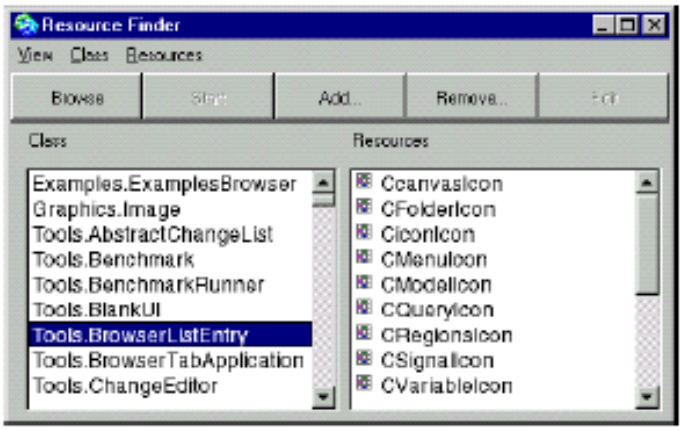


Рис. 3.2: Окно инструмента Resource Finder

Класс модели приложения может поддерживать несколько интерфейсов, каждый из которых может состоять из множества холстов (для главного окна, для вторичных окон, для диалоговых окон). Правая панель перечисляет все ресурсные методы выбранного класса. Двойной щелчок на ресурсе или нажатие кнопки Edit, открывает на выбранном ресурсе окно для его редактирования.

3.3 Menu Editor

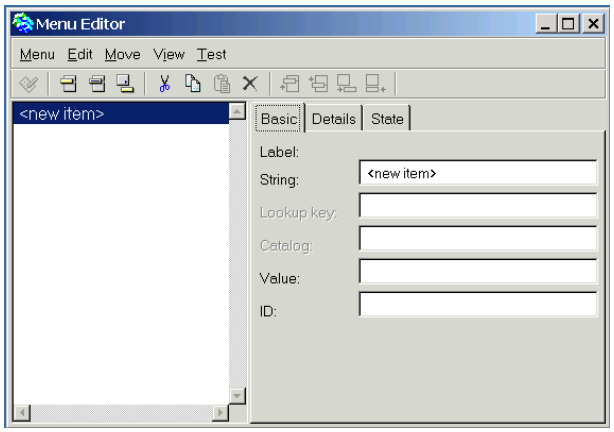


Рис. 3.3: Окно инструмента Menu Editor

Инструмент Menu Editor (РедакторМеню) обеспечивает визуальный, интуитивно понятный, способ построения меню. Чтобы открыть окно инструмента Menu Editor, следует выбрать в главном окне VisualWorks командуPainter → Menu Editor.

Меню может быть добавлено в любой графический интерфейс как стандартное раскрывающееся меню или как всплывающее (при нажатии правой кнопки мыши) меню панели окна. Предусмотрены команды меню и кнопки для добавления элементов, подменю, и для реконструкции элементов. К подменю обращаются из родительского меню.

Уровни меню обозначаются выравниванием элементов. Чтобы изменить уровень элемента, надо его выбрать и щелкнуть на кнопке со стрелкой влево или вправо. Изменить порядок элементов позволяют кнопки со стрелкой вверх или вниз.

Есть ещё три страницы свойств для каждого пункта меню. Страница Basic определяет отображаемую метку в виде строки, ключ поиска и каталог, если он используется, возвращаемое значение и идентификатор, для программного доступ к элементу. Страница Details определяет клавиши (ключи) доступа к пункту меню, иконку, если она нужна, и текст справки. Страница State позволяет устанавливать значение по умолчанию (для доступного пользователю пункта меню) и определять его доступность при открытии интерфейса.

Приложение во время выполнения часто должно изменять меню или его элементы, или, чтобы указать на состояние, активизировать и блокировать доступ к элементам меню. Детали того, как создавать меню и управлять им во время выполнения, описываются в главе 6 *“Меню”*.

3.4 Image Editor

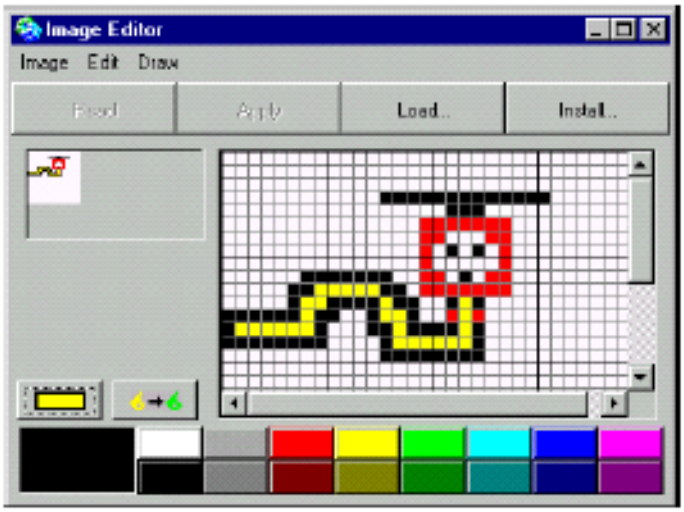


Рис. 3.4: Окно инструмента Image Editor

Графический интерфейс приложения, указывая на функции виджетов, обычно использует иконки и простую графику. Чтобы нарисовать их пиксел за пикселом небольшой рисунок (иконку, форму курсора), а затем

сохранить его в ресурсном методе, используется инструмент Image Editor (РедакторИзображения).

Чтобы открыть окно инструмента Image Editor, следует выбрать в главном окне VisualWorks команду Painter → Image Editor. Средства управления раскрашиванием достаточно стандартны и аналогичны командам простых программ для подготовки иллюстраций.

Чтобы сделать созданную графику доступной приложению, нужно нажать кнопку Install, определить класс приложения как класс, в который установится графика, и в этом классе имя метода класса (в протоколе resource) для графики. Как результат, графика установится как ресурс, к которому можно обращаться (в том числе и из браузера ресурсов).

3.5 Hot Region Editor

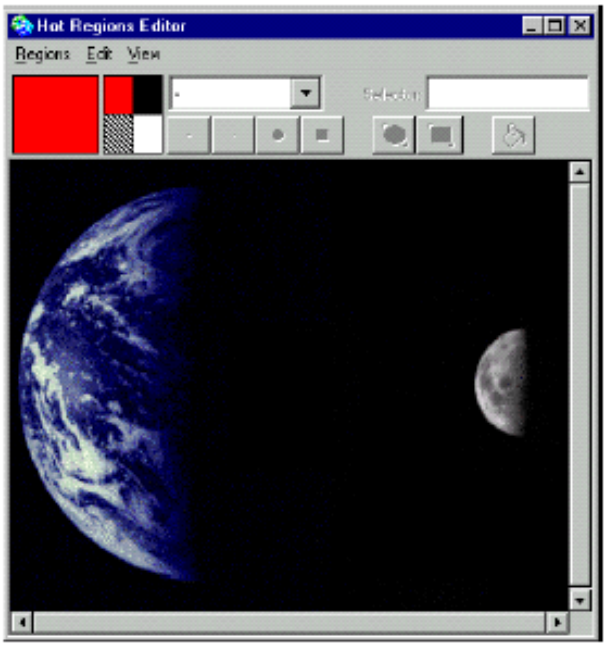


Рис. 3.5: Окно редактора Hot Region Editor

Изображение, на котором можно щелкнуть кнопкой мыши и выполнить некоторое действие, не только полезно с точки зрения построения функциональности приложения, но может придать графическому интерфейсу современный вид. Добавить такую возможность на холст позволяет виджет ClickWidget и инструмент Hot Region Editor (РедакторОбластиРеагирования), который позволяет определить области изображения, реагирующие на щелчок кнопкой мыши. Чтобы определять области реагирования, следует загрузить графический ресурс в редактор, а затем, используя инструменты для рисования, выделить и закрасить области, реагирующие на щелчок, определить селектор сообщения, который будет

посылаться при щелчке на нем кнопкой мыши.

3.6 Контрольные вопросы

1. Какой инструмент позволяет построить графический интерфейс пользователя приложения? Из каких окон он состоит?
2. Что такое ресурсы приложения? Какой инструмент позволяет работать с ресурсами?
3. Какой инструмент обеспечивает визуальный, интуитивно понятный, способ построения меню окна приложения?
4. Какой инструмент позволяет построить небольшой рисунок?
5. Какой инструмент позволяет щелкнуть кнопкой мыши на части рисунка, чтобы выпол- нить некоторое действие?

Глава 4

Построение приложения в UIPainter

Рассмотрим процесс построения графического интерфейса пользователя (GUI) и его связывания с моделью (моделями) предметной области. Считается, что перед построением GUI уже должна существовать модель предметной области, но это совсем не обязательно, можно начинать работу над приложением с GUI. В том же случае, когда модель предметной области проста, часто создание приложения — это создание только модели приложения.

Инструмент UIPainter создает спецификацию UI и модель приложения. Рассмотрим основные операции по построению GUI при использовании инструмента UIPainter, опишем процесс построения окна приложения и, наконец, рассмотрим несколько подходов, используемых при построении адаптеров, которые связывают GUI и предметную область. В последующих главах, опираясь на изложенный материал, приведем примеры построения различных интерфейсов.

4.1 Создание интерфейса пользователя

Чтобы создать визуальную часть графического интерфейса пользователя, надо определить содержание и размещение каждого окна (виджета) создаваемого приложения. Среди них главное окно приложения, и, возможно, вторичные и диалоговые окна. Как часть этой процедуры автоматически генерируются методы-заглушки и хранители значений для связи интерфейса с моделью предметной области. Позже, программируется характерное для данного приложения поведение каждого из определённых в окнах виджетов.

Создание окна происходит на холсте, на котором размещаются виджеты, а окно инструментов холста дает быстрый доступ к множеству общих команд их размещения. Если приложение использует несколько окон, каждое создается на отдельном холсте. Одно окно будет основным, остальные вторичными.

После раскрашивания холста, определяются свойства самого окна и каждого из его виджетов. Свойства определяют для всех виджетов разнообразные визуальных атрибуты (шрифт, цвет, границы, местоположение в окне, и так далее), для некоторых виджетов, например, полей ввода, свойства указывают на природу данных, которые будут отображаться, и на то, как на эти данные будет ссылаться приложение. Первоначально свойства устанавливаются во время создания GUI, используя страницы свойств окна инструментов Живописца.

Большинство свойств, доступных для установки, очевидны, но некоторые свойства нуждаются в начальном пояснении:

String — строка метки виджета.

Action — имя метода в модели приложения, которое посылается приложению, когда по виджету действия (например, кнопке) щелкают левой кнопкой мыши. Метод действия по умолчанию (который ничего не делает) создается тогда, когда выбирается команда Define для этого виджета или для всего окна, его содержащего.

Aspect — название свойства в модели приложения, которое представляет модель значения для виджета. Переменная экземпляра и метод доступа к ней в модели приложения по умолчанию создается для аспекта тогда, когда выбирается команда Define. Модель значения — обычно хранитель значения или адаптер аспекта.

ID — имя виджета в модели приложения. Инструмент определяет имя по умолчанию для каждого виджета, но можно изменить его на нечто более описательное. По имени можно легко идентифицировать виджет в списке виджетов. Когда нужно обратиться к виджету программно, можно в объекте, отвечающем за построение интерфейса (в экземпляре UIBuilder) использовать именно это имя.

Lookup Key — ключ поиска в каталоге сообщений. Если на странице Tools окна Settings Tool среды (открывается командой System → Settings) активизирована опция Show UI for Globalization, это поле отображается в метках тех виджетов, которые имеют метку. Дополнительная информация о каталогах сообщений и их использовании содержится в учебнике [8].

Fly-by Help — страница Fly-by Help позволяет определить строку-подсказку относительно предназначения виджета, которая отображается, когда пользователь размещает курсор поверх виджета и задерживает его.

Многие из свойств виджетов будут описаны далее в других разделах, по мере усложнения интерфейса.

Установка холста

В любое время создания холста, его можно сохранить в модели приложения (инсталлировать). Установка холста создает спецификацию интерфейса (interface specification), которая используется для того, чтобы отобразить UI. Спецификация сохраняется как метод класса в модели приложения. Чтобы установить холст, следует или нажать кнопку Install или выбрать команду меню Edit → Install в окне инструментов холста или выбрать команду Install в всплывающем меню операций холста. Диалоговое окно установки позволяет ввести необходимую информацию.

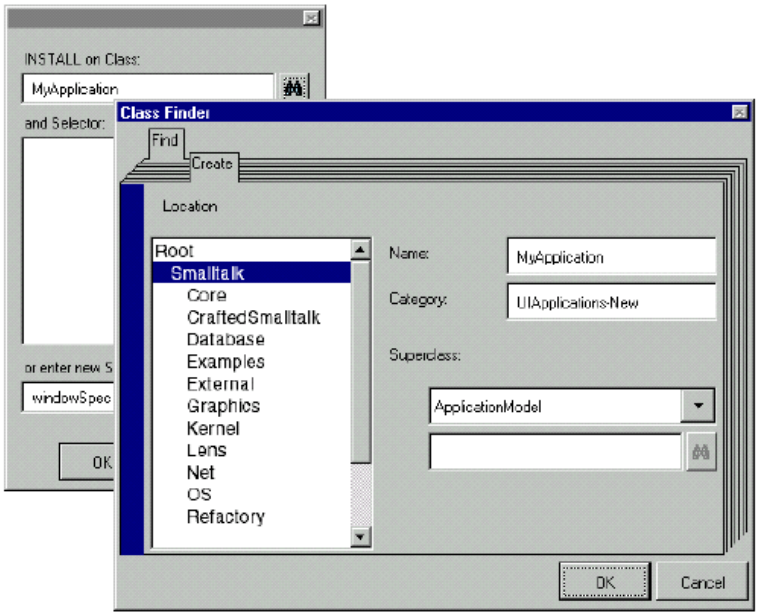


Рис. 4.1: Окна первоначальной установки холста

Для нового холста, нужно ввести имя класса в поле Install on Class. Это может быть имя нового класса или имя уже существующего класса. В случае нового класса, в последующем диалоговом окне появится запрос о соответствующем суперклассе.

Кроме того нужно будет ввести или выбрать селектор метода (имя) спецификации. По умолчанию используется имя windowSpec, которое является стандартным именем для спецификации основного окна. Если определяется вторичное окно, нужно будет выбрать другое имя. Остаётся нажать кнопку OK, чтобы сохранить спецификацию.

Если был определён новый класс, система попросит выбрать для него суперкласс [12, раздел 1.1.1] и пространство имён [13, глава 3], [6, chapter 6]. Для приложения обычно в качестве суперкласса выбирается

ApplicationModel и либо создаётся своё пространство имён, либо используется пространство имен Smalltalk.

Суперкласс ApplicationModel обеспечивает поддержку тех приложений, которые содержат графический интерфейс. Суперкласс SimpleDialog обеспечивает поддержку для диалоговых окон. Если же класс должен быть подклассом некоторого другого класса, следует выбрать Other, и ввести его имя. Поле Category позволяет установить категорию (пакет) [13, глава 4] для нового класса. Можно использовать существующую категорию или создать новую, которая лучше идентифицирует создаваемое приложение.

Повторное открытие холста

Когда сохраняется холст, его спецификация добавляется к ресурсам приложения. После того, как холст закрыт, его снова можно открыть для редактирования, используя браузер ресурсов, в котором следует выбрать имя класса, содержащее нужный холст, а затем выбрать спецификацию холста и нажать кнопку Edit (или выбрать команду edit из всплывающего меню операций панели ресурсов). Чтобы сохранить изменения, сделанные на холсте, надо повторно установить его в класс.

Окно приложения можно открыть и через системный браузер [13, глава 6], если выбрать вкладку Class над панелью категорий методов, выбрать в панели категорий методов категорию interface specs, выбрать метод windowSpec, выбрать вкладку Visual над текстовой панелью и, наконец, нажать кнопку Edit или Open.

Определение модели значения виджета

На холсте можно определить переменные экземпляра и методы доступа по умолчанию для аспектов и методы действия, которые были указаны как свойства виджетов. Для большинства виджетов, имена аспектных переменных и методов доступа к ним определяются именем, введенным в поле свойства Aspect. Имя метода действия определяется по имени, введенном в поле свойства Action. Некоторые виджеты вообще не используют переменные или методы, и таким образом не имеют соответствующих свойств.

Как только введены имена аспектных переменных и методов действия для одного или нескольких виджетов, надо выбрать эти виджеты (или по одному или как группу), и либо нажать кнопку Define или выбрать команду меню Edit → Define в окне инструментов холста, либо выбрать

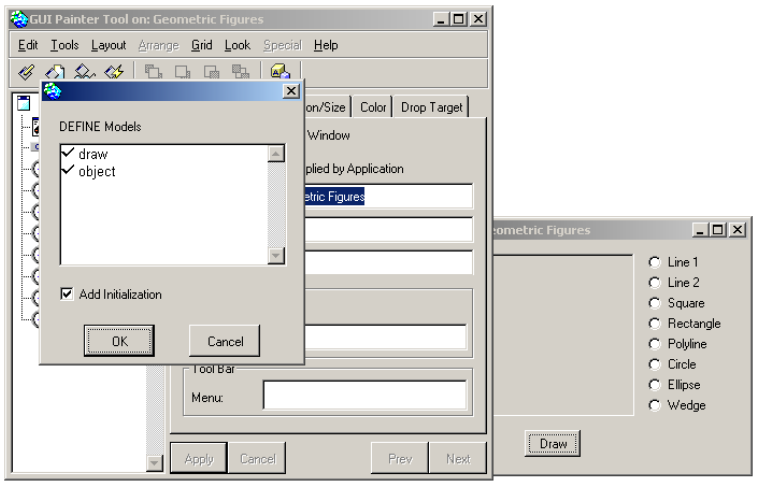


Рис. 4.2: Окно установки моделей значений виджетов

пункт Define в всплывающем меню операций холста. Отобразиться диалоговое окно, перечисляющее определенные имена аспектов и действий, требуя подтверждения на операцию. Если снять отметку с имени аспекта, для него модель значения создаваться не будет. Если снять отметку с имени метода действия, для него создаваться не будет ничего не делающий метод-заглушка. Отмена выбора элемента в последующем чрезвычайно важна в последовательности определения операций, поскольку переопределение будет обычно приводить к потере изменений, сделанных ранее программистом.

Переключатель Add Initialization определяет объем кода, который будет сгенерирован для аспектных методов доступа. Если переключатель не выбран, для аспекта просто возвращается переменная экземпляра. Если переключатель выбран, создается метод доступа, содержащий код ленивой инициализации переменной, (этот код проверяет, имеет ли переменная значение, и если нет, определяет ее начальное значение). Тип назначаемого значения зависит от виджета.

Как только методы по умолчанию сгенерированы, можно их редактировать, обеспечивая обработку потребностей приложения. Например, чтобы соединить UI с моделью предметной области, возможно нужно будет переопределить некоторые аспектные методы, позволяя использовать адаптеры аспекта (см. главу ch:4).

Тестирование интерфейса пользователя

Когда холст окна установлен в модели приложения, уже есть минимальное приложение, которое можно запустить. В этот момент, виджеты окна демонстрируют своё родовое поведение, поскольку пока ещё не до-

бавлено нужное поведение и отображаемые данные. Однако, уже можно посмотреть, как интерфейс будет смотреться при выполнении приложения. Чтобы запустить приложение, открывая холст, надо нажать кнопку Open или выбрать команду меню Edit → Open в окне инструмента холста, или выбрать в всплывающем меню операций холста команду Open. Теперь можно перейти в новое окно и поработать с ним: передвинуть, свернуть, развернуть, закрыть. Но пока ничего характерного именно для этого приложения в окне сделать нельзя.

4.2 Форматирование холста

Рассмотрим операции форматирования и размещения виджетов, которые позволяют скорректировать тот вид, в котором окно будет появляться на экране в приложении. Большинство операций выполняются на холсте при построении (сборке) GUI. Некоторые операции форматирования можно устанавливать или изменять программно во время выполнения приложения. Есть команды, которые делают это наряду с командами холста. Для общих команд доступа к виджетам из выполняющегося приложения используется ID виджета.

Установка размеров окна

При редактировании холста, можно менять размеры окна простым перемещением углов холста. Вообще же, можно выбирать размеры и соотношения, определяя как размещается в окне каждый виджеты. Когда создается холст, по мере добавления виджетов, вероятно изменяются и размеры холста.

Как только виджеты размещены, следует гарантировать, что окно будет открываться в тех размерах, которые были установлены при разработке. Чтобы сделать это, в окне инструментов холста надо выбрать из иерархического представления окна элемент Main Window, и перейти на страницу Position/Size.

Внизу страницы располагаются три набора для ширины и высоты. Размерности Specified определяют размер открывающегося окна. Чтобы установить те размерности, которые были установлены на холсте, надо просто нажать на кнопку Specified, и текущие размерности будут установлены, если применить изменения (кнопка Apply), и установить заново холст (кнопка Install). Можно ввести и другие размеры, если нужно.

Часто бывает полезно определить для окна некоторые размеры, и потребовать, чтобы пользователь не смог сделать размеры реального ок-

на больше или меньше заданных. Эти размеры обеспечиваются полями Minimum и Maximum. Здесь можно ввести точные размерности, но проще реально изменить размеры холста до наименьшего и нажать кнопку Minimum и до наибольшего и нажать кнопку Maximum. Если все три набора размеров будут одинаковы, то размеры окна во время выполнения приложения изменить будет нельзя.

Если окну позволяется изменять размеры, надо будет решить, как будут себя при этом вести размеры виджетов, установленных в окне (см. раздел *"Установка размеров виджетов"*).

Установка места открытия окна

В окне инструментов холста можно определить, где окно будет размещаться на экране, когда открывается приложение. Следует перейти на страницу Position/Size и выбрать опцию открытия окна приложения в верхней группе радио-кнопок:

System Default: — открывает окно так, как это определено по умолчанию в окне инструмента Settings: либо размещает окно по требованию пользователя (радио-кнопка By User), либо автоматически (радио-кнопка Automatic).

User Placement: — открывает окно так, как разместил его пользователь при создании.

Advanced: — активизирует установку следующих опций:

- System Default: — открывает окно так, как это определено по умолчанию в инструменте Settings Tool.
- Screen Center: — открывает окно в центре экрана.
- Mouse Center: — открывает окно, центрированное по положению курсора мыши.
- Last/Saved Position: — если флажок Auto выбран, окно откроется там, где оно было в последний раз закрыто. Если флажок Auto не выбран, окно будет открываться каждый раз в одном и том же месте.
- Cascade: — каждый раз окно будет открываться ниже и правее от того места, где оно открывалось последний раз. Это полезно тогда, когда могут последовательно открываться несколько копий окна.
- Specified Position: — окно открывается с верхним левым углом, расположенным в указанных координатах экрана.

После выбора нужных опций, следует применить изменения (кнопка Apply), и установить заново холст (кнопка Install).

Добавление в окно полос прокрутки

Полосы прокрутки позволяют содержать в окне виджеты, которые, в зависимости от размеров окна, не всегда будут отображаться в нем полностью. Если это нужно, можно добавить в окно горизонтальную и/или вертикальную полосу прокрутки. Для этого надо удостовериться, что в окне нет выбранных виджетов, или выбрать в иерархическом представлении окна элемент Main Window. На странице свойств Details, выбрать желаемые полосы прокрутки. Затем применить изменения (Apply), и установить заново холст (Install).

Добавление панели (строки) меню

Чтобы добавить в окно панель меню, надо сначала создать это меню, используя редактор меню (см. главу 6) и разрешить размещение меню на холсте.

1. Удостовериться, что на холсте нет выбранных виджетов, или выбрать в иерархическом представлении окна элемент Main Window.
2. На странице свойств Basics в группе свойств Menu Bar выбрать флажок Enable.
3. В поле Menu, ввести имя метода создания меню, определенного для меню в редакторе меню.
4. Установить заново холст (Install).

Редактор меню можно использовать для создания меню или до или после определения панели меню на холсте. Каждый элемент первого уровня в меню появится в панели меню окна, но только в том случае, если он имеет подменю.

Установка цвета в графический интерфейс

При установке цвета для окна и для его виджетов используется то же самое диалоговое окно свойств. Окно и его виджеты имеют четыре цветовых зоны: передний план (Foreground), фон (Background), зону выбора переднего плана (Selection foreground), зону выбора фона (Selection background).

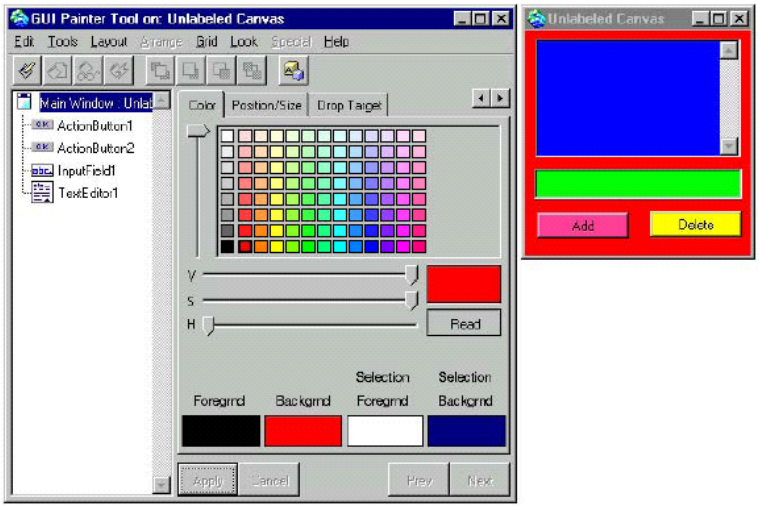


Рис. 4.3: Окно установки цвета окна и его виджетов

На странице свойств Color можно выбрать цвет для любой из этих зон. По умолчанию, цвет каждой зоны устанавливается как none, это означает, что окно или виджет наследует свои цвета. Окно наследует цвета администратора окон системы, а виджет наследует цвета от содержащего его виджета или окна.

Когда выбирается цвет, надо быть уверенным в хорошем конечном результате (что у людей без специального художественного образования бывает достаточно редко). Также, надо знать, что некоторые цвета плохо сочетаются друг с другом, а некоторые цвета не различаются некоторыми людьми из-за особенностей их глаз. Так могут не различаться красный и зеленый, и потому следует избегать красного переднего плана на зеленом фоне.

Чтобы изменить цвет окна или виджета, его надо сначала выбрать. На странице свойств Color выбрать желаемый цвет из диаграммы цветов. Щелкнуть на нужной цветовой зоне (чтобы возвратиться к none, щелкнуть на цветовой зоне еще раз, не меняя выбранного цвета. И, наконец, применить изменения (Apply), и установить заново холст (Install).

Установка размеров виджета

На холсте можно устанавливать размеры виджета, просто перемещая мышью маркеры размеров виджета (черные квадратики в угловых точках его границы). Можно сделать все выбранные виджеты выровненными, равными по высоте, ширине, равномерно распределенными, используя команды Arrange → ... из меню окна инструментов холста или команды меню Arrange окна палитры. Есть также в окне палитры меню соответствующих инструментов (под панелью меню) для выполнения таких

операций.

Чтобы произвести операцию с несколькими виджетами первый надо выбрать обычным образом — щелкая на нем левой кнопкой мыши, а остальные выбирать при нажатой клавише Shift. За образец, когда он нужен, инструмент будет использовать первый выбранный виджет.

Виджеты, когда окно открывается, появляются с теми размерами, с которыми они были созданы. Когда размер самого окна фиксирован, ничего более делать не нужно. Однако, когда размеры окна переменны, следует принять меры к тому, чтобы виджеты корректировали свои размеры относительно такого окна. Можно использовать команду Layout → Relative окна инструментов холста после чего заново инсталлировать холст, и тогда при изменение размеров окна изменение размеров виджета и в вертикальном, и в горизонтальном направлениях будут проводиться автоматически.

Создание виджета фиксированного размера

Фиксированный размер обычно используется для кнопок и меток.

1. На холсте выбрать виджет, размер которого должен быть фиксированным.
2. На странице Position выбрать кнопку планирования Origin + Width and Height, нижнюю из трех кнопок планирования, расположенных в левом нижнем углу страницы.
3. На странице свойств Position, установить значения свойств X Proportion и Y Proportion равными 0. Эти размеры управляют тем, перемещается ли виджет относительно размеров окна. Установка этих свойств равными 0 означает, что начало виджета (верхний левый угол) всегда остаться на месте.
4. На странице Position установить точку сдвига (Offset) виджета по осям OX и OY (левая верхняя точка окна имеет координаты $(0, 0)$).
5. Установить значение свойства Heigth равным высоте виджета, и значение свойства Widht равным ширине виджета в пикселах.
6. Применить свойства и инсталлировать холст.

Для более сложных ситуациях или для более точного контроля нужно устанавливать специальные свойства виджета на странице Position его окна свойств, подстраивая изменения виджета под изменения размеров окна. Эти проблемы здесь не рассматриваются.

Группировка виджетов

Окна часто содержат группы виджетов с примерно одинаковой функциональностью. С такими группами полезно имеют дело, в целях форматирования, как с единым объектом. Для этого виджеты можно сформировать в группу. Однажды сформированная группа может затем перемещаться по холсту и выравниваться.

Чтобы создать группу из нескольких виджетов, следует выбрать те виджеты, которые будут входить в группу, выбирая первый виджет обычным образом, а последующие при удерживаемой клавише Shift, а затем из меню операций или из меню окна инструментов холста выбрать команду Arrange → Group.

В иерархическом списке виджетов в окне инструментов холста появится новый псевдо виджет (с именем Composite1 или аналогичным). Составляющие его виджеты будут перечисляться под ним со сдвигом вправо. Список можно сокращать или расширять, скрывая или открывая составляющие группу виджеты.

Чтобы разрушить группу виджетов, надо ее выбрать и выбрать команду Arrange → Ungroup.

Индивидуальный виджет группы можно редактировать, не разрушая группы. Для этого в иерархическом представлении виджетов надо только выбрать желаемый виджет группы, а затем редактировать его свойства, как обычно.

Изменение шрифта виджета

Когда шрифт виджета по умолчанию не подходит, можно использовать меню Font на странице Details окна свойств виджета и выбрать альтернативный шрифт. Встроенными шрифтами являются шрифты:

System — шрифт, который соответствует шрифту, используемому в текущей операционной системе, когда он доступен. Шрифт меняется в зависимости от платформы и политики просмотра (определенной в классах Look Policy).

Default — шрифт по умолчания для данного виджета. Шрифт меняется в зависимости от политики просмотра и виджета.

Pixel — шрифты Small, Medium, Large, Fixed из TextAttributes.

Standard — шрифты Small, Medium, Large, Fixed, из VariableSizeTextAttributes.

Раскрывающиеся списки Font, Pixel и Standard также перечисляют все определённые именованные шрифты.

Именованные шрифты

VisualWorks автоматически не добавляет все шрифты используемой операционной системе в списки шрифтов. Однако, можно выборочно добавить шрифты и определить для них имена, чтобы иметь возможность их использовать по именам. Однажды определенный, получивший название шрифт добавляется в список шрифтов инструмента UI Painter и других утилит.

Чтобы открыть окно инструмента Named Fonts из окна инструментов холста, следует щелкнуть на кнопке Define Named Fonts в панели инструментов (последняя кнопка) или выбрать команду меню Edit → Define Named Fonts.

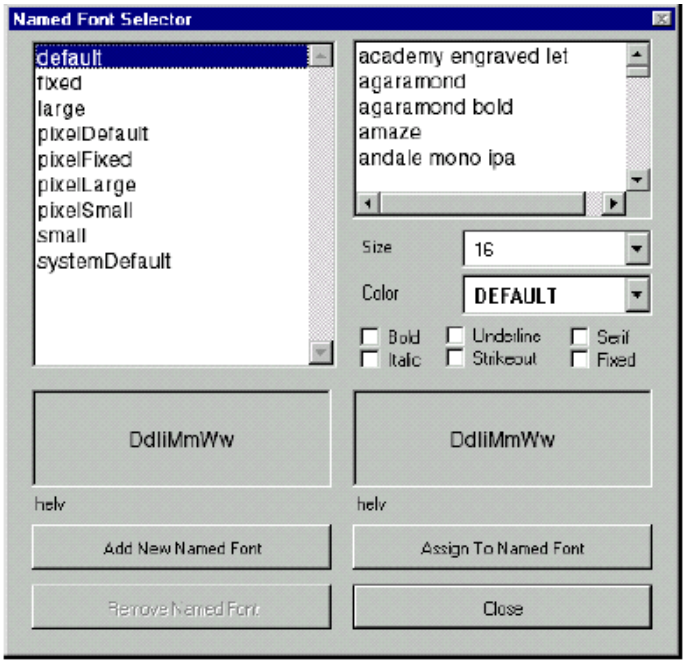


Рис. 4.4: Окно инструмента Named Fonts

Чтобы добавить именованный шрифт, нужно:

1. Щелкнуть на кнопке Add New Named Font и ввести подробное имя в окне подсказчика.
2. С выбранным новым именем, определить его атрибуты, выбирая шрифт (как известный системе) в раскрывающемся списке, размер, цвет, и другие характеристики.
3. Щелкнуть на кнопке Assign To Named Font и закрыть окно инструмента (щелкнуть на кнопке Close).

Именованный шрифт теперь можно назначить для использования в текстовом виджете или поле, выбирая его на странице Details.

Определения именованных шрифтов остаются в образе при его сохранении. Можно также сохранить определение в файле, используя команду из окна инструментов холста File → Named Fonts → File Out. . . .

Более важно то, что можно устанавливать именованные шрифты в приложение из окна инструментов холста командой File → Named Fonts → Install In Application. . . , которая добавляет метод экземпляра definedNamedFonts в модель приложения. Обратите внимание, что эта возможность доступна только тогда, когда текущее приложение уже было установлено (инсталлировано).

Изменение порядка табуляции

Когда приложение выполняется, пользователи могут использовать клавишу Tab, а не мышь, чтобы перемещаться от одного виджета окна к следующему.

Более определенно, клавиша Tab перемещает центр (фокус) на каждый виджет из цепочки табуляции. Можно добавлять виджеты в цепочку табуляции, включая свойство Can Tab на странице Details. Пассивные виджеты, типа меток и разделителей, не имеют свойства Can Tab, так что их нельзя включить в цепочку табуляции.

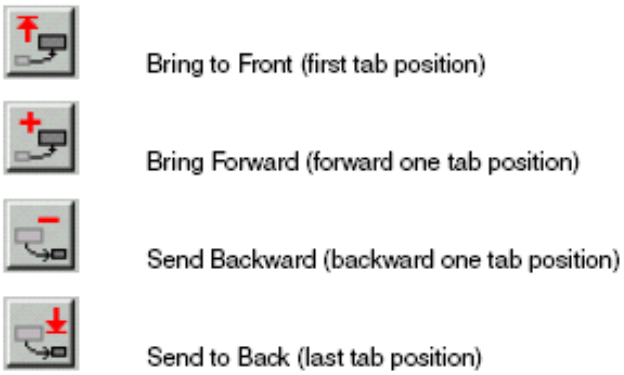


Рис. 4.5: Кнопки управления порядком табуляции виджетов

Обратите внимание, что нужно отключить свойство Can Tab в виджете текстового редактора или установить опцию Tab Require Control Key, если редактор должен интерпретировать клавишу Tab как литерал, который будет вводиться в текст. Иначе, клавиша Tab будет передвигать фокус на следующий виджет. При неактивном флажке Can Tab, пользователь не может производить табуляцию из текстового редактора. Но при установке флажка Tab Requires Control Key, пользователь может производить табуляцию из текстового редактора клавишами Ctrl + Tab.

Порядок, в котором клавиша Tab продвигает центр, определяется порядком, в котором виджеты перечисляются в иерархическом списке виджетов в окне инструментов холста. Чтобы изменить порядок табуляции, нужно выбрать виджет, а затем нажать одну из кнопок упорядочения: Bring to Front (на первую позицию табуляции) Bring Forward (вперед на одну позицию табуляции) Send Backward (назад на одну позицию табуляции) Send to Back (на последнюю позицию табуляции)

Для составных частей, сначала устанавливается порядок табуляции внутри группы, а затем группа перемещается в соответствующую позицию.

4.3 Открытие и закрытие окна приложения

После того, как приложение создано, его можно открыть и протестировать. Обычный способ открыть приложение состоит в том, чтобы открыть одну из спецификаций его интерфейса пользователя, обычно открыть его основное окно. Когда имя спецификации основного окна приложения — #windowSpec, для этого надо просто послать сообщение open классу модели приложения, например, AddressBook open.

Если спецификации основного окна была во во врем инсталляции модели приложения изменена и имеет другое имя, надо послать классу модели приложения сообщение openWithSpec: с символом определенной спецификации окна приложения в качестве аргумента. Эти выражения могут, например, выполняться в рабочем окне.

Рассмотрим детально ту последовательность событий, которая происходит после того, как класс прикладной модели получит сообщение open или openWithSpec:. Понимание этой последовательности существенно для инициализации и понимания некоторых других операций, имеющих отношение к интерфейсу пользователя.

Процесс построения и открытия интерфейса происходит в несколько этапов. После каждого этапа, модель приложения может вмешаться в процесс, чтобы внести в него изменения. Вот эти этапы:

1. Метод open посылает классу сообщение new, создавая новый экземпляр класса прикладной модели AddressBook.

2. Созданному экземпляру класса AddressBook посылается сообщение initialize. Если метод initialize был определен в классе модели приложения, то именно он будет выполнен. Если такого определения нет, то будет выполняться наследуемый метод initialize. По умолчанию этот метод ничего не делает и его единственная цель — обеспечить благоприятную возможность разработчику выполнить некото-

рые операции в критическом месте процесса. Метод initialize почти всегда используется разработчиками для корректной инициализации переменных экземпляра модели приложения. Чтобы гарантировать инициализацию, предусмотренную суперклассами, метод инициализации модели приложения должен начинаться с выражения super initialize.

3. Затем экземпляр модели приложения создает экземпляр класса UIBuilder — компоновщик интерфейса, который получает всю информацию об интерфейсе из метода windowSpecs (или его аналога) и начинает строить его. Но прежде модель приложения посылает себе сообщение preBuildWith: aBuilder, где aBuilder — созданный компоновщик интерфейса. Если метод preBuildWith: определен в классе модели приложения, он будет выполнен, иначе будет выполнен пустой метод preBuildWith:, наследуемый из класса ApplicationModel. Этот еще одно место, где разработчик может вмешаться в процесс построения интерфейса. Большинство приложений не должно вмешаться в процесс открытия окна с помощью метода preBuildWith: aBuilder. Но с помощью этого метода можно настроить работу компоновщика приложения специальным образом.

4. После выполнения метода preBuildWith: компоновщик интерфейса строит интерфейс пользователя в памяти, используя метод windowSpec: (или его аналог). Он собирает информацию о размерах окна, о метках и виджетах, создает связи виджетов с их моделями значений и создает экземпляр класса UIPolicy, который поможет построить окно, ориентируясь на операционную систему. Объект «политики интерфейса» определяет, какой стиль будет использоваться для рисования окна, позволяя выбирать между стилями Microsoft Windows, Macintosh и Motif.

5. После завершения построения окна в памяти, модель приложения посылает себе сообщение postBuildWith: aBuilder, еще один метод, с помощью которого разработчик может внести последние штрихи в уже созданный интерфейс. Наследуемая версия этого также ничего не делает. Разработчик на этой стадии, через метод postBuildWith:, используя компоновщика интерфейса, может обращаться к окну и любым его именованным виджетам (к тем виджетам, для которых были заданы идентификаторы ID). Обычно метод postBuildWith: используется для того, чтобы скрыть или отключить некоторые виджеты.

6. В заключение, модель приложения рисует окно на экране, но прежде

де, чем передать управление пользователю, посылает себе сообщение postOpenWith: aBuilder. Это последний метод для вмешательства в процесс (его наследуемое определение также ничего не делает). В этом методе приложение может использовать компоновщика интерфейса для того, чтобы обратиться к окну и его виджетам: в это время эти объекты уже отображены на экране. После выполнения метода postOpenWith: aBuilder, приложение становиться доступным пользователю через окно интерфейса.

В приложении можно использовать любой из методов, позволяющих вмешаться в процесс открытия приложения, все или ни одного. Одного и того же эффекта можно достичь, используя разные методы.

По умолчанию, когда классу модели приложения посылается сообщение open выполняются все указанные этапы. Но можно посылать вновь созданному экземпляру класса модели приложения сообщение allBut-OpenInterface: #windowSpec, чтобы выполнить все этапы до открытия окна на экране, и позже отдельно послать сообщение finallyOpen, чтобы открыть окно интерфейса.

Если экземпляр класса модели приложения создан, то чтобы открыть приложения можно послать ему сообщение openInterface: #windowSpec или open. Это полезно, например, тогда, когда надо повторно использовать уже существующий экземпляр, а не создавать новый.

Перед тем как закрыть приложение часто нужно выполнить некоторые завершающие действия. Например, приложение по обработке текстов должно спросить пользователя, следует ли сохранить или отбросить сделанные в редактируемом тексте изменения. Другое общее действие — уничтожение установленных зависимостей между объектами.

Чаще всего пользователь выходит из приложения, используя команду меню (например, Exit) или другой виджет интерфейса (например, кнопку Quit), тогда модель приложения может корректно выполнить процедуру выхода, пользуясь методом, содержащим любые требуемые процедуры по завершению работы приложения. Когда модель приложения запускает одно или несколько окон, можно закрыть их (все сразу, если их несколько), посылая приложению сообщение closeRequest. В примере ниже модель приложения закрывает все связанные с ней окна.

```
| editor |
editor := Editor2Example new.
editor openInterface: #windowSpec.
(Delay forSeconds: 1) wait.
editor closeRequest.
```

Когда окно приложения хотят закрыть, оно сначала посылает сообщение changeRequest своей модели приложения. Если модель возвращает false, окно не будет закрываться; если же модель возвращает true, окно продолжит процесс закрытия. Таким образом, модель имеет возможность проверить, что ничего повреждающего приложение не произойдет, если окно будет закрыто. Как и в методе initialize, чтобы выполнить все необходимые операции, которые может реализовывать родительский класс, в методе changeRequest модели приложения следует вызвать наследуемую версию сообщения changeRequest.

Окно — относительно «дорогой» объект, поскольку содержит визуальный компонент, который часто очень велик. Когда приложение должно неоднократно открывать и закрывать одно и то же окно, нет необходимости каждый раз создавать его заново. Вместо этого, можно выполнить сообщение unmap (не отображать), которое скроет окно без его разрушения. Позже, чтобы отобразить его, нужно просто выполнить сообщение map (отобразить).

```
| win |
win := (Editor2Example open) window.
win display.
(Delay forSeconds: 1) wait.
win unmap.
(Delay forSeconds: 1) wait.
win map.
```

4.4 Контрольные вопросы

1. Какие окна открываются при вызове инструмента UIPainter?
2. Какие визуальные объекты можно размещать на холсте? Где они располагаются?
3. Какие задачи при построении интерфейса пользователя решает панель свойств?
4. В какой класс в иерархии устанавливается созданный холст?
5. Какое стандартное имя используется для метода класса, хранящего описание (спецификации) холста?
6. Какие виджеты используют аспектную переменную? В каком поле ввода панели свойств для виджета холста вводится имя, определяющее аспектную переменную и метод доступа к ней?

7. Какие виджеты вызывают на исполнение метод действия? В каком поле ввода панели свойств для виджета холста вводится имя, определяющее метод действия?
8. Какую роль в определении аспектных переменных и методов действия играет кнопка Define в окне инструментов холста?
9. Как из инструмента UIPainter открыть для выполнения созданный холст?
10. Какое из свойств виджета используется в приложении в методах доступа к этому виджету?
11. Свойства какой страницы окна инструментов холста используется для установки места окрытия холста на экране и его размеров?
12. Как добавить в окно панель меню?
13. Свойства какой страницы окна инструментов холста используется для установки цветов самого холста и каждого его виджета?
14. Как установить размеры виджета холста?
15. Как изменить шрифт, используемый виджетом холста?
16. Что такое «порядок табуляции» виджетов? Как его установить и изменить?
17. Как открыть для выполнения созданный интерфейс вне инструмента UIPainter?
18. Из каких этапов состоит процесс построения и открытия интерфейса пользователя на экране?
19. В каких точках процесса построения и открытия интерфейса пользователя можно в него вмешаться, чтобы внести в него изменения?
20. Какой метод из класса ApplicationModel позволяет закрыть окно приложения?

Проектные задания к модулю I

1. Описать различия в механизмах зависимости, реализованных в классах Object и Model. Указать новые сообщения, реализованные в классе Model.
2. Нарисовать схему взаимодействия в рамках классической архитектуры MVC между моделью, представлением и его контроллером.
3. Нарисовать схему взаимодействия в рамках архитектуры MVC, с учетом изменений, вносимых системой VisualWorks, между моделью, моделью предметной области, представлением и его контроллером.
4. Создать простой рисунок, используя инструмент Image Editor.
5. Используя инструмент ResourseFinder, открыть и исследовать примеры приложений с графическим интерфейсом пользователя, поставляемые с системой VisualWorks.

Модуль II

Классы, используемые при построения GUI

Цель модуля — описать библиотечные классы Dialog и Menu, обычно используемые при построении графического интерфейса пользователя, а также особенности построения специальных диалоговых окон и меню.

Глава 5

Диалоговые окна

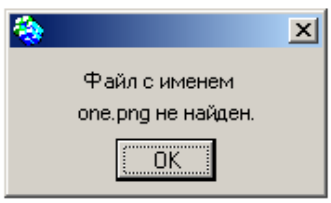
Диалоговые окна — специальные окна, отображающие сообщения или запрашивающие у пользователя некоторую информацию, необходимую для работы приложения. VisualWorks предоставляет достаточно методы отображения многих разновидностей диалоговых окон, а также полный набор возможностей для создания новых диалоговых окон. Простые диалоговые окна создаются посылкой сообщений классу Dialog. Для сложных диалоговых окон и окон, создаваемых пользователем, используются методы класса SimpleDialog, упрощающее их создание.

5.1 Стандартные диалоговые окна

Стандартные диалоговые окна очень просто встраиваются в создаваемое приложение. Обычно, нужно только посылать такое сообщение классу Dialog, которое построит нужную разновидность диалогового окна, возвращающего требуемую приложением информацию.

Отображение предупреждений

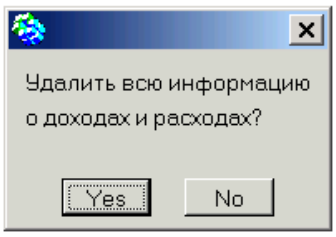
Диалоговое окно, содержащее предупреждение, отображает простой текст, который может включать символы возврата каретки и использовать текстовые стили. Предупреждающее диалоговое окно отображается на экран при посылке классу Dialog сообщения warn:. Когда пользователь нажимает в таком окне кнопку OK, сообщение возвращает nil.




```
Dialog
        warn: 'Файл с именем
              one.png не найден.' withCRs
```

Вопрос с ответом Yes/No (Да/Нет)

Часто приложение должно задать пользователю вопрос, предполагая короткий ответ — "yes (да)" или "no (нет)", например, запросить подтверждение на выполнение операции, которая может иметь непредвиденные побочные эффекты. Такое диалоговое окно называют окном подтверждения (confirmer). Обычно, задаваемый окном вопрос — фраза, составленная таким образом, что ответ "Да" вызывает некоторое действие, продолжающее работу. Окно подтверждения обычно отображается на экране посредством посылки классу Dialog сообщения confirm:, со строкой, содержащей вопрос.




```
Dialog
        confirm: 'Удалить всю информацию\
                 о доходах и расходах?\'
                 withCRs.
```

Если пользователь нажимает "Yes", диалоговое окно возвращает true, а если нажимает "No", диалоговое окно возвращают false. Ответ по умолчанию — "Yes". В опасных ситуациях, этот ответ плох. Чтобы изменить ответ по умолчания, надо послать классу Dialog сообщение

```
confirm:initialAnswer:.
```

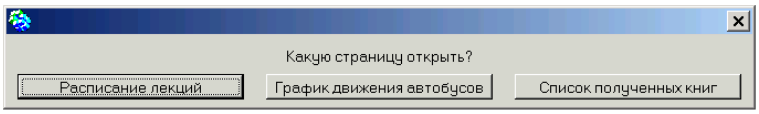
Второй аргумент — true (тогда ответ по умолчанию — "Yes") или false (тогда ответ по умолчанию — "No").

Вопрос, предполагающей несколько возможных ответов

Часто нужен ответ, отличающийся от "Yes/No" или требующий больше (но не очень много) вариантов ответа. Такую возможность обеспечивает диалоговое окно с множественным выбором. Варианты ответов выстраиваются в виде горизонтальной строки кнопок. Сообщение, которое создает диалоговое окно с множественным выбором, определяет символ для каждого варианта ответа. Когда пользователь нажимает одну из кнопок, сообщение возвращает приложению соответствующий символ.

Чтобы отобразить на экране диалоговое окно с множественным выбором, надо послать классу Dialog сообщение choose:labels:values:default:. Аргумент ключевого слова choose: — вопрос. Аргумент ключевого слова

labels: — массив тех строк, которые нужно отобразить на кнопках ответа. Аргумент ключевого слова values: — массив символов, которые нужно использовать в качестве возвращаемых значений после нажатия пользователем соответствующей кнопки ответа. Аргумент ключевого слова default: — символ, который ассоциируется с желаемым ответом по умолчанию.

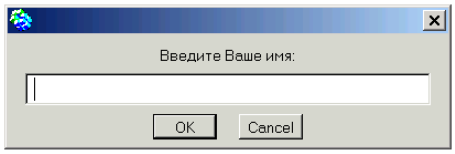



```
Dialog
        choose: 'Какую страницу открыть?'
        labels: #('Расписание лекций'
                  'График движения автобусов'
                  'Список полученных книг')
        values: #(#lectures #graphic #books)
        default: #lectures
```

Вопрос, требующий текстового ответа

Чтобы запросить у пользователя короткий текстовый ответ, используются диалоговые окна, которые отображают вопрос (метку) и поле ввода для заполнения (fill-in-the-blank).

Чтобы открыть самое простое такое диалоговое окно, надо послать классу Dialog сообщение request: с аргументом в виде строки, содержащей вопрос.




```
Dialog
    request: 'Введите Ваше имя:'
```

В таком окне по умолчанию в поле ввода появляется пустая строка. Когда пользователь вводит строку-ответ и нажимает кнопку OK, диалоговое окно возвращает введенную строку. Если пользователь нажимает кнопку Cancel, по умолчанию возвращается пустая строка.

Чтобы указать ответ по умолчанию, нужно послать классу Dialog сообщение request:initialAnswer: с ответом по умолчанию в виде строки, являющейся аргументом второго ключевого слова.

```
Dialog request: 'Введите Ваше имя:'
            initialAnswer: 'Юрий'
```

Если пустая строка не подходит как ответ на нажатие пользователем кнопки Cancel, и в этом случае можно определить альтернативное действие или значение. Для этого надо послать классу Dialog сообщение request:initialAnswer:onCancel:, с блоком в качестве аргумента третьего ключевого слова, содержащим необходимые действия, или значение, которое будет возвращено окном.

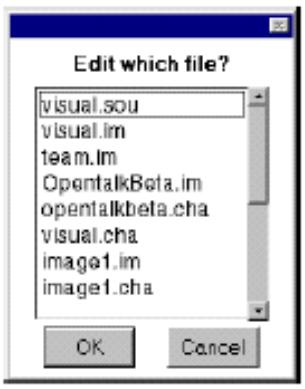
```
Dialog request: 'Введите Ваше имя:'
        initialAnswer: 'Юрий'
        onCancel: [self defaultCancelName].
```

Запрос имени файла

Диалоговое окно, запрашивающее имя файла, меняется от платформы к платформе. В Windows используется привычное системное окно для работы с файлами. Чтобы открыть диалоговое окно для запроса имени файла, надо послать классу Dialog сообщение requestFileName:, с аргументом в виде строки (в Windows можно даже с пустой строкой).

В некоторых ситуациях, может существовать естественное имя файла по умолчанию, как, например, при сохранении отредактированного файла. Чтобы определить имя файла по умолчанию, надо послать классу Dialog сообщение requestFileName:default:, со строкой в качестве аргумента второго ключевого слова, содержащей имя такого файла.

Выбор из списка



Диалоговое окно для выбора элементов из списка отображает список выбираемых элементов. Каждый элемент списка связан со значением, подобно пункту меню, и приложение может или вставить выбранное значение в хранитель значения, или запустить на выполнение некоторую операцию. По умолчанию, такое диалоговое окно содержит только список элементов, кнопки OK и Cancel, но можно добавить любые другие кнопки.

Чтобы открыть диалоговое окно для выбора элементов из списка, надо послать классу Dialog сообщение choose:fromList:values:lines:cancel: .

Аргумент ключевого слова choose: — строка, содержащая вопрос или подсказку. Аргумент ключевого слова fromList: — набор строк, каждая

из которых представляет метку для элемента списка. Аргумент ключевого слова values: — набор того же размера, что и аргумент для fromList:, который содержит значения, связанные с элементами списка. Аргумент ключевого слова lines: — целое положительное число, указывающее максимальное число отображаемых на экране элементов списка (для длинного списка). Аргумент ключевого слова cancel: — блок, содержащий те операции, которые надо выполнить, когда пользователем нажимается кнопка Cancel.

```
| files response |
files := Filename defaultDirectory directoryContents
            reject: [:name | name asFilename isDirectory].
response := Dialog
        choose: 'Edit which file?'
        fromList: files
        values: files
        lines: 8
        cancel: [^ nil].
response asFilename edit.
```

В примере, если файл выбран, открывается окно для его редактирования. Если выбранного файла нет, возвращается nil.

Запрос пароля

Приведем пример использования класса SimpleDialog и построим диалоговое окно для ввода пароля, создавая метод для сборки такого диалогового окна. Обратите внимание на использование типа #password, который маскирует ввод, заменяя буквы звездочками. Такое окно возвращает ассоциативную пару, ключ которой — введенное имя пользователя, а значение — введенный пароль.

```
| user pass |
(SimpleDialog initializedFor: nil)
    setInitialGap;
    addMessage: 'Username' textLine: (user := '' asValue) boundary: 0.4;
    addGap;
    addMessage: 'Password' textLine: (pass := '' asValue)
    type: #password boundary: 0.4;
    addGap;
    addOK: [true];
    openDialog.
```

user value − > pass value

5.2 Связь с главным окном

По умолчанию, встроенные диалоговые окна используют системные установки для отображения цветов и других составляющих. Если приложение использует набор специальных установок, можно сделать так, чтобы диалоговые окна подражали по цвету и внешнему виду главному окну. Кроме того, некоторые оконные системы создают визуальные связи между диалоговым окном и вызвавшим его главным окном.

Адаптация к виду главного окна

Чтобы диалоговые окна не выпадали из общего рисунка приложения, они должны во всем напоминать главное окно. Стандартные диалоговые окна имеют специальный вариант открывающего сообщения, которое заканчивается ключевым словом for:. Его аргумент — обычно текущее окно. Если предположить, что есть окно приложения с кнопкой действия Confirm (Подтвердить), то открытие диалогового окна подтверждения может быть задано так:

```
confirm
    ^Dialog
        confirm: 'Really erase all memories of adolescent period?' withCRs
        initialAnswer: false
        for: Window activeController view.
```

Связь диалогового окно с активным окном

Можно связывать диалоговое окно с окном, активным в данный момент, пользуясь сообщением warn:for:. Главное окно — это обычно главное окно приложения, к которому модель приложения может обращаться через объект self mainWindow (self builder window).

Чтобы связать диалоговое окно с другим окном, надо послать классу SimpleDialog сообщение useColorOveridesFromParent:. Его аргумент true откроет диалоговое окно с цветом, соответствующим связанному с ним активному окну, в дополнение к общему виду окна[1]. Затем в открывающем сообщении, посылаемом классу Dialog, аргументом ключевого слова for: должно быть главное окно.

[1]Обратите внимание, что слово "Overides" при определении метода написано с орфографической ошибкой (overwrites) и, следовательно, с орфографической ошибкой должно писаться при его использовании.

```
| masterWindow |
SimpleDialog useColorOveridesFromParent: true.
masterWindow := ScheduledWindow new.
masterWindow background: ColorValue yellow.
masterWindow open.
Dialog
    warn: 'This dialog has a yellow background, too.'
    for: masterWindow.
masterWindow sensor eventQuit: nil.
```

Чтобы вернуть поведение класса SimpleDialog к поведению по умолчанию, надо послать классу сообщение

```
useColorOveridesFromParent: false.
```

5.3 Создание диалогового окна пользователем

Когда стандартных диалоговых окон не достаточно, можно создавать собственные окна, используя инструмент Canvas. Одна из методик создания собственного диалогового окна обычно состоит в создании отдельной модели приложения для диалогового окна в подклассе класса SimpleDialog. Холст диалогового окна устанавливается в этот подкласс, а затем подкласс программируется так, чтобы обеспечить модели значений, действия и ресурсы, необходимые виджетам диалогового окна. Эта методика позволяет повторно использовать созданное диалоговое окно в других приложениях.

Но можно создавать собственное диалоговое окно в классе строящейся модели приложения сохраняя его спецификацию в методе с оригинальным именем. Такое окно также создается как экземпляр класса SimpleDialog, который имеет собственного составителя интерфейса для установки виджетов диалогового окна. В этой методике составитель диалогового окна получает все необходимые ему модели значений, действия и ресурсы для виджетов из модели приложения. Затем можно открывать созданную спецификацию интерфейса диалогового окна, посылая модели приложения сообщения openDialogInterface: с символом — именем спецификации интерфейса диалогового окна, например,

```
self openDialogInterface: #memoryZonesDialog.
```

При этом, обратите внимание, кнопки, значением свойства Action которых является символ #accept или #cancel, получают свои действия из экземпляра SimpleDialog. Эти предопределяемые действия весьма полезны для программирования поведения кнопок OK и Cancel диалогового

окна. Следовательно, если в модели приложения также определяются методы с символами #accept или #cancel, они будут проигнорированы.

Если необходимо, можно создавать динамические диалоговые окна (см.[7, p. 7-12 – 7-13 (190 – 191)]).

5.4 Контрольные вопросы

1. Какую роль в приложении обычно играют диалоговые окна?
2. Методы какого класса следует использовать для создавания программистом диалоговых окон?
3. Какие стандартные диалоговые окна определяются классом Dialog?
4. Как создать предупреждающее диалоговое окно?
5. Как создать диалоговое окно подтверждения с кнопками Yes/No (Да/Нет)?
6. Как создать диалоговое окно подтверждения с множественным выбором?
7. Как создать простое диалоговое окно, требующее текстового ответа от пользователя?
8. Как в среде Windows создать диалоговое окно, запрашивающее имя файла?
9. Как создать диалоговое окно для выбора элементов из списка?
10. Как создать диалоговое окно, являющееся экземпляром класса SimpleDialog, для ввода пароля?
11. Как установить связь диалоговых окон с главным окном приложения?

Глава 6

Меню

Система VisualWorks поддерживает работу строковых меню, кнопочных меню и всплывающих меню. Все три типа используют экземпляр класса Menu. Редактор меню (Menu Editor) — инструмент системы для создания меню, но, при желании, можно создавать меню и программно.

Строковое меню и всплывающее меню по умолчанию посылают сообщение модели приложения, в то время как кнопочное меню по умолчанию размещает значение в своем хранителе значения аспекта. При построение меню для столь разных поведений имеет значение то, что при построении меню назначается для каждого пункта меню.

С системой поставляется множество примеров меню. Для знакомства с которыми следует загрузить соответствующие пакеты командой System → Load Parsels Named..., а затем изучить их, просматривая через Resourse Finder (например, примеры MenuCommandExample, MenuEditorExample).

6.1 Создание меню

Использование редактора меню

Редактор меню (см. рисунок 3.3) позволяет минимизировать явное программирование. Но, будучи заменой программированию, он менее гибок, чем другие методы создания меню, и не может использоваться, например, для создания меню, которые изменяются во время выполнения. Опишем процесс построения меню редактором.

1. Открыть редактор меню, выбирая в основном окне системы команду Painter → Menu Editor.
2. Выбирая команду Edit → New Item, добавить элемент меню верхнего уровня и отредактировать его свойства на страницах Basic, Details, State.

Когда добавляется пункт меню, он отображается как <new item> (новый элемент). Следует отредактировать эту метку в поле свойства Label: String, давая пункту меню такую метку, которую оно должно отображать.

Чтобы связать с пунктом меню клавишу его быстрого выбора (мнемонику), следует вставить в метку амперсанд (&) перед той литерой, которую собираетесь использовать как мнемонику. Например, запись метки в виде &Item сделает букву I мнемоническим литералом для данного элемента меню, а сама буква будет выделена в пункте меню подчеркиванием.

Последующее использование команды Edit → New Item добавит новый элемент на тот же самый уровень, на котором находиться выбранный элемент меню.

3. Чтобы добавить подменю, надо выбрать команду Edit → New Submenu Item. Новый элемент тогда помещается ниже и со сдвигом вправо от выбранного элемента меню.

4. В свойстве с именем Value для каждого элемента следует или (1) ввести селектор сообщения, или (2) оставить поле незаполненным, если элемент — заголовок следующего меню.

   Отметим, что строковым и всплывающим меню селектор сообщения посылается модели приложения. Для кнопочного меню, селектор должен быть аспектным методом доступа, который модифицирует хранитель значения аспекта.

5. Определить для пункта меню, если это нужно, горячую клавишу. Горячая клавиша позволяет пользователю выбирать этот пункт меню, нажимая указанную клавишу при нажатой и удержании одной или нескольких клавиш <Ctrl>, <Shift>, <Alt>. Чтобы определить горячую клавишу, надо ввести соответствующий литерал в поле ввода Shortcut character: на странице Delails, и выбрать переключатель перед модифицирующей клавишей. Такой литерал будет отображаться в меню.

6. Если нужно, можно откорректировать уровни расположения каждого пункта меню командами Move → Left, Move → Right, а его место в меню — командами Move → Up, Move → Down.

7. Если во время работы приложения пункт меню должен становиться доступным или недоступным, на странице State в поле ввода Enablement Selector следует ввести селектор сообщения. Метод, ко-

торый следует написать для введенного селектора должен возвращать логическое значение (true или false).

8. Чтобы в меню использовать индикатор выбора пункта меню — галочку, появляющуюся в пункте меню, следует на странице State в поле ввода Indication Selector ввести селектор сообщения. Метод, который следует написать для введенного селектора, должен возвращать логическое значение (true или false). Кроме того, для каждого пункта меню можно выбрать одну из радио-кнопок, определяя каким будет данный пункт меню изначально.

9. Выбрать команду Menu → Install, чтобы установить спецификацию меню в метод класса из категории resources (ресурсы) модели приложения.

10. Создать все требуемые методы, имена сообщений которых были введены при определении пунктов меню, в том числе и в полях свойства Value.

Как пример, создадим всплывающее меню для списка выбора. Меню будет состоять из двух элементов: (1) add, который откроет диалоговое окно, позволяя пользователю ввести новую строку, затем добавляемую в список выбора, (2) delete, который позволит пользователю удалить в данное время выбранный элемент списка, если таковой есть.

Чтобы решить задачу, откроем инструмент UIPainter установим на холст виджет списка, установим, определим его аспект (в поле ввода Aspect) как list, а свойство Menu списка как listHolderMenu. Установим холст в класс ListWithPopup1, подкласс ApplicationModel и воспользуемся командой Define, определяя аспектный метод list.

Чтобы создать всплывающее меню для списка, откроем редактор меню командой меню Tools → Menu Editor из окна инструментов холста или той же командой из всплывающего меню холста. Введем пары "метка – команда". Метку определяем, изменяя строку <new item>, возникающую после нажатия кнопку Insert Item (второй слева в панели меню редактора). Символ вводим в поле Value — это имя метода экземпляра в классе прикладной модели, который будет выполняться, когда пользователь выбирает связанный с ним пункт создаваемого меню.

После ввода двух пар "метка – команда" (add – addItem, delete – deleteItem), выполним в окне редактора меню команду Menu → Install. Откроется диалоговое окно. Для класса ListWithPopup1 введем в нижнем поле ввода имя обращающегося к меню метода (listHolderMenu), ранее указанного как значение свойства Menu для списка выбора. Нажмем кнопку OK, создавая ресурсный метод, описывающий построенное меню,

и сохраняя его в протоколе resources методов класса. Выполняя команду Test редактора меню, можно сразу же увидеть созданное меню.

Теперь можно открыть приложение и проверить работу всплывающего меню. При нажатии в области поля ввода правой кнопки мыши появится созданное меню, но работать его команды не будут, поскольку ещё не определены соответствующие методы. Определим их. Метод, реализующий команду add должен получить новую метку от пользователя, добавить её в список меток, и определить её в виджет списка через сообщение list:

```
addItem
    "Запросить новую метку, и добавить её в
    список."
    list list: ((list list) add: (Dialog request:
                    'Enter new label' initialAnswer: ''); yourself)
```

Напомним, что сообщение add: возвращает свой аргумент, а не изменённый список. Чтобы возвратить список, следует после посылки списку сообщения add: послать ему сообщение yourself.

Метод delete сначала должен проверить, сделал ли пользователь выбор. Если нет выбранного элемента в списке, метод ничего не должен делать. Если выбор сделан, метод должен получить текущий список, удалить текущий выбранный элемент и определить новый список через сообщение list:.

```
deleteItem
    "Удалить выбранный элемент списка."
    | selection |
    selection := list selection.
    selection isNil ifTrue: [^ self].
    list list: ((list list) remove: selection; yourself)
```

Создание всплывающего меню списка завершено.

Использование объекта MenuBuilder

При программном построении меню удобно пользоваться экземпляром класса MenuBuilder — инструментом, который формирует объект Menu. Этот инструмент имеет множество методов, полезных для формировании меню, что позволяет сократить объем явного программирования. Но можно строить меню, непосредственно используя экземпляры классов Menu и MenuItem. Но такая методика весьма трудоемка, поскольку каждая деталь должна программироваться явно. Этот подход следует

применять только в случае острой необходимости, когда не достаточно возможностей, обеспечиваемых объектом MenuBuilder.

Чтобы объяснить, как использовать экземпляр класса MenuBuilder, вернемся к примеру из предыдущего раздела. Отметим, что построенное в нем меню списковой панели несовершенно: когда в списке нет выбранного элемента, пользователью нечего удалять и в этом случае меню не должно предлагать команду delete. Для надлежащих операций нужны два различных всплывающих меню: одно с командами add и delete, а другое — без команды delete. Каждое из этих меню должно отображаться в соответствующей ситуации. Такое динамическое меню не может быть создано редактором меню.

Построим в UIPainter тот же интерфейс пользователя, определяя те же виджеты и их свойства, инсталлируем его в класс Установим холст в класс ListWithPopup2 и воспользуемся командой Define, определяя аспектный метод list. Откроем на классе ListWithPopup2 системный браузер и добавим в его определение новую переменную экземпляра listHolderMenu, ссылающуюся на хранитель значения — экземпляр класса ValueHolder, который и будет хранить нужное меню. Управляться содержимое хранителя значения будет через сообщение value:. В нашем случае, меню изменяется всякий раз, когда пользователь выбирает или отменяет выбор элемента из списка. Чтобы реализовать такое поведение, зарегистрируем заинтересованность в любом изменении выбора из списка через сообщение onChangeSend:to:. Когда произойдет изменение выбора, сообщение изменения пошлет сообщение value: с соответствующим меню созданному хранителю значения.

Код, решаюший все перечисленные задачи, поместим в метод инициализации:

```
initialize
    "Назначить начальное всплывающее меню
    для списка и за-
    регистрировать свой интерес в изменении
    выбора из списка."
    listHolderMenu := self menuWithoutDelete asValue.
    list := SelectionInList new.
    list selectionIndexHolder onChangeSend: #changedMenu to: self.
```

Обратим внимание на сообщение asValue, которое создаёт хранитель значения. Подробно этот объект мы рассмотрим позднее в разделе 7.2.

Метод changedMenu должен проверить, сделан ли пользователем выбор из списка, и подставить соответствующее меню. Сообщение value:, используемое в методе, устанавливает новое значение в хранитель зна-

чения.

```
changedMenu
    "Контекст, возможно, изменился, определить
    соответствующее всплывающее меню."
    self listHolderMenu value:
            (list selection isNil
                    ifTrue: [self menuWithoutDelete]
                    ifFalse: [self menuWithDelete]).
```

Здесь listHolderMenu — сообщение меню, которое было определено в окне свойств виджета списка. Но соответствующий метод вместо кода, содержащего результат построения меню редактором меню, должен теперь просто возвращать хранитель значения, содержащий меню.

```
listHolderMenu
    ^ listHolderMenu
```

Методы, создающие меню, используют объект MenuBuilder и определяются следующим образом

```
menuWithDelete
    "Определить меню с командами add: и
    delete:."
    | menuBuilder |
    menuBuilder := MenuBuilder new.
    menuBuilder add: 'add' -> #addItem;
                    add: 'delete' -> #deleteItem.
    ^ menuBuilder menu


menuWithoutDelete
    "Определить меню только с командой add:."
    | menuBuilder |
    menuBuilder := MenuBuilder new.
    menuBuilder add: 'add' -> #addItem.
    ^ menuBuilder menu
```

Методы #addItem и #deleteItem были определены в примере 1 и остаются неизменными.

Методы menuWithDelete, menuWithoutDelete демонстрируют стандартную схему использования объекта MenuBuilder:

1. Создать новый объект MenuBuilder, формирующий меню.

2. Обеспечить для объекта MenuBuilder информацию о формируемом меню через последовательность сообщений add: и, возможно, другие сообщения формирования меню. Как результат, объект MenuBuilder сформирует экземпляр класса Menu, содержащий объекты MenuItem.

3. Возвратить созданное меню, посылая сообщение menu объекту MenuBuilder.

В заключение отметим, что любое меню, независимо от того, создано оно программно или редактором меню, можно хранить в экземпляре класса ValueHolder — в аспектной переменной, указанной как свойство Menu виджета. Следовательно, программно заменять можно и меню созданные редактором меню.

При построении меню, каждая метка меню связывается с некоторым действием, которое может быть просто значением, командой или блоком действий. Проиллюстрируем подход создания меню с помощью объекта MenuBuilder примером, использующим блоки действий.

Блоки действий обычно используются в строковых и всплывающих меню, помещая значения, в соответствующие хранители, или в кнопочных меню, выполняя команду. Для кнопочного меню, чтобы выполнить команду, хранитель значения конфигурируется так, чтобы посылать сообщение perform: с значением, хранимым в хранителе, в качестве аргумента, когда изменяется выбранная команда такого меню.

Меню обычно создается методом класса из протокола resources модели приложения. Но ресурсный метод может быть и методом экземпляра, который полагается на данные, обеспеченные прикладной программой. Обычно такие методы используют для построения меню объект MenuBuilder. Для пояснений воспользуемся как примером классом MenuValueExample, в котором определена переменная экземпляра letter — хранитель значения, содержащий отображаемый текст.

1. Создать методы экземпляра, формирующие меню. В примере таких методов три — templatesMenuForMenuBar, templatesMenuForPopUp, templatesMenuForMenuButton. Каждый из них первым делом создает составителя меню, посылая сообщение new классу MenuBuilder.

2. В первых двух методах для каждого пункта меню послать составителю меню сообщение add: с ассоциативной парой в качестве аргумента. Ассоциативная пара связывает строку, представляющую метку пункта меню, с блоком, который выполняет требуемое действие. В случае использования меткой меню значения или команды, на месте блока необходим хранитель значения или селектор сообщения, соответственно.

Выделение мнемонической буквы для пункта меню производиться точно также, как и при вводе метки в редакторе меню — вставить в метку перед буквой амперсант (&), которая в меню будет выделяться подчеркиванием.

3. Чтобы добавить для метки меню последовательность горячих клавиш (сочетание клавиш), следует определится алфавитную клавишу и управляющую клавишу или клавиши. Управляющая клавиша может определяться индивидуально методами ctrlMask, shiftMask, altMask), посылаемыми классу InputState или объединяться, используя сообщение bitOr: Например, чтобы определить последовательность <Ctrl> + <Shift> + A как последовательность горячих клавиш, нужно добавить строки кода вида

```
menuItem shortcutKeyCharacter: $A.
menuItem shortcutModifiers:
        (InputState ctrlMask bitOr: InputState shiftMask).
```

4. Чтобы вставить подменю, нужно послать составителю меню сообщение beginSubMenuLabeled: с меткой подменю в качестве аргумента. Это начало определения подменю. Затем следует добавить элементы подменю, посылая, как и прежде, составителю меню сообщение add:. В конце определения подменю, послать составителю меню сообщение endSubMenu.

5. Получить меню от составителя меню, используя сообщение menu и возвратите его как результат создающего меню метода.

В рассматриваемом примере, все пункты меню добавляются в подменю строкового меню. Методы класса firstNotice, secondNotice, finalNotice возвращают тексты писем. Пример также иллюстрирует программное добавление графического образа в качестве метки меню и изменение цвета фона меню.

```
templatesMenuForMenuBar
    | mb menu submenu |
    "Создать строку меню из одного подменю."
    mb := MenuBuilder new.
    mb
        beginSubMenuLabeled: 'Templates';
        add: ' ' -> [self letter value: self class firstNotice];
        add: ' ' -> [self letter value: self class secondNotice];
        add: ' ' -> [self letter value: self class finalNotice];
```

```
    endSubMenu.
"Добавить как метки графические образы."
menu := mb menu.
submenu := (menu menuItemLabeled: 'Templates') submenu.
(submenu menuItemAt: 1) labelImage: (self class oneImage).
(submenu menuItemAt: 2) labelImage: (self class twoImage).
(submenu menuItemAt: 3) labelImage: (self class threeImage).
"Установить цвет фона."
submenu backgroundColor: ColorValue chartreuse.
^menu
```

В этом меню роль меток исполняют графические объекты, а не строки. Для этого пункту меню, строковая метка которого должна иметь по крайней мере один литерал (пусть даже только пробел), посылается сообщение labelImage:. Его аргумент — любой визуальный объект, но обычно — графический образ.

Метод templatesMenuForPopUp определяет всплывающее меню текстовой панели. Это простой и, после приведенных выше пояснений, понятный метод.

```
templatesMenuForPopUp
  "Создать всплывающее меню текстовой па-
  нели."
  | mb |
  mb := MenuBuilder new.
  mb
    add: 'First Notice' -> [self letter value: self class firstNotice];
    add: 'Second Notice' -> [self letter value: self class secondNotice];
    add: 'Final Notice' -> [self letter value: self class finalNotice].
   ^mb menu
```

6.2 Добавление меню в интерфейс

Чтобы добавить меню в интерфейс пользователя приложения, следует добавить в окно приложения или строковое меню, или виджет кнопки меню, или определить всплывающее меню виджета. Они эти меню определяются при формировании холста в окне свойств в поле ввода Menu, задавая имя ресурса, формирующего меню.

Построение в окне строки меню

Строка меню, расположенная, как правило, под заголовком вдоль верхнего края окна, представляется пользователю как набор отдельных меню, метки которых расположены вдоль верхнего края окна. Все меню и подменю фактически реализованы единым объектом — меню. Метки меню, отображаемые на строке меню, представляют элементы меню верхнего уровня, а их содержимое — подменю, связанные с пунктами меню верхнего уровня.

Рассмотрим как пример, поставляемый с системой класс MenuCommandExample, в котором окно имеет строку меню, состоящую из двух меток File, Help.

1. На холсте для окна, удостовериться, что ни один виджет не выбран.
2. В окне свойств выбрать флажок Enable в разделе Menu Bar на странице свойств Basics.
3. В поле ввода Menu, ввести имя метода класса, создающего меню — #fileMenu.
4. Создать указанный ресурсный метод класса для определения меню и все другие методы для операций, которые будут вызываться пунктами меню.

Как результат, ресурсный метод класса #fileMenu в классе MenuCommandExample имеет следующий код:

```
fileMenu
    "Создать меню приложения."
    | mb menu submenu |
    mb := MenuBuilder new.
    mb
        beginSubMenuLabeled: 'File';
        add: 'Open' -> #openFile;
        line;
        add: 'Add' -> #addFile;
        add: 'Delete' -> #deleteFile;
        endSubMenu.
    mb
        beginSubMenuLabeled: 'Help';
        add: 'Usage' -> #explainUsage;
        endSubMenu.
    "Определить клавиши быстрого выбора команд (мнемоники) без обращения к меню."
```

```
menu := mb menu.
submenu := (menu menuItemLabeled: 'File') submenu.
(submenu menuItemLabeled: 'Open') shortcutKeyCharacter: $O.
(submenu menuItemLabeled: 'Add') shortcutKeyCharacter: $A.
(submenu menuItemLabeled: 'Delete') shortcutKeyCharacter: $D.
^ menu
```

Формирование в окне кнопки меню

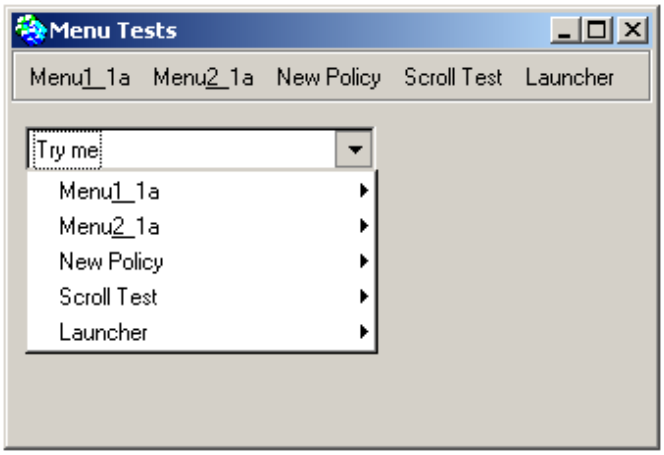


Рис. 6.1: Кнопочное меню примера MenuDemo

Кнопка меню — специальный виджет, который позволяет разместить в любом месте холста раскрывающееся меню. Кроме того, метка такого меню можно изменять, отражая текущую ситуацию в окне. По умолчанию, выбор элемента в кнопочном меню размещает значение пункта меню из хранителя значения аспекта этого виджета. В качестве примера, снова используем класс MenuCommandExample. Опишем процесс создания кнопочного меню. Напомним, что для внесения изменений на холст надо выполнять команду Accept и, в заключении инсталлировать холст в класс.

1. Разместить на холсте виджет MenuButton и в поле свойств String определить метку кнопочного меню.
2. В поле свойства Aspect виджета MenuButton ввести имя аспекта #action — хранителя значения, в котором будет храниться значение выбираемого пункта меню.
3. В поле свойства Menu виджета MenuButton ввести имя ресурсного метода создания меню (#fileMenu), который используется всеми виджетами меню. Будем предполагать, что этот метод был создан, как и методы действия для названных в меню команд.

4. Открыть системный браузер на классе MenuCommandExample. В метод initialize добавить строки инициализации хранителя значения и задействовать механизм зависимости, определяя заинтересованность приложения в изменении значения аспектной переменной.

```
initialize
    action := nil asValue.
    action onChangeSend: #performAction to: self.
```

Определить метод performAction:

```
performAction
    self perform: self action value.
```

Определить аспектный метод, как метод, просто возвращающий значение аспектной переменной

```
action
    ^action
```

Сообщение value возвращает данные из хранителя значения.

Остальные методы этого класса предлагаем читателю изучить самостоятельно.

Добавление в виджет всплывающего меню

Построение всплывающего меню для виджета списка было продемонстрировано в разделе 6.1. В ответ на нажатие кнопку $< Optrate >$ мыши отображают всплывающее меню и некоторые другие виджеты. В классе MenuCommandExample, чтобы воспользоваться уже созданным меню в качестве всплывающего меню списковой панели, достаточно ввести в поле свойств Menu этого метода имя уже существующего ресурсного метода меню.

В другом классе — MenuValueExample — определяется собственное всплывающее меню текстового редактора, для чего делается следующее:

1. В свойстве Menu виджета текстового редактора вводится имя метода templatesMenuForPopUp, возвращающего меню.

2. Используя системный браузер, создается метод меню (templatesMenuForPopUp). В меню, каждая метка соединяется с блоком, в котором аспектная переменная виджета letter модифицируется желаемым значением.

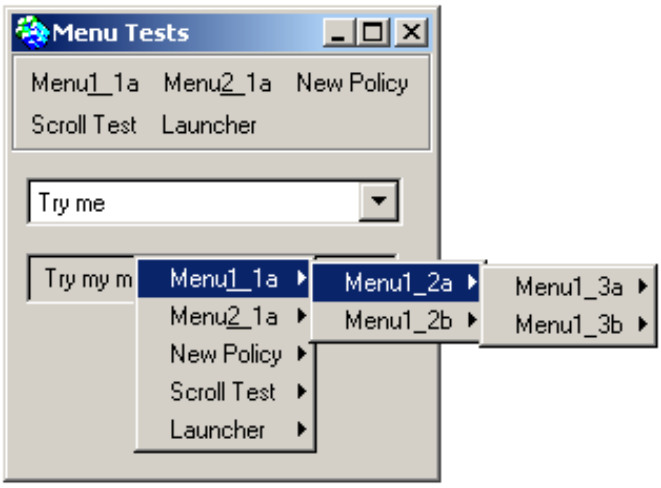


Рис. 6.2: Всплывающее меню примера MenuDemo

```
templatesMenuForPopUp
| mb |
  mb := MenuBuilder new.
  mb
    add: 'First Notice' -> [self letter value: self class firstNotice];
    add: 'Second Notice' -> [self letter value: self class secondNotice];
    add: 'Final Notice' -> [self letter value: self class finalNotice].
  ^mb menu
```

6.3 Программный доступ к меню

Отключение пункта меню

Приложения часто отключают ("делают серой") команду меню, когда её выбор при данном состоянии приложения не допустим. Это обычно происходит после проверки в приложении некоторого условия. Чтобы отключить пункт меню, надо послать этому пункту меню сообщение disable. Точно так же, чтобы сделать возможным доступ к пункту меню, надо послать ему сообщение enable.

В методе, который обеспечивает обращение к меню, надо сделать следующее:

1. Получить меню, посылая сообщение menuAt: составителю интерфейса builder с именем ресурсного метода меню в качестве аргумента.
2. Получить пункт меню:
   - для меню, определенного с помощью редактора меню, можно обращаться к его элементу через значение его свойства ID (именно-

го ключа), посылая меню сообщение atNameKey: с ключом, представленным экземпляром класса Symbol в качестве аргумента;

- для меню, определенного другим способом, можно обращаться к элементу меню по его метке, посылая меню сообщение menuItemLabeled: сострокой-меткой в качестве аргумента.

3. Получить подменю, посылая сообщение submenu к пункту меню, который представляет подменю.
4. Получить элемент подменю, посылая подменю сообщение menuItemLabeled: или atNameKey:, как и на шаге 2.

Чтобы отключить или разрешить доступ к пунктам меню, в методе configureMenu класса MenuCommandExample, обращаются к пунктам меню через сообщение menuItemLabeled:.

```
configureMenu
    "Отключить или разрешить доступ к пунктам
    меню в зависимости от того, выбран ли файл."

| menu submenu |
    menu := self builder menuAt: #fileMenu.
    submenu := (menu menuItemLabeled: 'File') submenu.
    self files selection isNil
      ifTrue: [
            (submenu menuItemLabeled: 'Open') disable.
            (submenu menuItemLabeled: 'Delete') disable]
      ifFalse: [
            (submenu menuItemLabeled: 'Open') enable.
            (submenu menuItemLabeled: 'Delete') enable]
```

Если предположить, что предыдущее меню создавалось, используя редактор меню, и определить свойство ID для каждого его пункта в вид системного имени, совпадающего посимвольно с меткой, то метод, аналогичный предыдущему должен использовать сообщение atNameKey:.

```
configureMenu
    "Отключить или разрешить доступ к пунктам
    меню в зависимости от того, выбран ли файл."

| menu submenu |
    menu := self builder menuAt: #fileMenu.
    submenu := (menu atNameKey: #File) submenu.
    self files selection isNil
```

```
ifTrue: [
    (submenu atNameKey: #Open) disable.
    (submenu atNameKey: #Delete) disable]
ifFalse: [
    (submenu atNameKey: #Open) enable.
    (submenu atNameKey: #Delete) enable]
```

Как вариант, можно оперировать на наборе пунктов меню. В примере MenuEditorExample, метод disableDarkColors получает набор пунктов данного меню, посылая меню сообщение menuItems. Затем посылает сообщение nameKey каждому пункту меню, чтобы получить его именной ключ, и, наконец, отключает каждый пункт меню, именные ключи которых находится в массиве darkColors.

Сокрытие пункта меню

Скрыть пункт меню иногда лучше, чем просто отключить его. Заблокированный элемент может вызвать у пользователя недоумение, если не ясно, почему пункт заблокирован. Когда же пункт вообще скрыть, нет и искушения его использовать. Такое решение должно приниматься во время проектирования интерфейса. Например, в программе редактирования документа, если нет открытого документа, могут быть доступными только команда открывающая документ и команда закрывающая приложение. Но, когда открывается некоторый документ, сразу же становятся доступными несколько меню с командами редактирования. Рассмотрим варианты программирования такой стратегии.

В примере MenuModifyExample строится приложение с двумя кнопочными меню, отображающими список должностей (jobTitlesMenu) и список доступных привилегий (benefitsMenu), который должен изменяться в зависимости от выбранной в первом меню должности. Это достигается следующим образом.

1. Получить меню и нужный пункт меню (может быть потребуется сначала получить подменю, чтобы добраться до нужного пункта).
2. Чтобы скрыть элемент, послать меню сообщение hideItem: с пунктом меню в качестве аргумента.
3. Чтобы показать элемент, послать меню сообщение unhideItem: с пунктом меню в качестве аргумента.

Реализуется эта стратегия в следующем методе.

```
adjustBenefitList
    "Скрыть те элементы списка benefit, которые
    должны быть не доступны для выбранного из
    списка Job Title должностного лица."
| bMenu item |
    bMenu := self builder menuAt: #benefitsMenu.
    item := bMenu menuItemLabeled: 'Golden Parachute'.

    "Только President (Президент) получает доступ
    к пункту меню Golden Parachute (Золотой Пара-
    шют)."
    self jobTitle value == #President
        ifTrue: [bMenu unhideItem: item]
        ifFalse: [bMenu hideItem: item].
```

Добавление нового пункта меню

Динамически можно добавить новый пункт в конец меню. В примере MenuModifyExample это достигается следующим образом.

1. Получить меню, посылая сообщение menuAt: составителю модели приложения, с ресурсным методом создания меню в качестве аргумента.
2. Послать меню сообщение addItemLabel:value:. Первый аргумент — строка, определяющая метку, второй аргумент — команда, значение или блок действия, соответствующие новой метке.

```
addTitle
    "Вызвать диалоговое окно для ввода нового за-
    головка и добавить его в список."
    | newTitle jMenu |
    newTitle := Dialog request: 'New title?'.
    newTitle isEmpty ifTrue: [^ self].

    jMenu := self builder menuAt: #jobTitlesMenu.
    jMenu addItemLabel: newTitle value: newTitle asSymbol.

    self jobTitle value: newTitle asSymbol.
```

Удаление элемента из меню

В том же классе MenuModifyExample динамическое удаление существующего пункта меню достигается, используя следующий алгоритм.

1. Получить меню и тот его пункт, который надо будет удалить (может быть потребуется получить подменю, чтобы добраться до нужного пункта).
2. Послать меню сообщение removeItem: с удаляемым пунктом меню в качестве аргумента.
3. В случае кнопочного меню, в котором отображается текущий выбор, в хранитель значения кнопочного меню отправить новое допустимое значение.

```
deleteTitle
    "Вызвать диалоговое окно для заголовка и уда-
    лить его из списка."
    | jMenu removableTitles title item |
    jMenu := self builder menuAt: #jobTitlesMenu.
    "Не разрешать удаления команды President)."
    removableTitles := jMenu labels
            reject: [ :nextTitle | nextTitle = 'President'].

    title := Dialog
            choose: 'Delete Title'
            fromList: removableTitles
            values: removableTitles
            lines: 8
            cancel: [^ nil]
            for: ScheduledControllers activeController view.

    item := jMenu menuItemLabeled: title.
    jMenu removeItem: item.
    "Если удаленная опция показывается, указать
    новый первый заголовок."
    self jobTitle value == title asSymbol
            ifTrue: [self jobTitle value: #President].
```

Подстановка другого меню

В примере MenuSwapExample всплывающее меню виджета списка меняется в зависимости от состояния приложения: если есть выбранный

элемент из списка предлагаемых цветов, отображается меню colorSelectedMenu, если выбранного элемента нет — меню nothingSelectedMenu. Добиться такого поведения можно следующим образом.

1. В свойстве Menu виджета списка ввести имя метода, который возвращает хранителя значения, содержащего всплывающее меню (menuHolder).
2. В модели приложения создать переменную экземпляра (menuHolder), чтобы хранить в ней указанный хранитель значения.
3. Создать метод (menuHolder), который просто возвращает значение переменной экземпляра.

```
menuHolder
    ^ menuHolder
```

4. Создать начальное меню (nothingSelectedMenu) и альтернативное меню (colorSelectedMenu).

```
nothingSelectedMenu
    | mb |
    mb := MenuBuilder new.
    mb add: 'Add At Bottom' -> #unimplemented;
        line;
        add: 'Delete All' -> #unimplemented.
    ^ mb menu

colorSelectedMenu
    | mb |
    mb := MenuBuilder new.
    mb add: 'Add Above' -> #unimplemented;
        line;
        add: 'Delete' -> #unimplemented;
        add: 'Delete All' -> #unimplemented.
    ^ mb menu
```

5. В методе инициализации initialize, получить начальное меню (nothingSelectedMenu), поместить его в хранитель значения, и назначить хранитель значения в переменную экземпляра (menuHolder). Принять меры к тому, чтобы вызывался изменяющий меню метод, когда происходят изменения в самом приложении (в методе инициализации эту задачу решает сообщение onChangeSend:to:).

```
initialize
    colors := SelectionInList with: ColorValue constantNames.
    colors selectionIndexHolder onChangeSend: #selectionChanged
    menuHolder := self nothingSelectedMenu asValue.
```

6. Создать метод (selectionChanged), который определяет, какое из меню должно использоваться, а затем помещает нужное меню в переменную menuHolder.

```
selectionChanged
    self colors selection isNil
        ifTrue: [self menuHolder value: self nothingSelectedMenu]
        ifFalse: [self menuHolder value: self colorSelectedMenu]
```

Отображения индикатора Вкл/Выкл (On/Off)

Часто в меню рядом с текстовой меткой полезно в качестве индикатора переключения вставлять галочку (√), а затем стирать ее. Такую методику можно использовать, например, чтобы моделировать в меню набор радио-кнопок, как это сделано в примере MenuValueExample.

1. Чтобы отобразить галочку перед меткой меню (включить — "on"), надо послать такому пункту меню сообщение beOn.
2. Чтобы стереть галочку перед меткой меню (выключить — "off"), надо послать такому пункту меню сообщение beOff.

```
setCheckMark
    "В всплывающем меню установить флажок, ука-
    зывая
    на отображаемый в настоящее время шаблон."
    | menu item |
    menu := self builder menuAt: #templatesMenuForPopUp.
    item := menu menuItemAt: 1.
    self letter value = self class firstNotice
          ifTrue: [item beOn] ifFalse: [item beOff].
    item := menu menuItemAt: 2.
    self letter value = self class secondNotice
          ifTrue: [item beOn] ifFalse: [item beOff].
    item := menu menuItemAt: 3.
    self letter value = self class finalNotice
          ifTrue: [item beOn] ifFalse: [item beOff].
```

Класс MenuSelectExample иллюстрирует, как можно включать один пункт меню ("on") и выключить ("off") все остальные, используя в качестве индикатора «жирную точку», а не галочку.

6.4 Контрольные вопросы

1. Какие типы меню поддерживает система VisualWorks?
2. Какой инструмент системы VisualWorks используется для создания меню?
3. Как открыть инструмент для создания меню?
4. Опишите процесс создания меню в редакторе меню.
5. Экземпляр какого класса удобно использовать при программном построении меню? Какова схема использования такого объекта?
6. Как встроить в интерфейс пользователя строкового меню?
7. Как встроить в интерфейс пользователя кнопки меню? Какой виджет позволяет это сделать?
8. Как встроить всплывающего меню в виджет интерфейса пользователя?
9. Как программно отключить пункт меню, когда ее выбор при данном состоянии приложения не допустим?
10. Как программно скрыть пункт?
11. Как программно динамически добавить новый пункт меню?
12. Какой алгоритм следует использовать, чтобы динамически удалить существующий пункт меню?
13. Как программно подменять меню в зависимости от состояния приложения?
14. Какой алгоритм следует использовать, чтобы иметь возможность в меню рядом с текстовой меткой устанавливать в качестве индикатора переключения специальную метку, а затем стирать ее?

Проектные задания к модулю II

1. Создать диалоговое окно, предупреждающее пользователя о невозможности выполнения запрошенной операции.
2. Создать диалоговое окно, требующее от пользователя ответа **«Да»** или **«Нет»** на вопрос о сохранении слеланных изменений.
3. Создать диалоговое окно, требующее от пользователя ответа на вопрос о том, какой тип документа создать — книга, отчет, статья презентация.
4. Создать диалоговое окно, требующее от пользователя ввода ответа на вопрос о занимаемой им должности.
5. Создать диалоговое окно, требующее от пользователя выбора из списка предпочитаемого им вида отдыха.
6. Пользуясь редактором меню создать стандартное меню редактирования текстового документа.

Модуль III

Примеры построения приложений с GUI

Цель модуля — показать, как использовать основные механизмы, заложенные в архитектуру MVC, инструменты (UIPainter, MenuEditor, ResourseFinder) и объекты (виджеты, диалоговые окна, меню) при создании простых приложений.

Глава 7

Приложение AddressBook

Построим приложение AddressBook, интерфейс пользователя которого вместе со спецификациями его виджетов изображен на рисунке **??**.

Цель создания этого класса — создание приложения, которое позволит запоминать адреса людей, соотнося их с их фамилиями, и напоминать адрес по введенной пользователем фамилии. Приложение очень простое и его моделью предметной области будет служить словарь, ключи которого фамилии людей, представленные строкой, а значения — адреса, также представленные строкой.

Используя палитру и инструменты, создадим приведенное на рисунке 7.1 окно и инсталлируем его в класс AddressBook (команда Install), выбирая в качестве подкласса класс ApplicationModel. Когда в окне нет выбранных виджетов, командой Define в окне инструментов холста, определим две аспектные переменные приложения для поля ввода и текстового редактора (name, address), методы доступа к ним (для этого должен быть выбран флажок Add Initialization) и методы действия для кнопок (doInput, doQuery, doExit). Теперь можно определять специфические функциональные возможности класса AddressBook.

7.1 Определение класса AddressBook

Определение класса AddressBook, которое было сгенерировано инструментом, отобразиться в браузере системы, после выполнения в окне инструментов холста команды Browse. Добавим в определение переменную экземпляра, определяющую функциональность самого класса (а не только переменных экземпляра, нужных для интерфейса пользователя). В таком простом приложении эта переменная будет хранить модель предметной области приложения. Назовём эту переменную addressBook. Определение класса должно быть таким[1]

[1]Смысл аргументов ключевых слов этого определения описан в [13, раздел 6.6].

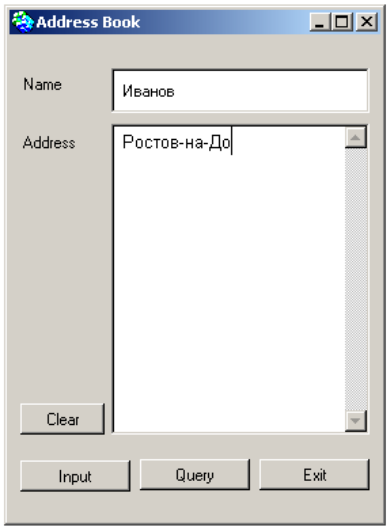


Рис. 7.1: Интерфейс класса AddressBook

```
Smalltalk defineClass: #AddressBook
    superclass: #UI.ApplicationModel
    indexedType: #none
    private: false
    instanceVariableNames: 'address name addressBook '
    classInstanceVariableNames: ''
    imports: ''
    category: 'UIApplications-New'
```

После внесения изменений определение класса нужно сохранить командой Accept. В классе уже есть два протокола методов, aspects — для методов доступа к аспектным переменным, и actions — для методов действия, которые будут выполняться при нажатии кнопок.

7.2 Протокол аспектных методов и ValueHolder

Посмотрим на методы, определённые инструментом в протоколе аспектов. Они имеют имена address и name. Это методы ленивой инициализации аспектных переменных, которые очень похожи и различаются только именем переменной, для которой они предназначены. Поэтому ограничимся изучением одного из них.

```
address
    ^address isNil
        ifTrue: [address := String new asValue]
        ifFalse: [address]
```

Ранее, при изучении меню, отмечалось, что сообщение asValue создаёт

хранитель значения или, по-другому, модель значения (в данном случае для виджета TextEditor — РедакторТекста). Познакомимся с хранителями значения подробнее.

Простая модель значения виджета

Модели значений — мощный механизм установки такой зависимости между моделью предметной области и виджетом, чтобы об изменениях, произошедшие с одним объектом, автоматически сообщалось другому. Этот механизм — фундаментальная особенность среды приложений VisualWorks.

Операция Define в UI Painter создает простую модель значения для каждого виджета, обычно это экземпляр класса ValueHolder. Такого объекта вполне достаточно для небольших приложений, у которых модель предметной области создается внутри модели приложения, как, например, в приложении AddressBook.

Любой виджет, который представляет данные (например, поле ввода, редактор текста) и позволяет управлять ими (такие виджеты называют виджетами данных), использует дополнительный объект, называемый моделью значения. То есть вместо того, чтобы хранить данные самому, виджет делегирует решение этой задачи модели значения. Таким образом, когда виджет данных принимает ввод от пользователя, он посылает эти данные модели значения для хранения, а когда он должен изменить свое визуальное представление, он спрашивает модель значении о данных, которые нужно отобразить.

Модель значения обеспечивает простые сообщения для доступа к хранимым ею данным. Все виджеты данных, чтобы сохранять и обновлять данные, пользуются стандартным протоколом. Они используют сообщение value, чтобы получить данные из модели значения, и сообщение value:, чтобы установить новые данные в модель значения и изменить представление виджета на экране. Другие объекты, например сама модель приложения, тоже могут посылать эти сообщения модели значения, чтобы программным образом получать или изменять данные.

Виджет данных всегда является иждивенцем своей модели значения в том смысле, что виджету модель значения всегда сообщает, что хранимые ею данные изменились. Виджет отвечает на такое уведомление, запрашивая модель значения о новых данных и отображая их, сохраняя визуальное представление виджета синхронизированным с реальными данными.

ValueHolder (ХранительЗначения) — наиболее часто используемый тип

модели значения. Как подразумевает его имя, хранитель значения хранит соответствующие данные. Хранитель значения более всего полезен для тех виджетов, которые принимают временные куски информации, хранимые интерфейсом до тех пор, пока они не будут обработаны.

Как и другие модели значения, объект ValueHolder позволяет более четкого отделить интерфейс пользователя от модели. Идея состоит в том, что модель должна представлять только структуры данных и связанные с ними методы. Однако в классической архитектуре MVC информация об интерфейсе часто попадает в модель. Например, модель должны посылать себе changed-сообщения, делая соответствующую информацию доступной представлению (тогда представление получает update-сообщение). Использование объектов типа ValueHolder — попытка удалить подобное из модели.

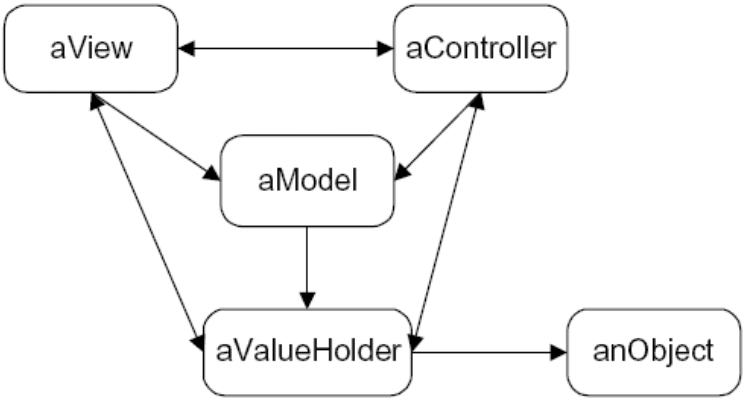


Рис. 7.2: Изменение архитектуры MVC в VisualWorks

Теперь модель обращается к объекту ValueHolder, который обращается к фактическим модельным объектам. В этом случае, модель не делает прямых ссылок к другие объекты. Когда модель хочет обратиться к значению, запрашивает о нём ValueHolder. Точно так же, если требуется изменить значение, она посылает сообщение об изменении ValueHolder, и последний уже изменяет значение.

Когда объект, хранимый объектом ValueHolder, изменяется, сообщить связанному с ним представлению, что значение изменилось и что представление должно отобразить новое значение — ответственность объекта ValueHolder (в классической архитектуре MVC это ответственность модели). Точно так же, если контроллер получает ввод от пользователя, который непосредственно изменяет значение в ValueHolder, контроллер просто сообщает хранителю значению, что его значение должно измениться. В этой схеме модель реально отделена от представления и контроллера, что, несмотря на техническое усложнение, делает задачу построения графического интерфейса более простой. Чтобы создать объект

ValueHolder, можно послать сообщение asValue тому объекту данных, который будет содержаться в создаваемом хранителе значения. В нашем примере экземпляр ValueHolder создает выражение String new asValue. Тот же результат можно получить, посылая классу ValueHolder сообщение newString. Каким методом пользоваться определяется только вкусами программиста.

Аналогично, посылка классу ValueHolder сообщения newBoolean эквивалентна выражению false asValue, посылка классу ValueHolder сообщения newFraction — выражению 0.0 asValue, посылка классу ValueHolder сообщения with: 230 — выражению 230 asValue.

Таким образом, в приложении AddressBook переменные address и name ссылаются на экземпляры класса ValueHolder, хранящие изначально пустые строку.

7.3 Протокол инициализации

Поскольку кроме аспектных переменных в классе AddressBook определена и переменная addressBook, ее нужно инициализировать. Лучшее место для этого — метод инициализации initialize. Для обращения к этой переменной создадим два методв доступа (в категории accessing), и метод определения значения переменной используем в методе инициализации.

```
addressBook
    ^addressBook.

addressBook: aDictionary
    addressBook := aDictionary.

initialize
    | adBook |
    super initialize.
    adBook := Dictionary new.
    self addressBook: adBook
```

7.4 Протокол действий

По умолчанию, методы действия определены инструментом так, что возвращают self. Поэтому их нужно переопределить. Сначала напишем метод doExit, как самый простой. Он просто будет закрывать окно приложения, используя наследуемый метод closeRequest.

```
doExit
    self closeRequest.
```

Метод doInput сложнее. Он будет получать имя и адрес из хранителей значений и сохранять их в модели предметной области — переменной addressBook, но не напрямую, а используя метод newAddress:for: из категории accessing класса AddressBook, который тоже нужно определить.

```
newAddress: anAddress for: aName
    | alreadyThere |
    alreadyThere := addressBook at: aName ifAbsent: [].
    alreadyThere isNil
            ifTrue: [addressBook at: aName put: anAddress]
            ifFalse: [Dialog warn: 'That key is already present']

doInput
    | aName anAddress |
    aName := name value.
    anAddress := address value.
    self newAddress: anAddress for: aName.
```

Метод doQuery будет возвращать в поле текстового редактора (изменяя его) адрес по имени, которое отображается в поле ввода. Для решения этой задачи потребуется еще один метод доступа addressFor:, который позволит получить адрес по имени (его тоже напишем). Если найденный адрес не nil, то поместим его в поле ввода/вывода адреса. Если адрес nil, отобразим диалоговое окно (см. главу **??**) с предупреждением об отсутствии в модели указанного имени.

```
addressFor: aName
    | anEntry |
    anEntry := addressBook at: aName ifAbsent: [nil].
    ^anEntry

doQuery
    | aName anAddress |
    aName := name value.
    anAddress := self addressFor: aName.
    anAddress isNil
        ifTrue: [Dialog warn:'There is no address for ', name value.]
        ifFalse: [address value: anAddress asText.]
```

Если всё сделано правильно, новое приложение должно запускаться при выполнении выражения AddressBook open в окне Workspace или при

нажатии кнопки Start в окне инструментов холста или в окне браузера ресурсов.

Немного поработав с окном AddressBook, приходишь к выводу о его несовершенстве. Полезно добавить еще одну кнопку, которая будет стирать записи в поле ввода и редакторе текстов. Для этого откроем окно браузера ресурсов, найдем в левой панели приложение AddressBook, в правой панели выделим ресурс windowSpec и нажмем кнопку Edit, открывая холст с его инструментами. Добавим в интерфейс еще одну кнопку действия, например, выше кнопки Input, определим ее метку (в поле ввода Label) как Clear, а имя метода (в поле ввода Action) как doClear. Не будем пользоваться командой Define (что можно было сделать, выделив новую кнопку), а только инсталлируем холст и в категории actions класса AddressBook напишем нужный метод.

```
doClear
    name value: String new.
    address value: String new.
```

Глава 8

Использование списков

Надо признать, что интерфейс AddressBook плохо спроектирован. Он не позволяет увидеть все ранее введенные имена, из них выбрать нужное имя и отобразить связанный с ним адрес. Чтобы построить приложение, решающее такую задачу, нужно привлечь в интерфейс более сложные объекты, а именно списки и виджет для списка.

8.1 Списки

Класс List объединяет свойства упорядоченных и сортируемых наборов и, кроме того, добавляет механизм зависимости. Но основное предназначение списка — выступать в качестве хранителя элементов для многоэлементных виджетов. Поэтому класс List находится в категории UIBasics-Collections.

В терминах реализации, класс List не входит в иерархию OrderedCollection, а является подклассом в ArrayedCollection. Его переменные экземпляра — collection, limit, collectionSize, dependents.

Переменная collection содержит элементы списка и метод создания экземпляра класса List определяет ее значение, как массив[1]. Это означает, что сам список не является набором, но содержит набор, как один из своих компонентов. Переменная limits содержит индекс последнего элемента в наборе, содержащего допустимый элемент. Переменная collectionSize содержит реальную вместимость набора, так что есть различие между размером и вместимостью списка.

Переменная dependents изначально имеет значение nil, но может указывать на объект или набор объектов, являющихся иждивенцами данного списка, и реализует механизм зависимости: когда в список добавляется или удаляется элемент, все иждивенцы уведомляются об этом.

Списки могут использоваться как сортируемые наборы, поскольку могут сортировать свои элементы, хотя и не имеют блока сортировки. По-

[1] Можно, при желании, заменить массив другим индексированным набором.

этому новые элементы добавляются в список, используя протокол adding, аналогичный такому же из класса OrderedCollection, например, addLast:, addFirst:. Чтобы затем отсортировать список, ему надо послать сообщение sortWith: aSortBlock или сообщение sort, которое использует блок сортировки по умолчанию [:a :b | a <= b].

Так как список не имеет блока сортировки и не может сортировать себя, в коде методов сортировки список просит произвести в нем сортировку класс SequenceableCollectionSorter:

```
sort
    ^SequenceableCollectionSorter sort: self

sortWith: aBlock
    ^SequenceableCollectionSorter sort: self using: aBlock
```

8.2 Виджет для списка

В интерфейсе пользователя список отображается через виджет List, который может использоваться в двух вариантах, позволяя производить выбор только одного элемента, или позволяя производить выбор нескольких элементов. Списки с выбором одного элемента используются намного чаще (например, в системных браузерах). Их цель подобна выпадающим меню, но меню имеют то преимущество, что не занимают место в окне. Списки с выбором нескольких элементов используются, например, в окне инструмента Definer из UIPainter.

Основные функции виджета списка хорошо известны: создать виджет и отобразить в нем список элементов, добавить новый элемент в список, удалить существующий элемент или элементы из списка, позволить выбрать пользователю элемент или элементы, определить выбранный элемент или элементы, определить индекс выбранного элемента или индексы всех выбранных элементов.

Виджет списка для выбора одного элемента

Моделью значения (аспектом) для виджета списка с выбором одного элемента (объектом, от которого зависит виджет списка) обычно является экземпляр класса SelectionInList. Экземпляр класса SelectionInList может быть создан сообщением new, как например, в выражении SelectionInList new, если нет отображаемого списка, или сообщением with:, как в выражении SelectionInList with: aSequenceableCollection, где аргумент содержит элементы, которые будут отображаться виджетом. В принципе

набор aSequenceableCollection может быть любым, но он должен поддерживать все те операции, которые должны будут выполняться с ним в приложении.

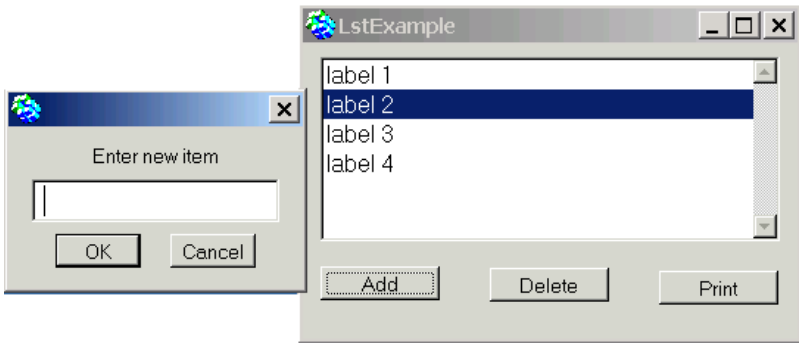


Рис. 8.1: Окно приложения ListExample

Класс SelectionInList определяет две переменные экземпляра с именами listHolder и selectionIndexHolder, которые содержат экземпляры класса ValueHolder. Переменная listHolder хранит список отображаемых виджетом элементов, а selectionIndexHolder — индекс выбранного в данное время элемента или 0, если выбранного элемента нет.

Чтобы получить выбранный элемент списка, надо послать объекту SelectionInList сообщение selection, которое возвратит выбранный в списке объект или nil. Чтобы получить индекс выбранного в данное время элемента списка, надо послать объекту SelectionInList сообщение selectionIndex, которое возвратит индекс выбранного объекта или 0. Чтобы программно изменить выбор, следует послать объекту SelectionInList сообщение selection: aValue или selectionIndex: anInteger. Чтобы гарантировать, что виджет списка будет посылать уведомление прикладной модели, когда произойдет изменение выбранного в нем элемента, нужно во время инициализации послать сообщение onChangeSend:to: (см. раздел 1.4 переменной selectionIndexHolder, как в следующем выражении

```
classList selectionIndexHolder onChangeSend: #changedClass to: self.
```

Получить набор всех элементов, отображенных в виджете, позволяет сообщение list. Чтобы изменить отображаемый виджетом список, и заставить иждивенцев отобразить изменение, нужно послать сообщение list: aSequenceableCollection с индексированным набором элементов в качестве аргумента. Например, выражение

```
listHolder list: #('line 1' 'line 2' 'line3')
```

сотрёт ранее отображаемые строки и заменит их на 'line 1' 'line 2' 'line3'. Рассмотрим на простом примере основное поведение виджета списка для

выбора одного элемента. Реализуем приложение с графическим интерфейсом, состоящим из окна с виджетом списка, отображающим три элемента 'label 1', 'label 2', 'label 3', и тремя кнопками с метками Add, Delete, Print. По нажатию кнопки Add приложение должно запросить у пользователя новый элемент и добавить его в список. По нажатию кнопки Delete приложение должно удалить выбранный элемент из списка. Нажатие кнопки Print позволит отобразить все элементы списка в окне Transcript. Кроме того, сделаем так, чтобы при каждом изменении выбора элемента из списка, в окне Transcript отображалось соответствующее уведомление, а элементы списка всегда сортировались по алфавиту.

Начнём с того, что установим в окно требуемые виджеты, и назовём модель значения (аспект списка — значение поля Aspect) — равным #labels. Методы действия для кнопок (значение поля Action) определим как #add, #delete, #print, соответственно. Так как окно должно открываться с тремя элементами в списке, мы должны инициализировать ими объект SelectionInList. Сделаем это в методе инициализации. Постановка задачи требует, чтобы, во-первых, в набор можно было добавлять элементы и удалять их, при этом сохраняя их порядок (поэтому определим используемый набор как экземпляр SortedCollection), и, во-вторых, каждое изменение выбора из списка должно посылать сообщение окну Transcript, поэтому зарегистрируем свой интерес в изменении выбора через сообщение onChangeSend:to: (см. раздел 1.4).

```
initialize
    labels := SelectionInList with:
                        #('label 1' 'label 2' 'label 3') asSortedCollection.
    labels selectionIndexHolder onChangeSend: #printChange to: self
```

Если сейчас открыть приложение, оно будет выглядеть так, как надо, но кнопки не будут работать, а любая попытка выбрать метку в списке откроет окно исключения, поскольку мы не определили метод printChange, указанный как сообщение, посылаемое при изменении выбора. Этот метод просто должен спрашивать список о текущем выборе в нём и посылает соответствующие сообщение вывода окну Transcript:

```
printChange
    labels selection isNil
        ifTrue: [Transcript cr; show: 'no selection']
        ifFalse: [Transcript cr; show: 'selection is ', labels selection]
```

Метод delete сначала должен проверить, есть ли выбранный элемент, и если есть, удалить его из списка. Чтобы сделать это, надо запросить у списка его текущий список элементов (сообщение labels listHolder value

или более короткое labels list), удалить из него выбранный элемент, и определить изменённый список, как новый отображаемый список, используя сообщение list:. Необходимо использовать именно сообщение list:, поскольку, если мы не сделаем этого, уведомление об изменении не будет передаваться виджету и связь между аспектом и списком будет разрушена.

```
delete
    labels selection isNil
        ifFalse: [| newList |
                Transcript cr; show: 'deleted ' , labels selection.
                newList := (labels listHolder value)
                                remove: labels selection; yourself.
                labels list: newList]
```

Обратите внимание на использование сообщения yourself непосредственно после того, как произошло удаление элемента. Оно возвращает изменённый список — приемник сообщения yourself. Это необходимо потому, что метод remove: изменяет приемник, но возвращает свой аргумент. Без yourself, значением newList стала бы строка, соответствующая удаленному элементу, и виджет обращался бы к ней, как к индексированному набору, и отображал литеры строки вместо отображения элементов списка. Потеря yourself — очень распространённая ошибка.

Метод add должен запрашивать новый элемент у пользователя. Он во многом подобен delete, но добавляет новый элемент в список, а не удаляет существующий из списка:

```
add
    | string |
    string := Dialog request: 'Enter new item' initialAnswer: (String new).
    string isEmpty
        ifFalse: [Transcript cr; show: 'added ', string.
                labels list: ((labels listHolder value) add: string; yourself)]
```

Наконец, метод печати всех элементов списка должен получать список от объекта SelectionInList и отображать его элементы один за другим в окне Transcript:

```
print
    Transcript cr.
    labels listHolder value do: [:item | Transcript show: item; cr]
```

Виджет списка для выбора нескольких элемента

Виджет списка для выбора нескольких элементов по существу тот же, что и виджет списка для выбора одного элемента. Они различаются только в следующем:

- Чтобы поместить в окно список для выбора нескольких элементов, надо разместить в окне тот же виджет List, но выбрать флажок Multiple Selection на странице Details его окна свойств.
- Модель значения (аспект) для виджета выбора из списка нескольких элементов — экземпляр класса MultiSelectionInList.
- Методы доступа к выбранным элементам имеют имена selections и selectionIndexes, selections: и selectionIndexes:, соответственно. Сообщение selections возвращает упорядоченный набор, содержащий все выбранные элементы списка, сообщение selectionIndexes — возвращает хранитель значения, содержащий множество индексов. Аргументами сообщений selections: и selectionIndexes: должны быть набор и множество, соответственно.

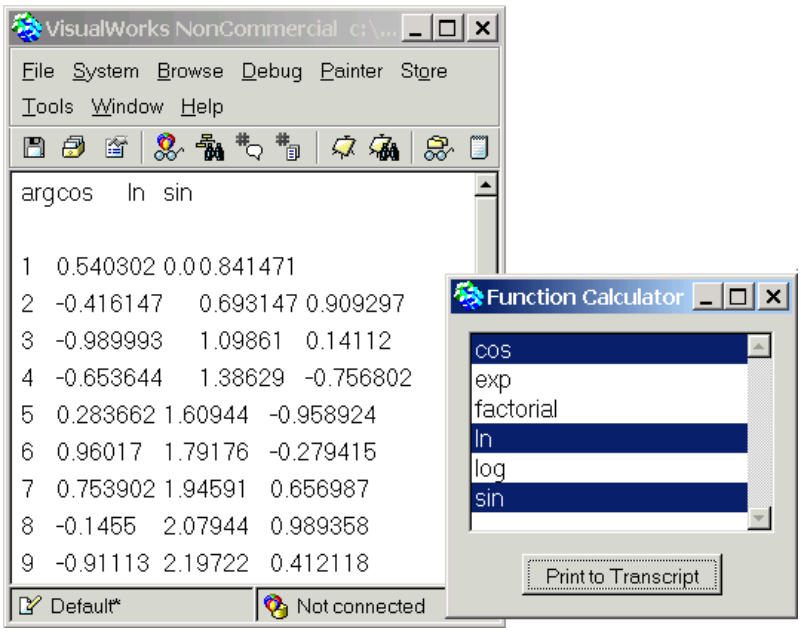


Рис. 8.2: Интерфейс приложения FunctionCalculator

Как пример напишем приложение FunctionCalculator, генерирующее и отображающее таблицу значений математических функций в окне Transcript, выбранных пользователем из списка в окне интерфейса пользователя приложения. Если по крайней мере одна функция выбрана, то щелчок на кнопке Print окна должен печатать значения всех выбранных

функций для аргументов от 1 до 10 с шагом 1, используя следующий формат:

x	cos	ln	log
1	0.540302	0.0	0.0
2	-0.416147	0.693147	0.30103
3	-0.989993	1.09861	0.477121
4	-0.653644	1.38629	0.60206
5	0.283662	1.60944	0.69897
6	0.96017	1.79176	0.778151
7	0.753902	1.94591	0.845098
8	-0.1455	2.07944	0.90309
9	-0.91113	2.19722	0.954242
10	-0.839071	2.30259	1.0

Когда выбранных функций нет, щелчок на кнопке Print должен открыть окно, с предупреждением об отсутствии выбора.

Для решения задачи достаточно создать в классе модели приложения всего два метода: метод инициализации, определяющий список функций (аспект списка будет называться у нас functions) и метод действия в ответ на нажатие кнопки Print — метод печати таблицы значений.

```
initialize
    functions := MultiSelectionInList with:
                    #( #cos #exp #factorial #ln #log #sin).

print
    functions selectionIndexes isEmpty
        ifTrue: [^ Dialog warn: 'You must select at least one function.'].
    Transcript clear; show: ' x '; tab.
    functions selections do: [:selection |
                        Transcript show: selection asString; tab].
    Transcript cr; cr.
    1 to: 10 do: [:arg |
            Transcript show: arg printString; tab.
            functions selections do: [:function |
                Transcript show:(arg perform: function) printString; tab].
            Transcript cr]
```

8.3 Пример с объектами ValueHolder и списком

Рассмотрим более сложный пример, содержащий более сложную модель предметной области, чем предыдущие примеры, и интерфейс, ис-

пользующий виджет списка. Приложение AddresBook1, как и его предшественник, приложение AddresBook, должно запоминать адреса и телефоны людей, соотнося их с именами, и напоминать адрес и телефон по имени, выбранному пользователем из списка уже введенных имен. Моделью предметной области этого приложения будет служить сортируемый набор экземпляров класса Person, содержащих всю нужную нам информацию о человеке, в котором его адрес, номер телефона и имя будут представляться строками. В качестве модели значений виджетов будем использовать уже знакомый нам ValueHolder.

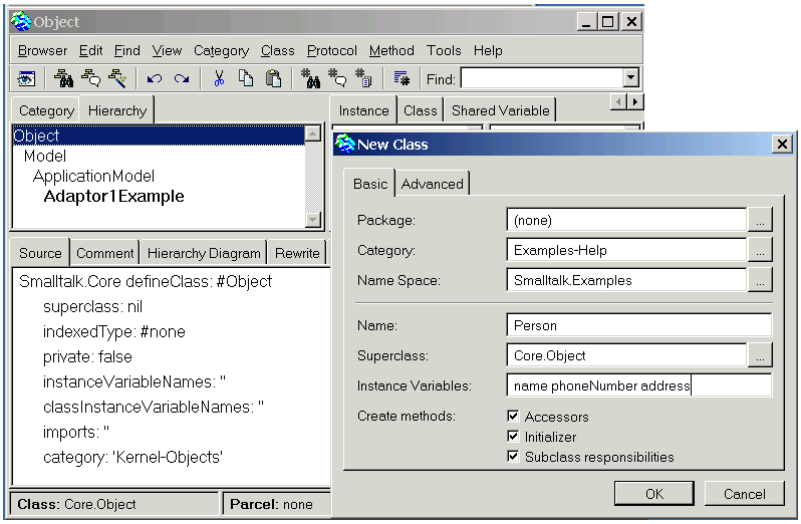


Рис. 8.3: Определение класса Person

Начнем с построения класса Person. Откроем браузер системы, выберем в иерархии классов класс Object и в всплывающем меню панели иерархии классов выберем команду New. Заполним поля окна New Class, например, так как на рисунке 8.3, определяя три переменных экземпляра: name, phoneNumber, address. Нажмем кнопку OK.

В методе initialize всем трем переменным присвоим значение String new, а в методах доступа, устанавливающих значения переменных и автоматически созданных системой, заменим параметр anObject на более содержательный aString. В протоколе displaying определим метод

```
displayString
    ^ '–', name
```

который нам потребуется в интерфейсе для отображения экземпляров класса Person.

Определим в классе Person еще протокол compareing, в который поместим методы сравнения двух экземпляров: они будут использовать соответствующие методы сравнения для строк, из которых удалены все

пробелы[2].

```
= aPerson
    ^(self name select: [:el | el ~= $ ]) =
                                (aPerson name select: [:el | el ~= $ ])

<= aPerson
    ^(self name select: [:el | el ~= $ ]) <=
                                (aPerson name select: [:el | el ~= $ ])
```

Чтобы не допускать в приложениях экземпляров класса Person с именами, сосоящими из одних пробелов, определим еще метод тестирования

```
isEmpty
    ^(self name select: [:el | el ~= $ ]) = ''
```

Прейдем к построению класса модели приложения. Используя палитру и инструменты, создадим указанное на рисунке 8.4 окно и инсталлируем его в класс AddressBook1 (команда Install), выбирая в качестве подкласса класс ApplicationModel. Когда в окне нет выбранных виджетов, командой Define в окне инструментов холста определим три аспектные переменные для полей ввода, аспектную переменную customers для виджета списка, методы доступа к ним (для этого должен быть выбран флажок Add Initialization) и методы действия для кнопок Add person, Clear и Exit. Методы доступа будут содержать код ленивой инициализации переменных, при этом переменные name, phoneNumber, address будут ссылаться на экземпляры класса ValueHolder, а переменная persons — на экземпляр класса SelectionInList. Теперь можно определять специфические возможности класса AddressBook1.

В протокол initialize-releaze методов экземпляра определим метод инициализации, в котором определим заинтересованность виджета списка в изменении выбираемого в нем элемента, для чего воспользуемся методом onChangeSend:to:.

```
initialize
    persons := SelectionInList with: SortedCollection new.
    persons selectionIndexHolder onChangeSend: #changedPerson to: self.
```

Создадим протокол методов экземпляра change messages и определим в нем сообщение, посылаемое приложению в ответ на изменение выбора элемента списка. Этот метод, посылая сообщения value: хранителям

[2]Разумеется, стоит написать метод condense, выполняющий эту работу, в классе String.

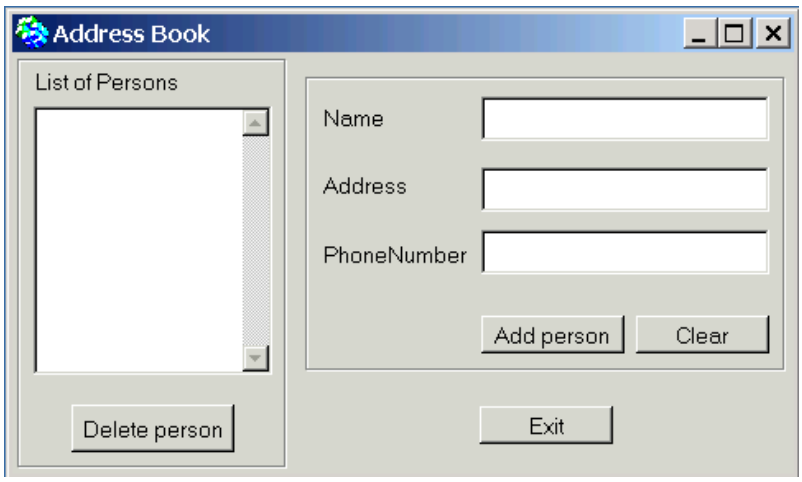


Рис. 8.4: Интерфейс приложения AddressBook1

значений, заставит связанные с ними виджеты изменить отображаемую ими информацию.

```
changedPerson
    | chosenPerson |
    chosenPerson := self persons selection.
    chosenPerson isNil
        ifTrue: [
            self name value: ''.
            self address value: ''.
            self phoneNumber value: '']
        ifFalse: [
            self name value: chosenPerson name.
            self address value: chosenPerson address.
            self phoneNumber value: chosenPerson phoneNumber].
```

Обратимся к методам действия кнопок интерфейса. Метод действия для кнопки Exit состоит из одного выражения: `^self closeRequest`. Метод действия для кнопки Clear должен очистить от информации все поля ввода и позаботиться о том, чтобы в виджете списка не было выбранных элементов.

```
clear
    name value: String new.
    address value: String new.
    phoneNumber value: String new.
    self persons selectionIndex: 0.
```

Метод delete должен удалить выбранный в списке элемент, изменить информацию в модели предметной области, заставить виджет списка ее правильно отобразить и очистить все поля ввода.

```
delete
    | chosenPerson list selectionInList |
    selectionInList := self persons.
    list := selectionInList list.
    chosenPerson := selectionInList selection.
    chosenPerson isNil
        ifFalse: [list remove: chosenPerson ifAbsent: [].
                selectionInList list: list.
                self clear.]
        ifTrue: [Dialog warn: 'No selection person'. ]
```

Последний метод add самый сложный. Чтобы избежать введения равных экземпляров класса Person (то есть с равными именами) сначала из списка удалим объект, равный добавляемому, если таковой есть, а потом добавим в список новый.

```
add
    | list selectionInList person |
    selectionInList := self persons.
    list := selectionInList list.

    "Добавить новое имя и соответствующую информацию в список."
    person := Person new.
    person name: self name value.
    person address: self address value.
    person phoneNumber: self phoneNumber value.
    person isEmpty
        ifFalse: [list remove: person ifAbsent: [].
                list add: person.
                selectionInList list: list.
                "Выбрать введенный объект."
                selectionInList selectionIndex: (list indexOf: person).]
```

Эта версия метода позволяет изменить адрес и телефон ранее введенного лица, только если после внесения изменений в соответствующих полях ввода нажать кнопку Add person. Для построения интерфейса, в котором при нажатии кнопки Add в список вводился новый экземпляр класса Person и последующее редактирование информации через интерфейс автоматически запоминалось в модели предметной области, воспользуемся более сложной моделью значения — объектом Aspect-Adaptor.

Глава 9

Адаптация виджета

9.1 Класс AspectAdaptor

Объект AspectAdaptor концептуально является указателем на удалённые данные и действует как посредник, не требуя при этом операции копирования. Практически это значит, что переменная экземпляра приложения должна ссылаться не на экземпляр ValueHolder, а на экземпляр класса AspectAdaptor, имеющим представление о теме (*subject*), которая соответствует модели предметной области, и об аспекте (*aspect*), которая является именем переменной, содержащей соответствующие данные.

Если воспользоваться предыдущим примером, можно создать экземпляр класса AspectAdaptor с экземпляром класса Person в качестве темы адаптера (сообщением subject: aPerson, посланным классу AspectAdaptor), и его переменной экземпляра phoneNumber, в качестве аспекта (сообщением forAspect: #phoneNumber, посланным созданному экземпляру адаптера). При этом, поле ввода номера телефона, регистрируется как иждивенец объекта AspectAdaptor, поэтому, когда пользователь изменит номер телефона, поле ввода сообщит объекту AspectAdaptor об изменении, и AspectAdaptor установит новый номер телефона в объект Person. Этот механизм основан на том, что адаптер аспекта обращает внимания на программные изменения данных только после получения сообщения value: и переводит получаемые им сообщения value и value:value в сообщения доступа (получения и модификации значения аспекта), понятные теме адаптера. По умолчанию AspectAdaptor посылает сообщение модификации update:with:from:.

Однако, модель предметной области при своем изменении не обязана посылает никаких сообщений, в том числе и адаптеру аспекта. Однако, когда создается экземпляр AspectAdaptor, можно определить, посылает ли тема сообщение модификации (из группы update) своему адаптеру. Если тема посылает такое сообщение модификации, AspectAdaptor регистрируется как иждивенец темы (сообщением subjectSendsUpdates: true), получает сообщения модификации, фильтрует их, и посылает соот-

ветствующие сообщения собственным иждивенцам (то есть, виджету). Если тема не посылает такого сообщения модификации адаптеру, AspectAdaptor не регистрируется как её иждивенец. Вместо этого, после установки нового значения в модели, он сам себе посылает сообщение модификации. Вот код, который делает это (класс AspectAdaptor наследует его из класса ProtocolAdaptor).

```
ProtocolAdaptor >> value: newValue
    self setValue: newValue.
    subjectSendsUpdates ifFalse: [self changed: #value]
```

Таким образом объект AspectAdaptor устанавливает связь между переменной экземпляра объекта предметной области и ее представлением на экране. Если редактируется, например, объект Person, с каждой переменной экземпляра объекта Person связывается другой экземпляр AspectAdaptor. Чтобы отредактировать другой экземпляр Person на том же экране, надо повторно подключить каждый объект AspectAdaptor к новому объекту Person.

Все это возможно выглядит достаточно сложно и, чтобы упростить ситуацию, VisualWorks для решения подобных задач позволяет использовать *тематический канал — subject channel*. При создании объекта AspectAdaptor, вместо определения темы определяется тематический канал сообщением subjectChannel: anObject, посылаемым классу AspectAdaptor. Объект anObject должен быть экземпляром класса ValueModel, например, экземпляром класса ValueHolder, который и определяет тематический канал. В нашем примере экземпляр класса ValueHolder будет хранить в качестве значения экземпляр Person.

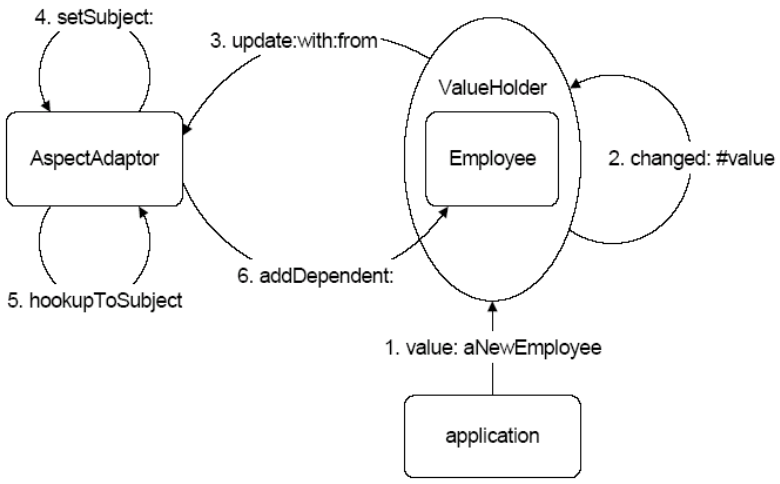


Рис. 9.1: Механизм работы тематического канала

В этом случае каждый экземпляр класса AspectAdaptor регистрирует-

ся как иждивенец объекта ValueHolder, так что, когда объект ValueHolder получает на хранение новый экземпляр Person (посылкой объекту ValueHolder сообщения value:), экземпляр AspectAdaptor информируется об изменении темы. Как только он получает такое уведомление, он изменяет свою тему, запрашивая ее у хранителя значения. Если несколько экземпляров AspectAdaptor зависят от объекта ValueHolder, то все они будут информироваться об изменение хранимого им значения, и установят его в качестве своей темы. (См. рисунок 9.1, при этом, обратите внимание, что шаг 6 будет выполняться только тогда, когда адаптер установлен так, что его тема посылает сообщение модификации.

По умолчанию, независимо от установки, адаптер аспекта предполагает, что модель предметной области не только имеет сообщения доступа для получения и установки значения переменной, но и эти сообщения соответствуют имени переменной (например, переменная name и методы доступа name и name:). Но как быть, если, например, аспект адаптера — переменная экземпляра с именем name, а методы доступа имеют имена getName и putName:? В этом случае после создания адаптера ему следует послать сообщение accessWith:assignWith:, первый аргумент — имя метода модели предметной области, который обращается к значению аспекта, второй аргумент — имя метода, который назначает аспекту новое значение.

Как пример использования объектов AspectAdapor, построим приложение AddressBook2, которое будет отличаться от AddressBook1 тем, что сначала определяется пустой экземпляр класса Person и записывается в список, а затем для него редактируется содержание полей ввода, которое автоматически записывается в объект Person из списка.

9.2 Пример приложения с адаптером аспекта

Построим приложение AddressBook2, почти с тем же окном интерфейса пользователя, что и приложение AddressBook1, но использующим адаптеры аспекта с тематическим каналом — хранителем значения, из которого каждый адаптер будет получать нужную ему информацию. Для этого определим новый класс (например, просто изменяя определение класса AddressBook1)

```
Smalltalk.Examples defineClass: #AddressBook2
    superclass: #UI.ApplicationModel
    indexedType: #none
    private: false
    instanceVariableNames: ”
```

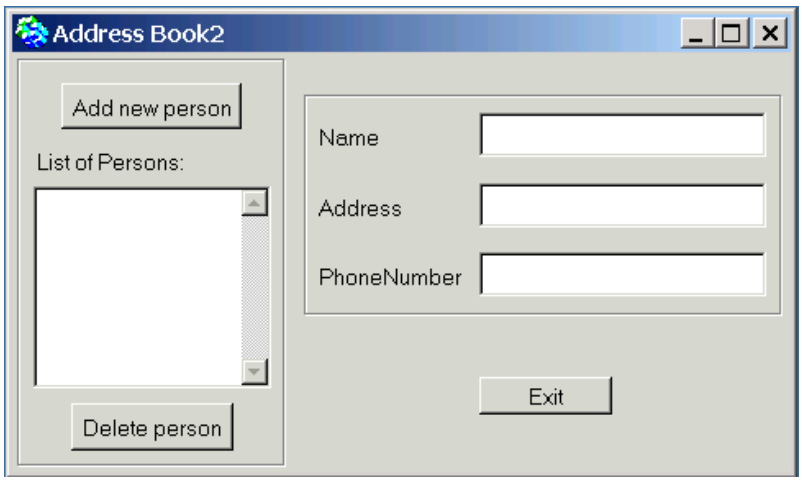


Рис. 9.2: Интерфейс приложения AddressBook2

```
classInstanceVariableNames: ''
imports: ''
category: 'Examples-Help'
```

Перенесем в него из класса AddressBook1 протокол методов класса interface spec и его метод windowSpec. В окне ResourceFinder нажмем кнопку Add..., на странице Find открывшегося блокнота найдем приложение AddressBook2, выберем его и нажмем кнопку OK, открывая на нем UIPainter. Внесем в него изменения, как на рисунке 9.2. На странице Details свойств полей ввода Name, Address, PhoneNumder выберем флажок Initially Disabled, делая эти поля изначально недоступными для ввода информации, определим аспектные методы и методы действия, пользуясь кнопкой Define, и откроем на классе AddressBook2 браузер классов.

В определении класса удалим переменные экземпляра address, phoneNumber, name и добавим переменную selectedPerson. В методе инициализации initialize модели приложения, определим не только модель значения для виджета списка, но и переменную экземпляра selectedPerson с хранителем значения, содержащим новый экземпляр класса Person.

```
initialize
    persons := SelectionInList with: SortedCollection new.
    persons selectionIndexHolder onChangeSend: #changedPerson to: self.
    selectedPerson := Person new asValue.
```

В протоколе методов экземпляра change messages определим сообщение, посылаемое приложению в ответ на изменение выбираемого из списка элемента. Этот метод, посылая сообщение value: хранителю значения, заставит связанные с ними виджеты изменить отображаемую ими информацию, а затем сделает все поля ввода доступными пользователю для ввода информации.

```
changedPerson
    | chosenPerson |
    chosenPerson := self persons selection.
    self selectedPerson value:
        (chosenPerson isNil
            ifFalse: [chosenPerson]
            ifTrue: [nil]).
    #(#name #address #phoneNumber)
            do: [ :componentName |
                ((self builder componentAt: componentName)
```

Переопределим аспектный метод **name**, в котором создадим экземпляр класса **AspectAdaptor** с тематическим каналом, посылая классу сообщение **subjectChannel:**. Аргумент этого сообщения — **selectedPerson** — хранитель значения, созданный в методе инициализации. После чего сообщим адаптеру аспекта, какой аспект модели предметной области он будет контролировать, посылая адаптеру сообщение **forAspect: #name**, и как он будет реагировать на изменение аспекта, посылая ему сообщение **onChangeSend: #redisplayList to: self**.

```
name
    | adaptor |
    adaptor := AspectAdaptor subjectChannel: self selectedPerson.
    adaptor forAspect: #name.
    adaptor onChangeSend: #redisplayList to: self.
    ^ adaptor
```

Чтобы этот метод корректно работал, нужно определить метод доступа к переменной экземпляра

```
selectedPerson
    ^ selectedPerson
```

и метод реагирования на изменение аспекта

```
redisplayList
    (self builder componentAt: #persons) widget invalidate.
```

который требует привести отображаемую виджетом информацию в соответствие с информацией из модели значения виджета, которая, возможно, была изменена. Переопределим и другие аспектные методы:

```
persons
    ^ persons
```

```
address
    | adaptor |
    adaptor := AspectAdaptor subjectChannel: self selectedPerson.
    adaptor forAspect: #address.
    ^ adaptor

phoneNumber
    | adaptor |
    adaptor := AspectAdaptor subjectChannel: self selectedPerson.
    adaptor forAspect: #phoneNumber.
    ^ adaptor
```

Обратимся к методам действия кнопок интерфейса. Метод действия для кнопки Exit состоит только из выражения ^ self closeRequest. Метод delete должен удалить выбранный в списке элемент, изменить информацию в модели предметной области, заставить виджет списка ее правильно отобразить и очистить от ранее введенной информации все поля ввода.

```
delete
    | chosenPerson list selectionInList |
    selectionInList := self persons.
    list := selectionInList list.
    chosenPerson := selectionInList selection.
    chosenPerson isNil
        ifFalse: [list remove: chosenPerson ifAbsent: [].
                selectionInList list: list.
                self selectedPerson value: Person new.]
        ifTrue: [Dialog warn: 'No selection person'.]
```

Последний метод add самый сложный. Чтобы избежать введения нового экземпляра класса Person с именем '-Unnamed' пока ранее введенный имеет еще это же имя (то есть оно еще не было изменено), сначала из списка удалим объект, равный добавляемому, если таковой есть, потом добавим в список новый элемент и приведем список в порядок, заново его сортируя, чтобы исправить нарушенный порядок элементов, вызванный изменением имени уже расположенного в списке элемента.

```
add
    | list selectionInList person |
    selectionInList := self persons.
    list := selectionInList list.
```

```
person := Person new.
person name: '-Unnamed'.
list remove: person ifAbsent: [].
list add: person; reSort.
selectionInList list: list.
selectionInList selectionIndex: (list indexOf: person).
```

Эта версия приложения позволяет изменить адрес и телефон ранее введенного лица простым изменением соответствующей информации в полях ввода. Однако, в отличие от версии приложения AddressBook1, приложение AddressBook2 не следит за введением двух элементов с одинаковыми именами, а просто заменит существующий элемент списка на новый.

Приложение работает. Чтобы, продемонстрировать возможности реагирования адаптеров на изменение его темы, проведем в приложение AddressBook2 небольшое изменение: добавим в него рядом с полем ввода номера телефона кнопку с именем Format и именем метода действия formatPN (см. рисунок 9.3), нажатие которой приведет к изменению строки представления номера телефона, добавляя в нее разделяющий символ '-'. На странице Details свойств кнопки Format выберем флажок Initially Disabled, делая кнопку изначально недоступной для пользователя. Инсталлируем измененный интерфейс и установим связь метода действия с именем formatPN с новой кнопкой, выбирая команду Define при выделенной кнопке Format.

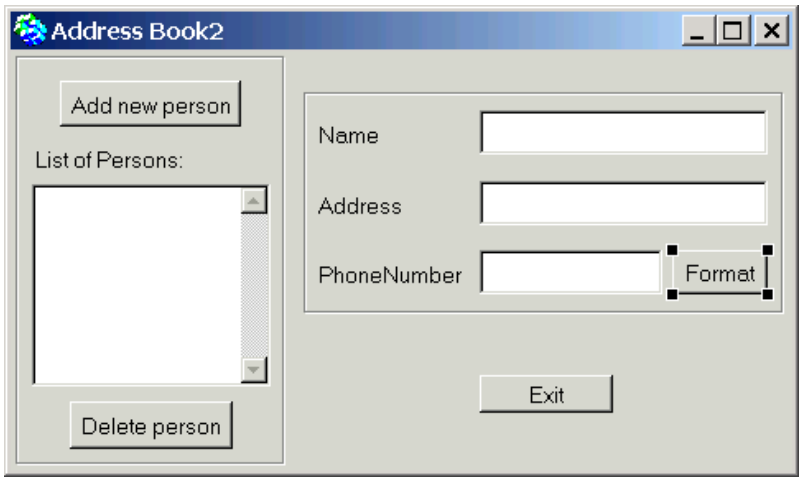


Рис. 9.3: Обновленный интерфейс приложения AddressBook2

В классе AddressBook2 изменим определение метода phoneNumber.

```
phoneNumber
    | adaptor |
    adaptor := AspectAdaptor subjectChannel: self selectedPerson.
```

```
    adaptor forAspect: #phoneNumber.
    adaptor subjectSendsUpdates: true.
    ^adaptor
```

Посылка адаптеру сообщения **subjectSendsUpdates: true** заставляет адаптер зарегистрировать себя в качестве объекта, зависимого от темы, то есть от объекта **Person**. Чтобы механизм зависимости корректно работал, в классе **Person** следует отредактировать каждый метод, непосредственно изменяющий значение данных так, чтобы он посылал сообщение **changed:** к **self** с именем того аспекта, который был изменен, в качестве аргумента (в этом случае, **#phoneNumber**). Это заставит приложение посылать сообщение зависимому адаптеру тогда, когда в модели предметной области произошли соответствующие изменения.

Затем определим новый метод действия

```
formatPN
    | chosenPerson |
    chosenPerson := self persons selection.
    chosenPerson formatPN.
```

Из кода метода понятно, что нужно определить метод **formatPN** в классе **Person**. Этот метод будет рассматривать телефонные номера не более чем из 11 цифр и представлять их в формате *-***-***-****.

```
formatPN
    | rawPhone rawSize basicCode prefix suffix1 suffix2 separator |
    rawPhone := self phoneNumber select: [ :ch | ch isDigit].
    rawSize := rawPhone size.
    basicCode := ''. prefix := ''. suffix1 := ''. suffix2 := ''. separator := '-'.
    rawSize >= 4 ifTrue: [basicCode :=
        (rawPhone copyFrom: (rawSize - 3) to: (rawSize -2)), separator,
          (rawPhone copyFrom: (rawSize - 1) to: rawSize)]
                ifFalse: [basicCode := rawPhone].
    rawSize >= 5 ifTrue: [prefix :=
              (rawPhone copyFrom: (1 max: (rawSize - 6))
                        to: (rawSize - 4)), separator].
    rawSize >= 8 ifTrue: [suffix1 :=
                  (rawPhone copyFrom: (1 max: (rawSize - 9))
                            to: (rawSize-7)), separator].
    rawSize > 10 ifTrue: [suffix2 := (rawPhone copyFrom: 1 to: 1),
                                separator].
    self phoneNumber: suffix2, suffix1, prefix, basicCode.
    self changed: #phoneNumber.
```

Глава 10

Прямое управление виджетами

Приложение во время выполнения часто должно управлять видом и поведением интерфейса. Для этого нужно иметь возможность его программно контролировать. Рассмотрим пути реализации такого контроля, методы идентификации и доступа к окнам и их виджетам. Знакомство с ними начнем с построения простого приложения, которое использует программный доступ к виджетам, опирающийся на свойство ID.

10.1 Игра в крестики–нолики

Реализуем игру Tic-Tac-Toe (крестики–нолики) с интерфейсом пользователя, представленным на рисунке 10.1.

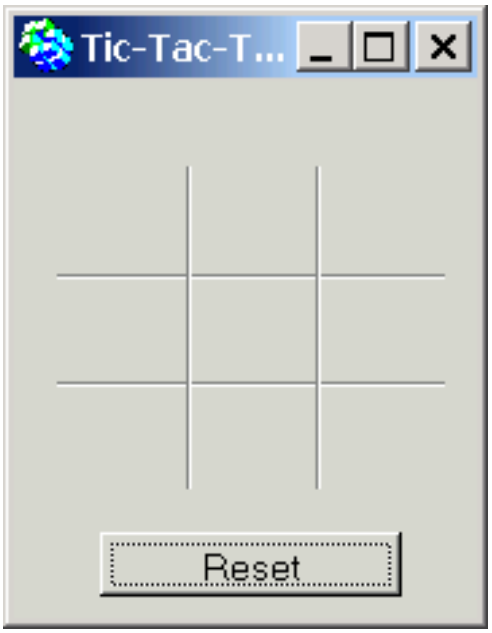


Рис. 10.1: Интерфейс пользователя игры Tic-Tac-Toe

В игре будут участвовать два игрока, обозначенных метками 'X' и 'O'; игрок, делающий первый ход, всегда 'X'. Когда окно открывается в первый раз, квадраты игрового поля пусты, и игроки начинают выбирать

квадрат игрового поля, щелкая на нем левой кнопкой мыши. Когда игрок щелкает на пустом квадрате, квадрат отображает символ игрока (’X’ или ’O’). Если эти действия приводят к появлению вертикальной, горизонтальной, или диагональной линии из трех одинаковых символов, то соответствующий игрок объявляется победителем; в противном случае игра продолжается. Если игрок щелкает по уже занятому квадрату, программа отображает предупреждение, призывая пользователя выбирать только пустые квадраты. Кнопка Reset очищает все квадраты игрового поля от введенных символов. Таким образом, в игре используются следующие сценарии:

Сценарий 1: Игрок щелкает на пустом квадрате.
Последовательность действий:

1. Пользователь щелкает на квадрате игрового поля.
2. Система отображает символ игрока.
3. Система проверяет, завершилась ли игра. Если это так, система отображает соответствующее уведомление и очищает игровое поле от введенных ранее символов; если это не так, она меняет текущего игрока — с ’X’ на ’O’ или наоборот.

Сценарий 2: Игрок щелкает на квадрате, уже содержащем символ.
Последовательность действий:

1. Пользователь щелкает на квадрате игрового поля.
2. Система отображает окно предупреждения, призывая пользователя использовать только пустые квадраты.

Сценарий 3: Игрок щелкает на кнопке *Reset*. *Последовательность действий:*

1. Пользователь щелкает на кнопке *Reset*.
2. Система очищает все квадраты.
3. Система устанавливает символ игрока равным ’X’.

Приступим к проектированию приложения. Объекты предметной области состоят из игроков и игрового поля. Единственная вещь, которую программа должна знать об игроках — кто является текущим игроком (’X’ или ’O’). Поэтому потребуется метка игрока, представленная объектом String. Таким образом, в приложении не нужны какие-либо классы предметной области. Потребуется только определить класс, содержащий

прикладную модель, и определяющий единственную переменную экземпляра player, содержащую строку, идентифицирующую текущего игрока. Класс модели приложения определим с именем TicTacToe как подкласс в ApplicationModel, который среди методов экземпляра будет содержать следующие категории:

- initialization, с методом инициализации переменной player значением 'X',
- actions, с методами, отвечающими на щелчок пользователя на квадрате игрового поля и кнопке *Reset*,
- private, с методами, проверяющими условие окончания игры, позволяющими переключаться между игроками после сделанного хода.

При раскраске холста реализуем квадраты игрового поля как командные кнопки (action button) с метками 'X', 'O' или пустой строкой. Кнопкам дадим имена в их полях ввода ID — от #button1 до #button9, считая квадраты слева направо и сверху вниз. Так как окно должно открываться с пустыми квадратами, поля *String*, связанные с метками кнопок, оставим пустыми (рис. 10.2). Можно разделить квадраты игрового поля вертикальными и горизонтальными разделителями (виджет Divider), а границы кнопок сделать невидимыми, для чего на странице Details панели свойств виджета отключить свойство Bordered. Еще одна командная кнопка — кнопка с меткой Resert и методом действия #resert.

Холст раскрашен, установим его в класс Tic-Tac-Toe, подкласс UI.ApplicationModel, командой Define определим методы действия (апектных методов не будет), командой Browse откроем на новом классе браузер системы, в определении класса Tic-Tac-Toe добавим переменную экземпляра player и девять переменных #button1, . . . ,#button9, затем сохраним класс. После этих операций определение класса Tic-Tac-Toe должно выглядеть примерно так:

```
Smalltalk defineClass: #TicTacToe
    superclass: #UI.ApplicationModel
    indexedType: #none
    private: false
    instanceVariableNames: 'button1 button2 button3 button4 button5
                button6 button7 button8 button9 player '
    classInstanceVariableNames: ''
    imports: ''
    category: '(none)'
```

В процессе игры нужно будет менять метки кнопок, когда игроки щелкают на них. Для этого необходим доступ к виджетам во время выполнения приложения.

10.1.1 Доступ к виджетам окна приложения

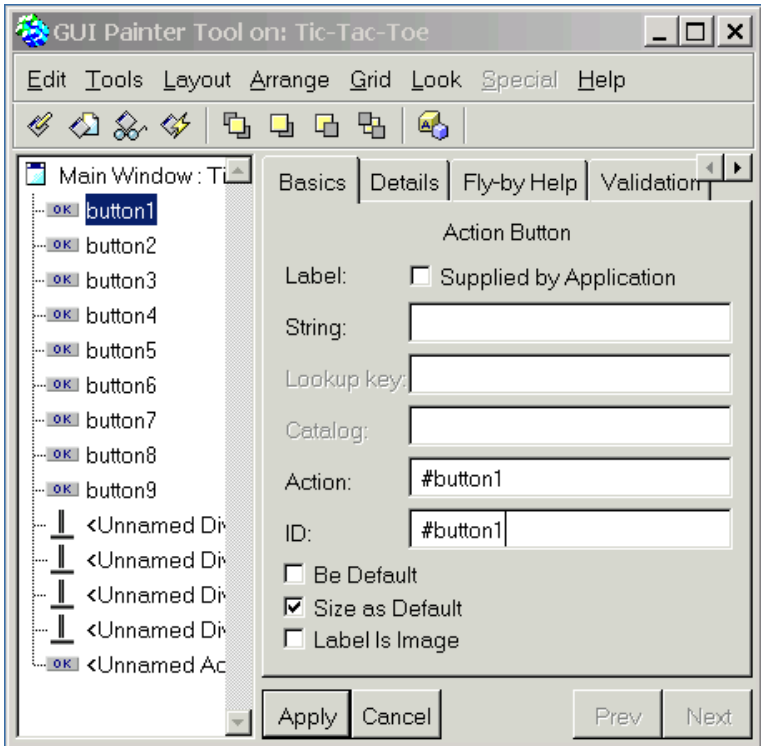


Рис. 10.2: Назначение идентификатора доступа к виджету

Доступ к виджетам осуществляется по именам виджетов, которые им назначаются свойством ID виджета в окне инструментов. Следовательно, чтобы сделать виджет доступным, следует заполнить поле ввода для его свойства ID в окне холста GUI Painter Tool (см. рис. 10.2).

После чего, для доступа к именованному виджету используются следующие методы (в более ранних версиях системы для этого требовалось обращение к составителю интерфейса)

wrapperAt: aSymbol — возвращает значение компонента с именем aSymbol, обычно это объект SpecWrapper или nil. В случае инструментальной панели, это фактический экземпляр ToolBar. Этот метод из класса ApplicationModel прямая замена выражения

```
self builder componentAt: aSymbol.
```

controllerAt: aSymbol — возвращает контроллер для компонента, связанного с aSymbol, обычно это объект Controller или nil. В случае инструментальной панели, это nil. Этот метод из класса ApplicationModel прямая замена выражения

```
(self builder componentAt: aSymbol) controller.
```

widgetAt: aSymbol — возвращает виджет, связанный с именем aSymbol, обычно это объект VisualPart или nil. Этот метод из класса ApplicationModel прямая замена выражения

```
(self builder componentAt: aSymbol) widget.
```

mainWindow — возвращает основное окно, связанное с экземпляром ApplicationModel, обычно это объект ApplicationWindow или nil, если окно для приложения не создавалось. Этот метод из класса ApplicationModel прямая замена выражения

```
self builder window.
```

windowMenuBar — возвращает экземпляр MenuBar, связанный с основным окном приложения. Может быть и nil, если окно не отображается и не открыто, или если в окне нет строки меню.

Теперь, когда есть возможность обращаться к виджету, как обратиться, например, к его метке? Очень просто: чтобы добраться до виджета, следует послать приложению сообщение widgetAt: aSymbol, затем виджету послать сообщение label, а чтобы получить текст метки, надо еще метке послать сообщение text. В целом, чтобы получить текст метки виджета, следует выполнить выражение

```
(anApplicftion widgetAt: aSymbol) label text.
```

Для изменения текста метки виджета следует выполнить выражение

```
(anApplicftion widgetAt: aSymbol) labelString: aString.
```

10.1.2 Реализация игры

Приступим к реализации приложения. Начнем с простого метода инициализации, в котором надо определить значение только переменной player.

```
initialize
    player := 'X'
```

Когда пользователь нажимает кнопку квадрата button1, её метод действия (из протокола actions) должен проверить, является ли метка кнопки пустой строкой и ответит на это следующим образом:

```
button1
    "Кнопка была нажата. Если кнопка имеет
    метку, вывести предупреждение."
        (self widgetAt: #button1) label text isEmpty
            ifFalse: [Dialog warn: 'This field is already occupied']
            ifTrue: [
                "Получить виджет, изменить метку. Выйти,
                если игра окончена, или заменить игрока."
                (self widgetAt: #button1) labelString: player.
                self endOfGame ifTrue: [^ Dialog warn: 'Game over.'].
                self newPlayer]
```

И таких методов надо 9! Повторы кода **(self widgetAt: #button1)** выглядят плохо. Кроме того, этот код потребуется снова в методе **reset** протокола **actions**. Поэтому можно вычислить такие выражения однажды во время инициализации, и сохранить результаты в переменных экземпляра, например, так

```
button1 := self widgetAt: #button1.
```

Следующий вопрос — где поместить присваивание значения переменной **button1** и переменным, соответствующим другим кнопкам. Ясно, что это должно произойти прежде, чем приложение откроется, в одном из методов, предназначенных для вмешательства программиста в процесс открытия интерфейса. Этого нельзя сделать в методе **initialize**, поскольку в этот момент компоновщика интерфейса еще не существует, этого нельзя сделать в методе **preBuildWith:**, поскольку компоновщик программы еще не обработал свойства виджетов и не знает их идентификаторов. Но это можно сделать в методе **postBuildWith:**, когда интерфейс в памяти уже построен, следующим образом:

```
postBuildWith: aBuilder
    "Сохранить виджеты кнопок для их по-
    следующего использования в методах дей-
    ствия."
    button1 := self widgetAt: #button1.
    button2 := self widgetAt: #button2.
    button3 := self widgetAt: #button3.
    button4 := self widgetAt: #button4.
    button5 := self widgetAt: #button5.
    button6 := self widgetAt: #button6.
    button7 := self widgetAt: #button7.
    button8 := self widgetAt: #button8.
    button9 := self widgetAt: #button9.
```

После этого, определение метода button1 примет вид

```
button1
    button1 label text isEmpty
        ifTrue: [button1 labelString: player.
                self endOfGame ifTrue: [^ self].
                self newPlayer]
        ifFalse: [Dialog warn: 'This field is already occupied']
```

В такой редакции, это определение нужно было бы повторить с незначительными изменениями для каждой кнопки. Кроме того, если надо будет сделать какие-либо изменения, их надо будет повторить девять раз. Лучшее решение состоит в том, чтобы выделить общий код в отдельный частный метод и с его помощью реализовать все методы button.

```
doButton: aButton
    "Кнопка aButton была нажата. Проверить
    использовалась ли кнопка раньше, если нет,
    поменять её метку. Проверить условие окон-
    чания игры и заменить игрока, если игра не
    окончена."
    aButton label text isEmpty
            ifTrue: [aButton labelString: player.
                    self endOfGame ifTrue: [^ self].
                    self newPlayer]
            ifFalse: [Dialog warn: 'This field is already occupied']
```

Тогда методы button будут очень простыми, похожими на метод

```
button1
    self doButton: button1.
```

Так как изменяется состояние объекта, связанного с переменными button1, . . . , button9, не нужно производить никаких изменений непосредственно с переменными — они всегда будут указывать на тот объект, свойства которого изменились методом doButton:.

Реализуем метод endOfGame, который определяет конец игры. Его реализация прямолинейна — надо просто проверить, заполнены ли копиями обозначения текущего игрока строки, столбцы, или диагонали. А чтобы проверить, равен ли текст метки кнопки тексту, определяющему игрока, сначала реализуем метод

```
isPlayer: button
    "Сравнить отображаемый кнопкой текст с
    текстом, определяющим игрока."
    ^ button label text = player asText
```

В этом коде важно не забыть преобразовать строку player, потому что метка виджета — экземпляр класса Text, который отличается от строки и не может сравниваться со строкой.

```
endOfGame
    "Проверить все строки, колонки и диаго-
    нали на условие окончания игры. Возвра-
    тить true или false."
    | end |
end :=
((self isPlayer: button1) & (self isPlayer: button2) & (self isPlayer: button3)) |
((self isPlayer: button4) & (self isPlayer: button5) & (self isPlayer: button6)) |
((self isPlayer: button7) & (self isPlayer: button8) & (self isPlayer: button9)) |
((self isPlayer: button1) & (self isPlayer: button4) & (self isPlayer: button7)) |
((self isPlayer: button2) & (self isPlayer: button5) & (self isPlayer: button8)) |
((self isPlayer: button3) & (self isPlayer: button6) & (self isPlayer: button9)) |
((self isPlayer: button1) & (self isPlayer: button5) & (self isPlayer: button9)) |
((self isPlayer: button3) & (self isPlayer: button5) & (self isPlayer: button7)).
end ifTrue: [self reset. Dialog warn: 'Player ', player, ' wins. Game over.'].
 ^ end
```

Здесь использованы логические операции (& — и, | — или), определенные в классе Boolean.

Метод newPlayer тоже очень прост и просто заменяет игрока:

```
newPlayer
    player = 'X'
        ifTrue: [player := 'O']
        ifFalse: [player := 'X']
```

Наконец, метод reset сбрасывает метки всех квадратов игрового поля и устанавливает значение переменной player равным 'X':

```
reset
    "Сбросить метки кнопок и снова инициали-
    зировать переменную player."
        button1 labelString: ''.
        button2 labelString: ''.
        button3 labelString: ''.
        button4 labelString: ''.
        button5 labelString: ''.
        button6 labelString: ''.
        button7 labelString: ''.
        button8 labelString: ''.
```

```
button9 labelString: ''.
player := 'X'
```

10.2 Текст в виджетах

Как уже отмечалось в метках виджетов используются экземпляры класса Text, которые существенно отличаются от экземпляров класса String и Symbol. И строка, и текст, и символ — наборы литералов. Строка — последовательность литералов. Строки не содержат информации о собственном представлении — об используем шрифте, его размере, и так далее. Когда отображаются строки, они используют систему представления по умолчанию. Символ — уникальная последовательность литералов, её правила формирования немного другие, а основное предназначение — именование объектов, гарантирующее очень быстрый доступ к ним в силу свойства уникальности. Текст — составной объект, содержащий строки и информацию относительно её представления — о шрифте, о размере шрифта, о цвете, и другую информацию.

Внутренняя природа строки и символа похожи — это наборы литералов. Текстовые объекты связаны с ними концептуально, но с другой внутренней структурой. Это отражено и в иерархии классов: классы String и Symbol находятся в той же ветви иерархии, класс Text — отдельно. Все три класса наследуют много полезных методов из абстрактного класса CharacterArray.

Предназначение текстовых объектов состоит в том, чтобы собирать всю информацию, необходимую для отображения литералов на отображающей поверхности, типа экрана компьютера или принтера. Конечно, можно отобразить строку без явного преобразования её в текст, поскольку она автоматически преобразовывает себя в текстовый объект при отображении. Но использование текста необходимо, если надо отобразить строку не с шрифтом или цветом по умолчанию, а, например, подчеркнутым, курсивом или полужирным шрифтом.

Текст, как экземпляр класса Text имеет две переменные экземпляра — string и runs. Переменная string содержит отображаемые литералы (строку), а runs — экземпляр класса RunArray, который содержит индексные диапазоны и информацию относительно выразительности литералов. Например, runs может определить, что первый литерал использует стиль по умолчания, а следующие 15 — подчеркнуты. Преобразование от стилей к фактическим шрифтам выполняется объектом TextAttributes, который указывает для каждого стиля те параметры исполнения, которые нужно использовать.

Основное сообщение, управляющее выразительностью текста, сообщение вида emphasizeFrom: start to: stop with: emphasis, где start и stop — индексы начала и конца подстроки, которая будет выделена, а emphasis — ассоциативный список, связанный с определенным цветом, символ выразительности или массив, содержащий символ выразительности и, возможно, информацию о цвете[1]. Вот простые примеры (сначала надо преобразовать строку в текст!):

```
'Bold text' asText emphasizeFrom: 1 to: 4 with: #bold
'Bold text' asText emphasizeFrom: 1 to: 4
                    with: #color -> ColorValue red
```

Важно отметить, что каждое сообщение, определяющее новую выразительность, переопределяет выразительность, определённую ранее. Если нужно определить несколько характеристик выразительности для литерала, следует определять их как массив одним из двух следующих способов:

```
| text |
text := 'Bold italicized text' asText.
text emphasizeFrom: 1 to: 15 with: #(#bold #italics).
text emphasizeFrom: 5 to: 15 with: #bold

| emphases text |
text := 'Bold italicized red text' asText.
emphases := Array with: #bold
                    with: #italics
                    with: #color -> ColorValue red.
text emphasizeFrom: 1 to: 19 with: emphases
```

Рассмотрим пример простого приложения, в котором проведем эксперимент с метками виджетов, которые используют текст. Спроектируем приложение с интерфейсом пользователя, состоящим из окна, в котором располагаются метка и кнопка с именем New label. Сценарий работы с интерфейсом прост:

Когда пользователь нажимает кнопку, последовательно появляется ряд диалоговых окон, запрашивающих новую строку, характеристику выразительности (используется диалоговое окно с одиночным выбором), индексы начала и конца подстроки, к которой должна быть применена выбранная выразительность. Диалоговое окно выбора выразительности повторно появляется до тех пор, пока пользователь не нажмёт в нем

[1]Стандартные символы выразительности: #bold, #italic, #large, #normal, #serif, #small, #strikeout, #underline.

кнопку Cancel. После чего программа отобразит вместо первоначального текста метки 'Label' текст с указанными характеристиками.

Это простое приложение требует только класса модели приложение, которое назовем TestText. После создания графического интерфейса пользователя, следует определить значение свойства ID виджета метки равным #label, чтобы к нему можно было обращаться и изменять во время выполнения, и определить для кнопки New label метод действия newLabel, который должен выполняться при ее нажатии пользователем, затем инсталлировать интерфейс пользователя в модель приложения и написать ее код.

```
newLabel
    "Получить текст и характеристики его выразительности. Отобразить метку в окне интерфейса."
    |text |
    text := (Dialog request: 'Enter new text' initialAnswer: '') asText.
    "Получить характеристики выразительности и применить их к ранее введенному тексту."
    [| emphasis noSelection |
        emphasis := self getEmphasis.
        (noSelection := emphasis = = #noChoice)
            ifFalse: [text := self emphasizeText: text with: emphasis].
        noSelection] whileFalse.
    "Послать новую метку составителю программы для её непосредственного отображения."
    (self widgetAt: #label) label: (Label with: text)

getEmphasis
    "Получить характеристику выразительности через диалоговое окно."
    ^ Dialog choose: 'Select emphasis for text of label '
            fromList: #('bold' 'italic' 'small')
            values: #(#bold #italic #small)
            lines: 3
            cancel: [#noChoice]

emphasizeText: text with: emphasis
    "Применить выбранную пользователем выразительность к выбираемой им подстроке введенного ранее текста."
```

```
| start end |
start := (Dialog request:
            'Start index (', text,')' initialAnswer: '1') asNumber.
end := (Dialog request: 'End index (', text,')'
             initialAnswer: text size printString) asNumber.
^text emphasizeFrom: start to: end with: emphasis.
```

Глава 11

Графика в VisualWorks

Группа классов, поддерживающих графические возможности VisualWorks одна из наиболее сложных. Ее детальное описание не входит в задачу этой книги. Но, чтобы использовать эти классы при создании обычных приложений и расширений основной библиотеки, нужно знать несколько основных идей.

Первое, что требуется при создании графики, в том числе и графического интерфейса пользователя, — возможность рисовать на экране объекты типа линий, виджетов, текста. Следует отметить, что отображение графических объектов на экране — сложная задача и различные языки, и даже различные диалекты Смолтока решают ее по-разному. VisualWorks для этого использует поверхность отображения (такую, как экран компьютера или бумага в принтере), графические объекты, которые надо будет нарисовать на поверхности отображения, и объект, который будет создавать рисунок и хранить параметры, описывающие контекст рисунка (шрифты, используемые цвета, характер выравнивания и так далее) — графический контекст. В этой главе рассматриваются основные концепции, заложенные в VisualWorks, которые иллюстрируются простыми примерами.

11.1 Поверхность отображения

Поверхность отображения — экземпляр одного из подклассов класса GraphicsMedium — носитель, на котором отображается всё визуальное: геометрические объекты, рисунки, виджеты, текст, ... Иерархия классов таких носителей очень большая, но её скелет можно представить следующим образом:

```
Object
    GraphicsMedium
        DisplaySurface
            UnmappableSurface
```

```
        Mask
        Pixmap
    Window
        ScheduledWindow
            ApplicationWindow
        TransientWindow
HostPrintJob
MockMedium
PostScriptFile
```

Абстрактный класс GraphicsMedium определяет протокол доступа по умолчанию к такой информации, как параметры отображения (цветовая палитра, шрифт, стиль окна просмотра, который содержит цвет фона и символов), размер самой поверхности отображения, число битов для представления цвета пикселов, графический контекст (объект, ответственный за отображение геометрических объектов, рисунков и текста).

Подклассы GraphicsMedium можно разделить на те, которые отображаются на экране, используются для создания поверхности отображения в памяти компьютера, предназначены для печати. Рассмотрим только первые два типа поверхностей. Группа классов, ответственных за отображение на экране и конструкцию отображения, определяется абстрактным классом DisplaySurface, экземпляры подклассов которого действуют как адресаты графических операций на растровом экране. Он определяет следующие переменные экземпляра:

handle — экземпляр класса GraphicsHandle или nil, который управляет используемыми ресурсами, определяет интерфейс с операционной системой, выполняющей функции отображения нижнего уровня.

width — число (SmallInteger), определяющее ширину поверхности в пикселах.

height — число (SmallInteger), определяющее высоту поверхности в пикселах.

background — экземпляр класса Paint, определяющий расцветку для свободной поверхности.

Подклассы DisplaySurface подразделяются на классы, экземпляры которых соответствуют окнам, отображаемым на экране, и классы, которые реализуют поверхности отображения в памяти, не отображая их.

11.1.1 Окна

Наверху иерархии окон располагается конкретный класс Window, имеющий большие функциональные возможности, но мало полезный сам по себе, поскольку его экземпляр не имеет контроллера (таким образом, возможно только минимальное взаимодействие с пользователем), не имеет ни одного компонента (таким образом, окно неспособно содержать виджеты). Однако, многое в поведении, наследуемом из класса Window весьма существенно. Например, знание о начале координат окна на экране, о метке, об иконке, и датчике, который содержит информацию о событиях мыши и клавиатуры, происходящих в пределах поля окна. Окно также определяет методы для открытия окна (они переопределяются в подклассах), его свертывания и развертывания, перемещения и изменения размеров, превращения окна на экране в активное окно. Класс также знает, как найти текущее активное окно (сообщение класса currentWindow).

Как пример программного управления окном, напишем код, демонстрирующий его некоторые функциональные возможности.

```
"Открыть окно с размерами, заданными пря-
моугольником, свернуть его, развернуть и
закрыть."
| window |
window := Window openNewIn: (100@100 corner: 200@200).
(Delay forSeconds: 3) wait.
window label: 'Test'.
(Delay forSeconds: 3) wait.
window collapse.
(Delay forSeconds: 3) wait.
window expand.
(Delay forSeconds: 3) wait.
window close
```

Обратим внимание на выражение window close. Если такого выражения нет, закрыть окно не удастся.

Класс Window имеет немного методов прямого использования. Более богатым на них является его подкласс ScheduledWindow (УправляемоеОкно). Этот класс добавляет несколько новых методов и переменных экземпляра, наиболее важными из которых являются переменные controller, обрабатывающая взаимодействие с пользователем, component, позволяющая включать визуальную часть или набор визуальных частей в окно, model, разрешающая другому объекту управлять окном.

Контроллер в экземпляре класса ScheduledWindow является экземпляром класса StandardSystemController, который обеспечивает оконное меню, методы для закрытия окна, изменения его размеров, перемещения, изменения метки, и другие операции с окном. Контроллер окна еще является и средством, с помощью которого Смолток следит за всеми окнами: контроллеры всех открытых смолтоковских окон хранятся в объекте ScheduledControllers, экземпляре класса ControlManager.

Пример, приводимый ниже, показывает как определить экземпляр ScheduledWindow, определить его цвет, метку, размеры на экране, продержать открытым три секунды и закрыть.

```
"Открыть окно с меткой, но без компонентов, и позволить пользователю закрыть его."
| window |
window := ScheduledWindow new.
window insideColor: ColorValue blue;
      label: 'Test'.
window openIn: (100@100 corner: 200@200).
(Delay forSeconds: 3) wait.
window close
```

Приложения, основанные на модели приложения и строящиеся инструментом UIPainter, используют класс ApplicationWindow — подкласс класса ScheduledWindow, из которого наследуется основное поведение окна приложения. Класс ScheduledWindow позволяет достаточно просто создавать окна программно, а не рисовать их, используя UIPainter. Из приложения можно получить экземпляр класса ApplicationWindow, посылая сообщение mainWindow модели приложения.

Класс ApplicationWindow добавляет функциональные возможности, требуемые инструментами построения интерфейса, и обеспечивает более плотную координацию с соответствующим объектом ApplicationModel. В частности ApplicationWindow может уведомлять своё приложение о событиях окна (например, закрытии или свертывании), так, чтобы другие, зависимые окна, могли следовать за ним должным образом. Распознаваемыми событиями являются такие события, как #expand, #collapse, #bounds, #enter, #exit, #close, #hibernate, #reopen, #release.

Класс ApplicationWindow среди прочих определяет следующие переменные экземпляра:

keyboardProcessor — содержит экземпляр класса KeyboardProcessor , который управляет вводом в приложение с клавиатуры.

application — обычно содержит само приложение, котороя является экземпляром класса ApplicationModel (или nil).

receiveWindowEvents — содержит экземпляр класса Array (или nil), определяющий те оконные события, на которые должен отвечать приемник.

damageRepairIsLazy — содержит экземпляр класса Boolean, который используется для того, чтобы определить, должно ли поврежденное ранее окно восстанавливаться (повторно прорисовываться) немедленно или может подождать более подходящего момента.

Типичными сообщениями, посылаемыми прикладной моделью приложению, являются

- изменение метки или цвета фона во время выполнения (сообщения label: и background:),
- перемещение окна в новую позицию (сообщение moveTo:),
- получение ограничивающего прямоугольника (displayBox),
- закрытие окна обычно достигаемое посылкой сообщения closeRequest прикладной модели),
- свертывание окна в иконку и расширение иконки в окно — сообщения collapse и expand,
- сокрытие окна без уничтожения внутреннего представления (сообщение unmap) и восстановление его без необходимости его реконструирования (сообщение map).

  В многих случаях, окно, изменённое во время выполнения, должно быть заново отображено сообщением display.

Когда прикладная модель должна обращаться более чем к одному окну, создаётся спецификация нового окна, которая устанавливается в ту же прикладную модель. Можно сделать новое окно иждивенцем главного окна (сообщением slave). Окна могут создаваться как партнеры при посылке окну сообщения bePartner.

11.2 Графический контекст

Когда нужно отобразить геометрический объект, например, прямоугольник, следует определить на какой части поверхности отображения его нужно нарисовать, какой цвет и какую ширину линий использовать. Для отображения текста надо знать еще шрифт. VisualWorks хранит эту

информацию в объекте GraphicsContext (ГрафическийКонтекст), связанном с поверхностью отображения, который выполняет роль, схожую с ролью пера или кисти, но который знает всё о графической среде и отвечает за все аспекты операции отображения в ней. Место класса GraphicsContext в иерархии классов следующие

```
Object
    GraphicsContext
        HostPrinterGraphicsContext
        PostScriptGraphicsContext
        ScreenGraphicsContext
```

а в комментарии класса говорится

> Я отображаю графические объекты в среде отображения и поддерживаю состояния графических параметров, определяющих, как должны представляться графические объекты.

Этот класс определяет много переменых экземпляра. Вот описание только некоторых из них.

clipOriginX, clipOriginY, clipWidth, clipHeight — описывают прямоугольник отсечения — часть поверхности отображения, на которой GraphicsContext будет рисовать графический обхект. Этот прямоугольник действует как трафарет и весь рисунок, который не попадает в него, игнорируется.

capStyle — определяет стиль, используемый для завершения прямых линий (важен только при рисовании толстых линий).

offsetX, offsetY — определяют точку, на которую перемещаются координаты графического контекста по отношению к началу координат среды.

scaleX, scaleY — коэффициенты масштабирования по соответствующим осям.

paint — определяет расцветку по умолчанию, когда рисуемая визуальная часть не определяет расцветку. Концепция расцветки требует краткого комментария. Поверхности отображения могут окрашиваться (заполняться) или цветом или шаблоном. Различие примерно такое же, как между окрашиванием стены краской и оклеиванием стены обоями: краска заполняет всю область, закрашивая ее одним и тем же цветом, в то время как плитка повторяется по определенному алгоритму по области.

phaseX, phaseY — определяют отправную точку расположения первого шаблона при закрашивании.

paintPolicy, fontPolicy — если платформа не может обеспечить требуемый цвет или шрифт, этот объект находит подходящую доступную замену.

lineWidth — содержит число (SmallInteger), которое определяет ширину в пикселах рисуемой линии.

font — содержит объект ImplementationFont, определяющий шрифт для отображаемого текста.

Большинство методов отображения определено именно в этом классе, а подклассы реализуют существующие различия, например, между экранном и печатающим устройством.

Приведем примеры, показывающие, как использовать объект GraphicsContext в разных ситуациях, как управлять отображением через графический контекст, как использовать шаблоны. Обратите внимание, что код примеров не обеспечивает автоматического ремонта повреждений. Это означает, что если изменились размеры окна, окно было свернуто или частично перекрыто другим окном, а затем стало активным, рисунок будет потерян. Автоматический ремонт повреждений — отдельный механизм, который будет рассматриваться позже.

11.2.1 Пример отображения геометрических объектов

Реализуем приложение с простым интерфейсом пользователя, представленным на рисунке 11.1. Щелчок на кнопке Color будет открывать диалоговое окно для выбора доступного цвета. Щелчок на кнопке Draw будет открывать последовательность диалоговых окон, требующих ввода информации, необходимой для отображения объекта, выбранного радио-кнопкой. При открытии интерфейса сделаем выбранной радио-кнопкой кнопку Line. Определяемый методом инициализации прямоугольник отсечения внутри окна будет ограничивать пространство отображения его верхней частью.

Решение поставленной задачи сводится к работе с графическим контекстом, который управляет параметрами отображения и создаёт рисунок. Графический контекст можно получить из окна приложения через выражение

```
self mainWindow graphicsContext
```

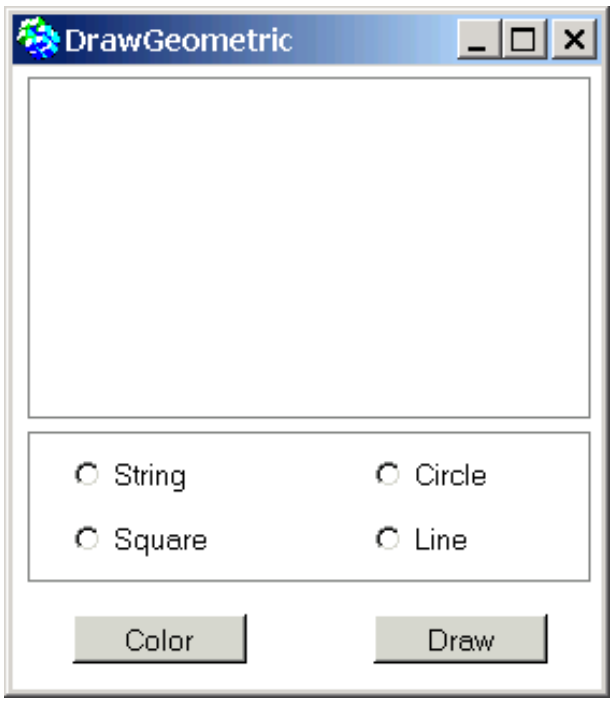


Рис. 11.1: Приложение для отображения геометрических объектов

Хотя это выражение наводит на мысль, что окно хранит графический контекст, это не так: окно порождает графический контекст с параметрами по умолчания всякий раз, когда его об этом просят. Таким образом, изменения, сделанные в графическом контексте не запоминаются, и когда графический контекст потребуется снова, он будет снова графическим контекстом с параметрами по умолчанию.

Чтобы реализовать приложение, нарисуем приведенный выше интерфейс пользователя, в котором кнопки действия выполняют методы действия #color и #draw, соответственно, а все радио-кнопки образуют связанную группу, из которой можно выбрать только одну, с единым аспектом object. Кроме того, для каждой радио-кнопки определим значение поля Select как символ с тем же именем, которое указали в поле String. На странице Color окна выберем для его фона цвет по умолчанию — white. На странице Position окна определим его фиксированные размеры, например, 300 по оси OX (width) и 330 по оси OY (height). Установим интерфейс в класс DrawObjects, и определим аспектный метод object и методы действия #color, #draw, выполняя команду Define. Откроем браузер на созданном классе и определим в нем дополнительные переменные экземпляра color, clippingRectangle, width. Таким образом, определение класса DrawObjects должно выглядеть примерно так

```
Smalltalk.Examples defineClass: #DrawObjects
    superclass: #UI.ApplicationModel
    indexedType: #none
```

```
private: false
instanceVariableNames: 'object clippingRectangle color width '
classInstanceVariableNames: ''
imports: ''
category: 'Examples-Help'
```

Метод инициализации определим в категории initialize-releaze следующим образом

```
initialize
    clippingRectangle :=
        10 @ 10 corner: 280 @ 200.          "Прямоугольник отсечения."
    color := ColorValue black.              "Цвет по умолчанию."
    object := #Line asValue.                "Выбранная радио-кнопка."
    width := 2                              "Ширина рисуемых линий."
```

Переопределим методы действия, созданные по умолчанию инструментом UIPainter. Метод color будет получать символ цвета из диалогового окна, а чтобы получить сам цвет, обратиться к классу ColorValue.

```
color
    "Получить цвет, затребованный пользовате-
    лем, и сохранить его в переменной color."
    | colorName |
    colorName := Dialog choose: 'Choose color'
                    fromList: ColorValue constantNames
                    values: ColorValue constantNames
                    lines: 8
                    cancel: [nil].
    colorName isNil ifFalse: [color := ColorValue perform: colorName]
```

Метод действия draw будет получать графический контекст окна, изменять его расцветку, ширину линий и прямоугольник отсечения в соответствии со значениями переменных экземпляра, и посылать сообщение, рисующее выбранный объект:

```
draw
    | gc |
    gc := self mainWindow graphicsContext.
    gc paint: color; lineWidth: width; clippingRectangle: clippingRectangle.
    self perform: 'draw', object value, 'On:' asSymbol with: gc
```

Последнее выражение возможно, поскольку радио-кнопки были определены так, чтобы они соответствовали требуемому сообщению отображения. Например, для кнопки Circle, последняя строка метода будет

эквивалентна self drawCircleOn: gc и попросит графический контекст нарисовать объект. Вот определение метода, рисующего квадрат:

```
drawSquareOn: aGraphicsContext
    "Нарисовать заданный по умолчанию квад-
    рат."
    ^aGraphicsContext displayRectangularBorder:
                        (30 @ 30 extent: 230 @ 160)
```

Стоит отметить, что вместо просьбы к графическому контексту нарисовать визуальный объект, можно попросить объект нарисовать себя, используя заданный графический контекст. Например, выполнение каждого из выражений

```
aGraphicsContext displayRectangularBorder: aRectangle
aRectangle displayOn: aGraphicsContext
```

приведут к одним и тем же последствиям. Преимущество второго способа состоит в том, что его можно использовать даже тогда, когда не известно, какой объект предстоит нарисовать, и потому его можно использовать при перечислении рисуемых объектов:

```
objects do: [:object| object displayOn: aGraphicsContext]
```

Остальные три метода создания рисунка строятся аналогично

```
drawStringOn: aGraphicsContext
    ^aGraphicsContext displayString: 'Hello, world!' at: 115 @ 50

drawLineOn: aGraphicsContext
    ^aGraphicsContext displayLineFrom: 30 @ 30 to: 260 @ 190

drawCircleOn: aGraphicsContext
    ^aGraphicsContext displayDotOfDiameter: 80 at: 150 @ 110
```

В заключение отметим, что в аспектном методе object, код ленивой инициализации переменной можно заменить на код, просто возвращающий значение переменной, поскольку ее начальная инициализация происходит в методе initialize.

11.3 Использование MVC при отображении графики

Хотя графический контекст, поверхность отображения, отображаемые графические объекты обеспечивают множество возможностей, они не

могут помочь при решении трех таких важных задач построения интерфейса пользователя, как: (1) управление графикой со стороны пользователя, (2) ремонт повреждений окна с графикой, (3) организация автоматической зависимости графического представления от данных предметной области. Сначала объясним, что подразумевается под этими задачами, а затем то, как VisualWorks решает их.

1. Говоря об управлении графикой со стороны пользователя, мы подразумеваем его способность взаимодействовать с графикой приложения, используя мышь и клавиатуру. Очевидно, что классы, созданные в предыдущих разделах, не позволяли пользователю взаимодействовать с нарисованными в интерфейсе объектами.

Поскольку в Смолтоке используется архитектура MVC, то согласно ей, каждый компонент UI, представляющий значение объекта предметной области, использует это значение как модель. Графическое представление модели — ответственность представления, и объекта, который управляет взаимодействием с пользователем в пределах границ представления, то есть контроллера представления.

Взаимодействие с пользователем через мышь и клавиатуру постоянно контролируется операционной системой. Когда происходит некоторое событие, операционная система посылает информацию о нем выполняющейся прикладной программе, в данном случае VisualWorks. После посылки некоторого сообщения, экземпляр конкретного подкласса Event посылает сообщение, соответствующее происшедшему событию, активному контроллеру, обычно контроллеру, ответственному за область экрана под курсором. Сообщение содержит информацию относительно вида происшедшего события и его параметры. Если контроллер заинтересован в событии, он должен иметь соответствующий метод, иначе «ничего не происходит» — так определено в наследуемом из класса Controller методе, который и будет выполнен.

2. Автоматический ремонт повреждений — еще одна особенность GUI, которая считается само собой разумеющейся: если окно повреждено, например, сворачиванием его в иконку с последующим возвращением в первоначальное состояние, или перекрыто другим окном, а затем переведено в активное состояние, естественно ожидать, что повреждение будет автоматически отремонтировано.

Основной механизм для автоматического ремонта можно получить, создавая класс представления как подкласс класса View. Когда окно или его часть повреждается, окно запоминает наименьший прямоугольник, содержащий повреждение. Когда потребуется, окно перерисует поврежденный прямоугольник или объединение всех повреждённых прямоугольников.

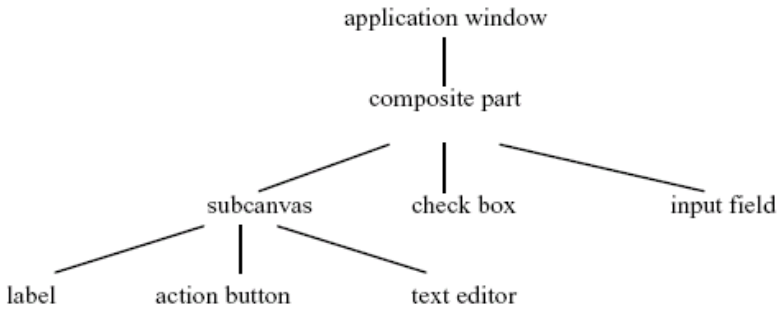


Рис. 11.2: Простейшая структура многокомпонентного окна

Если окно состоит не из одного виджета, оно содержит «составную часть», содержащую другие части (см. рисунок 11.2). Для ремонта повреждений посылается сообщение displayOn: graphicsContext вниз по дереву всем представлениям. Каждое представление должно отвечать на сообщение displayOn: (возможно наследуемое) и его реализация должна перерисовывать представление в пределах области отсечения графического контекста.

3. Естественно ожидать, что при изменении значения в модели предметной области, независимо от того, как оно произошло, изменение будет автоматически отражаться в соответствующем представлении. Например, если окно отображает круг и изменяется его диаметр, то интерфейс должен автоматически отобразить круг с новым диаметром.

Основной способ организации реагирования представления на изменение модели состоит в использовании механизма зависимости, для чего программист должен включить сообщение changed в каждый метод, изменяющий модель. В ответ на это, модель автоматически будет посылать сообщение модификации каждому из своих иждивенцев — в том числе и представлению. Представление, настроенное на метод модификации, запросит модель о необходимых данных и перерисует себя соответствующим образом. Чтобы перерисовать себя, представление обычно посылает себе сообщение invalidate, которое пересылается по дереву компонентов окну наверх, а затем окно таким же образом посылает сообщение display вниз по дереву компонентов.

Признание несоответствия представления модели и последующие действия, могут использовать два дополнительных механизма. Первый, состоит в том, чтобы определить, следует ли изменение представления производить немедленно или нет (последний механизм называется "ленивым ремонтом"). Второй механизм позволяет определить, подлежит ремонту целое представление или только его часть.

После общего описания, обратимся к примеру.

11.3.1 Интерфейс с хранителем представления

Инструмент UIPainter содержит в палитре стандартные виджеты, а нестандартные компоненты интерфейса можно легко создавать самим, пользуясь виджетом ViewHolder. Будем далее использовать термин «хранитель представления» для этого виджета, и термин «подпредставление» для визуального компонента, отображаемого в хранителе представления.

Процесс создания нового компонента, использующего виджет хранителя представления, прост. Он состоит из переноса виджета хранителя представления на холст, определения его свойств, создания подпредставления, которое он будет отображать, и определение его контроллера. Продемонстрируем эту процедуру на простом приложении, которое не позволяет пользователю взаимодействовать с представлением, и поэтому не требует пользовательского контроллера, а затем на примере, который требует проектирования и разработки пользователем контроллера.

Изменим реализацию приложения DrawObjects, сохранив интерфейс пользователя, представленный на рисунке **??**, и то же поведение: когда пользователь выбирает одну из радио-кнопок и щелкает на кнопке Draw, окно отображает соответствующую геометрическую форму. Для этого инсталлируем построенный ранее интерфейс в класс прикладной модели с именем GeometricFigures, но построим такое приложения, используя механизм зависимости и виджет ViewHolder. Перед обсуждением деталей реализации, опишем процесс создания хранителя представления:

1. Создать холст и разместить на нём хранитель представления — объект ViewHolder.
2. Определить свойства объекта ViewHolder, как минимум определить свойство View, которое является именем метода, возвращающего то подпредставление, которое будет отображаться в хранителе представления (в нашем случае это будет метод drawingView).
3. Инсталировать холст в класс модели приложения.

Начнем работу с того, что в область окна, ранее содержащую прямоугольник отсечения, поместим виджет ViewHolder, и в поле ввода View на его странице Basic окна инструментов введем имя метода #drawingView. Имя метода действия для кнопки Color изменим на setColor. Инсталируем холст и, воспользовавшись командой Define, определим аспектную переменную object и методы действия для кнопок. Откроем на классе модели приложения GeometricFigures браузер системы. Модель приложения должна хранить ссылку на представление в своей переменной экземпляра, поэтому введем в определение класса переменную drawingView. Введем переменную color, которая, как и раньше, будет хранит выбранный

пользователем цвет (по умолчанию — черный). Аспектная переменная object будет содержать текущий выбор радио-кнопки. Таким образом, определение нового класса будет выглядеть примерно так

```
Smalltalk.Examples defineClass: #GeometricFigures
    superclass: #UI.ApplicationModel
    indexedType: #none
    private: false
    instanceVariableNames: 'drawingView object color'
    classInstanceVariableNames: ''
    imports: ''
    category: 'Examples-Help'
```

Метод инициализации должен определить используемый цвет, создать экземпляр класса GeometricView, который еще предстоит создать, определить его в качестве значения переменной drawingView, и определить себя (экземпляр класса GeometricFigures) как модель. Класс GeometricFigures наследует механизм зависимости из суперкласса Model.

```
initialize
    "Выбрать при открытии радио-кнопку Line,
    установить цвет и определить экземпляр
    представления, создать связь с подпредстав-
    лением, определяя себя, как модель."
    color := ColorValue black.
    object := #line asValue.
    drawingView := GeometricView new.
    drawingView model: self
```

Метод color будет теперь возвращать значение переменной color. Метод действия setColor будет совпадать с методом color предыдущего примера, но в конце метода следует добавить еще одно выражение — self draw, которое позволит сразу же изменить цвет нарисованного объекта. Метод действия draw будет использовать механизм зависимости и посылать модели сообщение changed.

```
draw
    "Перерисовать подпредставление, вызывая
    механизм зависимости."
   self changed
```

Соответствующий метод модификации в классе GeometricView, подклассе класса View, будет только восстанавливать представление, а это поведение по умолчанию класса View, так что определять его не нужно. Метод update будет вызывать метод displayOn: класса GeometricView,

который спросит у модели о выбранном геометрическом объекте (полученном из значения аспекта выбранной радио-кнопки) и отобразит его.

При построении этого приложения не предполагается взаимодействие пользователя с представлением, поэтому будем использовать его с контроллером NoController.

Таким образом, следует создать новый класс и реализовать в нем указанные методы.

```
Smalltalk.UI defineClass: #GeometricView
    superclass: #UI.View
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: ''
    category: 'Examples-Help'

displayOn: aGraphicsContext
    "Получить выбранный в данное время геометрический объект и отобразить его в центре подпредставления."
    aGraphicsContext display: model currentObject at: self bounds center

defaultControllerClass
    ^ NoController
```

Вернемся к классу GeometricFigures. Метод display:at: из класса GraphicsContext может отображать любой отображаемый объект. В раннее приведённом приложении DrawObjects использовались сообщения типа displayLineFrom:to:, но такая стратегия в данном случае была бы не самой удачной. Поэтому для каждой радио-кнопки исправим значение свойства Select, так, чтобы первая буква была строчной (именно по этому в методе инициализации было написано #line asValue) и определим в классе GeometricFigures сообщение currentObject, которое использует выбранную радио кнопку, чтобы получить соответствующий геометрический объект.

```
currentObject
    ^ self perform: object value
```

Если выбрана кнопка, например, Line (аспект object равен #line), последняя строка примет вид self line. Чтобы завершить реализацию, надо определить методы, которые вычисляют и возвращаются соответствующий отображаемый геометрический объект. Очень важная особенность

их всех состоит в том, что метод display:at: не может отображать непосредственно сами геометрические объекты. Такие объекты должны сначала быть «обернуты» в соответствии с сообщениями asStroker (для пустых рисунков, типа линии) или asFiller (для заполняемых рисунков, типа закрашенного квадрата).

```
square
    "Возвратить отображаемый прямоугольник.
    Координаты задаются в предположении, что
    представление отображает объект в центре."
    ^(Rectangle origin: -60 @ -60 corner: 60 @ 60) asFiller

line
    ^(LineSegment from: -50 @ -50 to: 50 @ 50) asStroker

circle
    ^(Circle center: 0@0 radius: 50) asFiller

string
    ^'Hello, world'
```

Построенное приложение корректно восстанавливает отображение геометрических объектов в случае всевозможных повреждений его окна.

Глава 12

Использование контроллеров

12.1 Классы контроллеров

Контроллеры ответственны за управление вводом пользователя, и библиотека классов содержит много классов контроллеров для различных видов виджетов и инструментов, но только некоторые из них представляют интерес для большинства программистов. Существенная часть иерархии классов контроллеров такова:

Object ()
    Controller ('model' 'view' 'sensor')
        ControllerWithMenu
            ModalController
            ParagraphEditor
                TextEditorController
            SequenceController
        LauncherController
        MenuItemController
        NoController
        ScrollbarController
        StandardSystemController

Здесь контроллеры, обычно необходимые в приложениях, выделены полужирным шрифтом. Класс Controller определяет все существенные переменные экземпляра и поведение, и часто используется как прямой суперкласс создаваемых контроллеров. Переменные экземпляра model и view ответственны за связь с двумя другими компонентами триады MVC, а sensor — экземпляр конкретного подкласса абстрактного класса InputSensor, который обрабатывает ввод с мыши и с клавиатуры, и может обеспечить информацию относительно таких параметров, как координаты курсора.

Современная реализация контроллеров использует механизм управления событиями, когда VisualWorks немедленно уведомляется опера-

ционной системой о имевшем место событии ввода, и посылает эту информацию контроллеру в форме соответствующего предопределённого уведомляющего сообщения. Эти уведомляющие сообщения определены в протоколе событий (events) класса Controller и включает сообщения entryEvent: и exitEvent: (автоматически посылаемых контроллеру, когда курсор входит или оставляет область представления контроллера), doubleClickEvent:, redButtonPressedEvent:, keyPressedEvent:, и многие другие.

Все событийные сообщения имеют единственный аргумент — event, представляющий экземпляр соответствующего подкласса класса Event, типа CloseEvent, KeyPressedEvent, MouseMovedEvent. Объект Event содержит информацию, характерную для события, которая вызывает его, типа x-координаты и y-координаты текущей позиции горячей точки (острия) курсора или активизированной клавиши клавиатуры. Все предопределяемые сообщения — болванки, содержащие пустое тело, а конкретные подклассы переопределяют их, чтобы выполнить действия, необходимые тогда, когда реально происходит передача события.

Помимо класса Controller, наиболее важными его подклассами являются классы

StandardSystemController — отвечает за взаимодействие с окном, в частности с его всплывающим меню <window>.

ControllerWithMenu — имеет переменную menuHolder для хранения меню <operate>, и переменную performer, которая хранит объект, выполняющий команды меню.

TextEditorController — используется виджетом текстового редактора.

ParagraphEditor — суперкласс класса TextEditorController, определяющий большую часть его функциональных возможностей.

NoController — контроллер, который не отвечает на ввод со стороны пользователя.

Мы уже видели, что при создании интерфейса можно использовать виджет хранителя представления, определяя пару «контроллер – представление». Но в предыдущем разделе мы фактически не использовали контроллер, поскольку использовали экземпляр класса NoController, не позволяющий взаимодействовать с пользователем. Теперь создадим активный интерфейс пользователя и покажем, как создать активный контроллер, который отвечает на ввод пользователя.

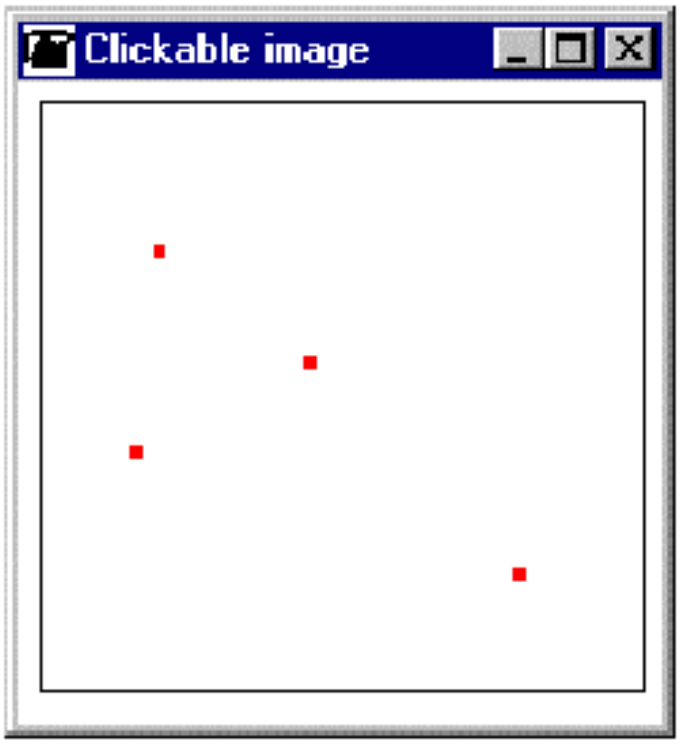


Рис. 12.1: Интерфейс программы отображения точек

12.2 Пример: приложение для отображения точек

Реализуем приложение (рисунок 12.1), которое отображает представление, снабженное меню операций и способное отвечать на щелчки мыши. Когда представление пусто, меню содержит только команду Add hot spot (Добавить горячую точку). Когда представление содержит по крайней мере одну горячую точку, меню отображает еще и команду Remove hot spot (Удалить горячую точку).

Когда пользователь выбирает команду Add hot spot, программа запрашивает строку (имя горячей точки) и курсор принимает форму перекрестия. (Курсор изменяет форму на первоначальную всякий раз, когда он оставляет область представления и вновь принимает форму перекрестия, когда снова возвращается на область представления.) Щелчок левой кнопки мыши внутри представление создает «горячую точку» и отображает её в виде маленького красного прямоугольника. Затем курсор изменяет свою форму на первоначальную.

Когда пользователь выбирает команду Remove hot spot, форма курсор принимает форму перекрестия и щелчок левой кнопкой мыши по горячей точке отображает предупреждение с именем горячей точки и удаляет горячую точку из представления. (В течение этой процедуры, форма курсор снова изменяется на первоначальную всякий раз, когда он покидает область представления.) После выполнения операции, курсор

снова принимает первоначальную форму.

Для решения поставленной задачи нужно уметь управлять курсором, то есть знать основную функциональность, обеспеченную библиотечным классом Cursor, и создать три новых класса:

(1) класс контроллера ClickableController,
(2) класс подпредставления ClickableView,
(3) класс модели приложения ClickableImage.

12.2.1 Курсоры

Курсор — визуальное представление текущей позиции мыши на экране. Курсор в VisualWorks поддерживается классом Cursor, который определяет более 20-ти предопределённых форм курсора (write, hand, . . .) и поведение курсора. В случае необходимости, пользователь может создавать новые формы курсора, используя, например, редактор образа Image Editor.

Класс Cursor определяет четыре переменных экземпляра: image (образ), mask (маску), hotSpot («горячая точка» курсора) и name (имя курсора). «Горячая точка» (hot spot) — возвращаемая точка, когда программа запрашивает позицию курсора и представляет позицию мыши на экране. Маска определяют непрозрачные пикселы курсора и прозрачные пикселы, что позволяет видеть фон позади курсора.

К классу Cursor обычно обращаются для того, чтобы отобразить специальный курсор во время специальной операции. Например, сама система VisualWorks отображает специальный курсор при чтении файла, при выполнении сборки мусора или при ожидании выбора пользователем точки на экране (для этого система использует курсор в виде перекрестия). Обычными сообщениями для изменения формы курсора являются showWhile: aBlock и show. Сообщение showWhile: aBlock изменяет курсор на время выполнения блока, и заменяет его на первоначальную форму, когда выполнение блока завершается. Типичный пример использования — следующий метод из класса SimpleBrowserModule:

```
hardcopyStream: aStream
    "Использовать курсор 'wait' во время печати
    потока aStream."
    Cursor wait showWhile: [aStream contents asComposedText hardcopy]
```

Метод show также изменяет форму курсора, но программист должен сам изменить его форму на первоначальную. Типичное использование

демонстрирует определение метода showWhile: в классе Cursor:

```
showWhile: aBlock
    "При выполнении блока aBlock, заставить
    приемник изменить форму курсора, затем
    возвратиться к первоначальной форме кур-
    сора."
    | oldcursor |
    oldcursor := self class currentCursor.
    ^self == oldcursor
            ifTrue: [aBlock value]
            ifFalse: [self show.
                    aBlock ensure: [oldcursor show]]
```

Сообщение currentCursor, посланное классу курсора, возвращает форму текущего курсора системы VisualWorks.

12.2.2 Класс ClickableController

Класс ClickableController будет подклассом в ControllerWithMenu, поскольку нужно меню операций. Он будет определять методы построения меню в зависимости от наличия «горячих точек», и поведение на события входа курсора в область представления и выхода из неё, а также реакцию на событие нажатия левой кнопки мыши.

Чтобы реализовать желаемое поведение курсора, класс ClickableController должен иметь переменную экземпляра, которая хранит форму курсора до её изменения на форму перекрестия, что позволяет восстановить его при выходе за границы представление или когда заканчиваются «горячие точки». Назовем эту переменную cursor. Так как эта переменная еще не определена перед первым действием пользователя, метод cursor будет обращаться к ней, используя метод ленивого доступа:

```
cursor
    ^cursor isNil
            ifTrue: [Cursor currentCursor]
            ifFalse: [cursor]
```

Сообщение enterEvent: посылается тогда, когда курсор входит в область подпредставления. Он должен спросить модель приложения, какая команда в настоящее время выполняется (это будет делать метод state), и отобразить курсор в форме перекрестия, если команда не nil:

```
enterEvent: event
```

```
    "Если модель выполняет команду Add или
    Remove, изменить форму курсора на пере-
    крестие."
    model state isNil
            ifFalse: [cursor := Cursor currentCursor.
                    Cursor crossHair show]
```

Так как этот метод определен в контроллере подпредставления, он будет выполняться только тогда, когда контроллер получает управление, а это случается только тогда, когда курсор находится в области подпредставления.

Метод **exitEvent:** переустанавливает форму курсора на предыдущую, если переменная **state** не **nil**.

```
exitEvent: event
    model state isNil
            ifFalse: [self cursor show]
```

Метод **state** из **ClickableImage** возвращает символы **#addHotSpotAt:**, **#removeHotSpotAt:** или **nil**, в зависимости от той команда, которая в настоящее время выполняется.

Обработчик события щелчка левой (красной) кнопкой мыши (снова будет выполняться только в том случае, когда курсор находится в пределах представления) должен попросить модель выполнить операцию, соответствующую текущему состоянию (добавить/ удалить горячую точку или ничего не делать), посылая координаты курсора в качестве аргумента:

```
redButtonPressedEvent: event
    “Послать сообщение с координатами курсо-
    ра модели.”
    |position|
    position := self sensor cursorPointFor: event.
        “Извлечь позицию из события.”
    model executeOperationWith: position
        “Результат зависит от состояния модели.”
```

Следующая задача контроллера — реализовать меню. Так как меню зависит от состояния приложение, предоставим решение этой задачи приложению, которое будет создать меню по мере необходимости. Таким образом, разработка класса контроллера завершена!

12.2.3 Класс ClickableView

Обратимся к классу ClickableView, который естественно сделать подклассом View. Из предыдущего следует, что в его обязанности входит доступ к классу контроллера ClickableController, реакция на сообщение update (через механизм зависимости), и на сообщение displayOn:.

Прежде всего определим, что его контроллер по умолчанию — экземпляр класса ClickableController:

```
defaultControllerClass
    ^ ClickableController
```

Метод displayOn: должен отобразить все «горячие точки», как красные небольшие квадраты, центры которых расположены в точках, соответствующих их координатам.

```
displayOn: aGraphicsContext
    "Отобразить все горячие точки, как красные
    квадраты, центрированные по их координа-
    там. "
    aGraphicsContext paint: ColorValue red.
    model hotSpots do: [:hotSpot|
        aGraphicsContext displayRectangle: ((hotSpot at: 1) - 2 extent: 4 @ 4)]
```

12.2.4 Класс ClickableImage

Класс ClickableImage, представляющий прикладную модель, последний класс, требующий определения. Начнем построение с того, что построим интерфейс пользователя в виде окна с именем Clickable image. Разместим в нем единственный виджет ViewHolder и определим его свойство View: равным imageView. Инсталлируем это окно в класс с именем ClickableImage, подкласс класса ApplicationModel.

Определим в нем следующие основные переменные экземпляра: imageView, для доступа к представлению, отображаемому в подпредставлении (хранитель представления, определяемый свойством View), hotSpots — набор, содержащий массивы с двумя элементами — координатой горячей точки и именем), state, для хранения одного из символов #removeHotSpotAt:, #addHotSpotAt: или nil. Полезно ввести две дополнительных переменные: hotSpotName, чтобы хранить имя новой горячей точки, и controller, чтобы обеспечить прямой доступ к контроллеру для управления меню, что удобнее обращения к контроллеру через переменную imageView. Если не создал системный браузер, следует создать get-методы доступа к переменным imageView, hotSpots, state.

Переменная state требует повышенного внимания. Как уже отмечалось, действия по добавлению и удалению горячей точки, вызываются через сообщение executeOperationWith: из метода redButtonPressedEvent: контроллера. Самый простой способ выполнить сообщение, которое зависит от символа, выполнить выражение perform: aSymbol. Так как эта операция требует координат курсора, сообщение должно быть ключевым. Чтобы корректно реагировать на разные возможные состояния переменной state (#addHotSpotAt:, #removeHotSpotAt: или nil), нужно будет определить методы, соответствующие ее двум первым значениям.

Начнем реализацию класса ClickableImage с метода инициализации, который определить начальные значения переменных экземпляра.

```
initialize
    hotSpots := OrderedCollection new.
    imageView := ClickableView new.
    imageView model: self.
    controller := imageView controller.
    controller performer: self.
    controller menuHolder value: self menuWithAdd
```

Следующие два метода создают два меню операций: (1) когда нет «горячих точек» в представлении, (2) когда есть по крайней мере одна «горячая точка».

```
menuWithAdd
    "Меню, когда нет горячих точек."
    | mb |
    mb := MenuBuilder new.
    mb add: 'add hot spot' -> #addHotSpot.
    ^mb menu

menuWithRemove
    "Меню, когда существует по крайней мере
    одна горячая точка."
    | mb |
    mb := MenuBuilder new.
    mb add: 'add hot spot' -> #addHotSpot.
    mb add: 'remove hot spot' -> #removeHotSpot.
    ^mb menu
```

Когда пользователь выполняет команды меню add hot spot или delete hot spot, контроллер посылает сообщения addHotSpot или removeHotSpot исполнителю меню — объекту ClickableImage. Поэтому нужно определить эти два методы. Метод addHotSpot запрашивает у пользователя имя

новой горячей точки и изменяет состояние на #addHotSpotAt:. Управление передаётся контроллеру, который отслеживает положение курсора (находится он или нет в области представления) и посылает сообщение executeOperationWith: объекту ClickableImage, когда пользователь нажимает левую (красную) кнопку мыши в представлении.

```
addHotSpot
    "Начало операции добавления горячей точки — получить имя и состояние; остальное оставить контроллеру."
    hotSpotName := Dialog request: 'Enter hot spot name.'.
    state := #addHotSpotAt:.
    controller enterEvent: #addHotSpot
```

Метод removeHotSpot только изменяет состояние:

```
removeHotSpot
    "Начало операции удаления горячей точки — изменить состояние; остальное оставить контроллеру."
    state := #removeHotSpotAt:.
    controller enterEvent: #remoneHotSpot
```

После выполнения любого из этих двух методов, приложение будет ждать от пользователя перемещения курсора и/или нажатия кнопки мыши, чтобы затем послать соответствующее событие мыши контроллеру. Это вызывает посылку сообщения redButtonPressedEvent:, определенного выше, которое пошлет сообщение executeOperationWith: aPoint. Как уже было сказано, этот метод просто выполнит метод perform: с символом из state и параметром aPoint:

```
executeOperationWith: aPoint
    state isNil
        ifFalse: [self perform: state with: aPoint.
                  self changed]
```

Реальная работа по добавлению и удалению «горячей точки» выполняется методами addHotSpotAt: и removeHotSpotAt: и механизмом зависимости, вызываемым сообщением changed).

```
addHotSpotAt: aPoint
    "Добавить горячую точку в набор, переустановить состояние и курсор, модифицировать меню."
    controller cursor show.
```

```
    hotSpots add: (Array with: aPoint with: hotSpotName).
    controller menuHolder value: self menuWithRemove.
    state := nil
```

Метод **removeHotSpotAt:** более сложен, поскольку должен проверить, была ли нажата кнопка мыши в пределах одного из красных квадратов:

```
removeHotSpotAt: aPoint
    "Проверить, что aPoint соответствует «горячей точке» и, если это не так, сообщить пользователю; отобразить имя, удалить горячую точку из набора, переустановить состояние и курсор, модифицировать меню в случае необходимости."
    | hotSpot |
    controller cursor show.
    (hotSpot := self hotSpotAt: aPoint) isNil
            ifTrue: [^ Dialog warn: 'Not a hot spot'].
    Dialog warn: 'Hot spot to be removed: ', (hotSpot at: 2).
    hotSpots remove: hotSpot.
    hotSpots isEmpty
            ifTrue: [controller menuHolder value: self menuWithAdd.
    state := nil
```

Осталось реализовать метод **hotSpotAt: aPoint**, который проверяет, соответствует ли **aPoint** некоторой «горячей точке» и возвращает горячую точку, соответствующую позиции, когда курсор находится в области красного квадрата, окружающего «горячую точку», а иначе возвращает **nil**:

```
hotSpotAt: aPoint
    "Находится ли aPoint в пределах 2-х пикселов от центра «горячей точки»? Если это так, возвратить »горячую точку»."
    ^ hotSpots detect: [:hotSpot |
            (aPoint - (hotSpot at: 1)) abs <= (2 @ 2)]
                ifNone: [nil]
```

Построение последнего класса завершено и можно проверить его функциональность!

Проектные задания к модулю III

1. Создать приложение, которое позволяет ввести иностранное слово и его перевод и, в последствии, посредством выбора одного из них, получить другое.
2. Создать приложение, которое позволяет вести учет семейных доходов и расходов, получить информацию о всех источниках доходов, суммах и датах получения доходов из каждого источника, о всех причинах и суммах расходов.
3. Создать приложение, которое позволяет начертить любой правильный n-угольник.
4. Создать приложение, которое позволяет нарисовать любое из платоновых тел (правильных многранников).
5. Создать приложение, которое позволяет нанести точки на плоскость и, соединияя некоторые из них прямолинейными отрезками, автоматически образовать минимальный выпуклый многоугольник, содержащий все точки. Как вариант: приложение должно позволять вводить точки заданием их координат.
6. Создать приложение, которое позволяет ввести необходимую информацию и по ней нарисовать бинарное дерево.
7. Создать приложение, которое позволяет задать и нарисовать любой граф, выделить в нем кратчайший путь из одной заданной вершины в другую.
8. Создать приложение, позволяющее пользователю играть с компьютером в игру, состоящую в бросании двух игральных костей: выигрывает тот, кто наберет больше очков. Интерфейс приложения должен указывать очередь хода, победителя и выяснять, продолжать ли игру.
9. Создать приложение, позволяющее пользователю играть с компьютером в карточную игру «21 очко»: выигрывает тот, кто наберет больше очков, но не более 21. Интерфейс приложения должен указывать очередь хода, спрашивать, будет ли игрок боать следующую карту, объявлять победителя, выяснять, продолжать ли игру.

10. Создать приложение решающее классическую задачу программмирования: подсчитать, какое число "шагов"сделает "пьяный таракан"пока не коснется всех квадратных плиток пола, имеющего N плиток в ширину и M плиток в длину. Сделать "шаг"таракан может с равной степенью вероятности на любую из соседних плиток. Интерфейс приложения должен позволять задать размеры пола (N, M), вывести общее число шагов, сделанных тараканом по полу, и сколько раз каждая плитка пола посещалась тараканом (см. [1, ch. 11]).

Предметный указатель

GeometricFigures, 133

C++, 3

адаптер, 22
адаптер аспекта, 32
архитектура MVC, 16
аспект адаптера, 101

виджет
  List, 91
  TextEditor, 85
  ViewHolder, 133
  свойства, 25, 32

горячая точка, 138
графический контекст, 121
графический объект, 121

зоны выбора цвета, 38

иждивенец, 9
инструмент
  Hot Region Editor, 29
  Image Editor, 29
  Menu Editor, 27, 60
  Named Fonts, 42
  ResourceFinder, 26, 104
  UIPainter, 24

класс
  AddresBook1, 97
  AddressBook2, 103
  AddressBook, 83
  ApplicationWindow, 124
  AspectAdaptor, 23, 101
  CharacterArray, 117
  ClickableController, 141
  ClickableImage, 143
  ClickableView, 143
  ControlManager, 124
  ControllerWithMenu, 138
  Controller, 17, 137
  Cursor, 140
  Dialog, 52
  DisplaySurface, 122
  DrawObjects, 128
  Event, 131, 138
  GraphicsContext, 126
  GraphicsHandle, 122
  GraphicsMedium, 121, 122
  InputSensor, 137
  InputState, 67
  KeyboardProcessor, 124
  List, 90
  MenuValueExample, 66
  MenuBuilder, 63, 65
  MenuCommandExample, 69–71
  MenuModifyExample, 74–76
  MenuSelectExample, 79
  MenuSwapExample, 76
  MenuValueExample, 71, 78
  Model, 12, 17
  MultiSelectionInList, 95
  NoController, 138
  Paint, 122
  ParagraphEditor, 138
  Person, 97, 101
  PluggableAdaptor, 23
  ProtocolAdaptor, 102
  ScheduledWindow, 123
  SelectionInList, 91

SequenceableCollectionSorter, 91
SimpleDialog, 52
StandardSystemController, 124, 138
String, 117
Symbol, 117
TestText, 119
TextEditorController, 138
Text, 117
Tic-Tac-Toe, 109
ValueHolder, 23, 64, 85, 92, 102
ValueModel, 13, 102
View, 17, 131
Window, 123
ApplicationModel, 24
команда
Accept, 84
Browse, 83, 111
Define, 34, 62, 83, 98, 111, 128, 133
Install, 33, 83, 98
New Item, 60
New Submenu Item, 61
выравнивания виджетов, 39
группировки виджетов, 41
установки позиции окна, 37
установки размеров окна, 36

ленивая инициализация, 84

метка виджета, 116
метод
postBuildWith:, 114
действия, 83, 84, 87, 93, 98, 106
доступа, 83, 84
инициализации, 87, 93, 96, 98
механизм зависимости, 8
модель
данных, 21
предметной области, 21
прикладная, 21
приложения, 21
модель значения, 22

наследование, 8

окно
Transcript, 93
Canvas, 25
Painter Tool on:, 25
Palette, 25

переключатель
Add Initialization, 35
переменная
аспектная, 83
переменная класса
DependentsFields, 10
переменная экземпляра
collectionSize, 90
collection, 90
component, 123
controller, 18, 123, 143
dependents, 90
hotSpot, 140
image, 140
limits, 90
listHolder, 92
mask, 140
menuHolder, 138
model, 18, 123, 137
name, 140
performer, 138
selectionIndexHolder, 92
sensor, 137
view, 18, 137
поверхность отображения, 121
подпредставление, 133
поле
Action, 34
Aspect, 34
пункт меню
горячая клавиша, 61
именной ключ, 73, 74

индикатор выбора, 62
мнемоника, 61, 69
мнемонический литерал, 61
свойство ID, 72

ресурс приложения, 26, 34
родитель, 9

свойство
Bordered, 111
Enablement Selector, 61
ID, 109, 111, 112
Indication Selector, 62
Value, 61
Смолток, 3
сообщение
accessWith:assignWith:, 103
addDependent:, 10
addItemLabel:value:, 75
asFiller, 136
asStroker, 136
asValue, 84
atNameKey:, 73
background:, 125
beOff, 78
beOn, 78
bePartner, 125
choose:fromList:values:
lines:cancel:, 55
choose:labels:values:default:, 53
closeRequest, 125
collapse, 125
confirm:, 53
currentCursor, 141
currentWindow, 123
disable, 72
displayBox, 125
display, 125
emphasizeFrom:to:with:, 118
enable, 72
enterEvent:, 141
entryEvent:, 138
exitEvent:, 138
expand, 125
expressInterestIn:for:sendBack:,
12
forAspect:, 101, 105
hideItem:, 74
label:, 125
labelImage:, 68
labelString:, 113
label, 113
list:, 92, 94
list, 92
mainWindow, 124
map, 125
menuAt:, 75
menuItemLabeled:, 73
moveTo:, 125
newBoolean, 87
newFraction, 87
newString, 87
onChangeSend:to:, 13, 64, 92,
105
redButtonPressedEvent:, 145
removeDependent:, 10
removeItem:, 76
request:initialAnswer:onCancel:, 55
request:initialAnswer:, 54
request:, 54
requestFileName:default:, 55
requestFileName:, 55
retractInterestIn:for:, 13
selection:, 92
selectionIndex:, 92
selectionIndexes:, 95
selectionIndexes, 95
selectionIndex, 92
selections:, 95
selections, 95
selection, 92
showWhile: aBlock, 140
show, 140

slave, 125
subject:, 101
subjectChannel:, 102, 105
subjectSendsUpdates:, 101
text, 113
unhideItem:, 74
unmap, 125
value:, 64, 85, 98, 101, 104
value, 71, 85, 101
warn:, 52
with:, 87
yourself, 94
confirm:initialAnswer:, 53
controllerAt:, 112
mainWindow, 113
redButtonPressedEvent:, 142
widgetAt:, 113
windowMenuBar, 113
wrapperAt:, 112
о модификации, 9, 11
об изменении, 9, 10
спецификация интерфейса, 33
сценарий, 110

табуляция, 26
тема адаптера, 101
тематический канал, 102

флажок
Add Initialization, 83, 98
Initially Disabled, 104, 107
Multiple Selection, 95
форма курсора, 140

холст, 24
хранитель значения, 32
хранитель представления, 133

Литература

[1] *Goldberg A., Robson D.* Smalltalk-80: The Language. — Reading, MA: Addison-Wesley, 1989.

[2] *Hopkins T., Horan B.* Smalltalk: An introduction to application development using VisualWorks. — Prentice-Hall, 1995.

[3] *Lewis S.* The Art and Science of Smalltalk. — Prentice Hall, 1995.

[4] *Linderman M.* Developing Visual Programming Applicftions Using Smalltalk. — New York: SIGS Books, 1996.

[5] *Sharp A.* Smalltalk by Example. The developer's guide. — McGraw-Hill, 1997.

[6] VisualWorks: Application Developer's GuideVisual.— Cincom, 2005.

[7] VisualWorks: GUI Developer's Guide.— Cincom, 2005.

[8] VisualWorks: Internationalization Guide — Cincom, 1995 – 2003.

[9] VisualWorks: Walk Through. — Cincom, 2000 – 2005.

[10] *Бадд Т.* Объектно-ориентированное программирование в действии. — СПб.: Питер, 1997.

[11] *Буч Г.* Объектно-ориентированное проектирование с примерами применения. — М.: Конкорд, 1992.

[12] *Кирютенко Ю. А., Савельев В. А.* Объектно-ориентированное программирование. Язык Smalltalk. — М.: «Вузовская книга», 2007.

[13] *Кирютенко Ю. А.* Объектно-ориентированное программирование. Среда VisualWorks. — М.: «Грант ЮФУ», 2007.

[14] *Савельев В. А.* Архитектура MVC (перевод с английского). — http://www.math.rsu.ru/smalltalk/.

[15] Смолток. Объектно-ориентированная система программирования. Руководство пользователя, часть 1, 2, 3. — М.: Ин-т проблем информатики РАН, 1995.

Оглавление

Введение **3**

I Механизмы и инструменты построения GUI **6**

1 Механизм зависимости **8**
1.1 Основной механизм зависимости 9
1.2 Механизм зависимости класса Model 12
1.3 Сообщение expressInterestIn:for:sendBack: 12
1.4 Сообщение onChangeSend:to: 13
1.5 Особенности механизма зависимости 14
1.6 Контрольные вопросы 15

2 Архитектура MVC **16**
2.1 Основные понятия 16
2.2 Взаимодействие объектов MVC 18
2.3 Расширение MVC в VisualWorks 21
2.4 Контрольные вопросы 23

3 Среда GUI системы VisualWorks **24**
3.1 Инструмент UIPainter 24
3.2 Resource Finder 26
3.3 Menu Editor 27
3.4 Image Editor 28
3.5 Hot Region Editor 29
3.6 Контрольные вопросы 30

4 Построение приложения в UIPainter **31**
4.1 Создание интерфейса пользователя 31
4.2 Форматирование холста 36
4.3 Открытие и закрытие окна приложения 44
4.4 Контрольные вопросы 47

Проектные задания к модулю I **49**

II Классы, используемые при построения GUI **50**

5 Диалоговые окна **52**
5.1 Стандартные диалоговые окна 52
5.2 Связь с главным окном 57
5.3 Создание диалогового окна пользователем 58
5.4 Контрольные вопросы 59

6 Меню **60**
6.1 Создание меню 60
6.2 Добавление меню в интерфейс 68
6.3 Программный доступ к меню 72
6.4 Контрольные вопросы 79

Проектные задания к модулю II **80**

III Примеры построения приложений с GUI **81**

7 Приложение AddressBook **83**
7.1 Определение класса AddressBook 83
7.2 Протокол аспектных методов и ValueHolder 84
7.3 Протокол инициализации 87
7.4 Протокол действий 87

8 Использование списков **90**
8.1 Списки 90
8.2 Виджет для списка 91
8.3 Пример с объектами ValueHolder и списком 96

9 Адаптация виджета **101**
9.1 Класс AspectAdaptor 101
9.2 Пример приложения с адаптером аспекта 103

10 Прямое управление виджетами **109**
10.1 Игра в крестики–нолики 109
10.2 Текст в виджетах 117

11 Графика в VisualWorks **121**
11.1 Поверхность отображения 121
11.2 Графический контекст 125
11.3 Использование MVC при отображении графики 130

12 Использование контроллеров **137**
12.1 Классы контроллеров 137
12.2 Пример: приложение для отображения точек 139

Проектные задания к модулю III **147**

Предметный указатель **148**

Литература **153**