Министерство общего и профессионального образования
Российской Федерации
РОСТОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Ю.А. Кирютенко В.А. Савельев

# Объектно-ориентированное программирование и язык Smalltalk.

## Построение интерфейса приложений в системе Smalltalk Express

Ростов-на-Дону
1999

**Ю.А. КирютенкоВ.А. Савельев**

# Объектно-ориентированное программирование и язык Smalltalk.

# Построение интерфейса приложений в системе Smalltalk Express

## Аннотация

Методическая разработка посвящена объектно-ориентированной методологии программирования и языку Smalltalk и является продолжением вышедших ранее методических разработок по объектно-ориентированному программированию и языку Smalltalk. В данной методической разработке описывается разработка приложения и его интерфейса в системе Smalltalk/V for Windows фирмы ParkPlace/Digitalk (США).

Методическая разработка предназначена для студентов 3–5 курсов механико-математического факультета и слушателей ФПК.

Печатается в соответствии с решением кафедры математического анализа Ростовского государственного университета, протокол № 5 от 9 января 1999 года.

Настоящие методические указания набраны в системе LaTeX с использованием кириллических шрифтов семейства «Литературная» (AOParaGraph) и математических шрифтов семейств Computer Modern и Math Symbol (American Mathematical Society).

# 1 Окна, панели и события

Система классов, обеспечивавшая в системе Smalltalk-80 интерфейс пользователя на основании триады модель-вид-контроллер (Model-view-controller — MVC) [3], в системе Smalltalk/V for DOS (изначально, V 286) была упрощена, и, во избежание путаницы, соответствующая триада была переименована в модель-панель-диспетчер (model-pane-dispatcher — MPD) Таким образом, вид и панель — почти синонимы, так же как контроллер и диспетчер, так как системы классов пользовательского интерфейса в этих средах совершенно различны.

Распространение на персональных компьютерах собственных оконных систем потребовало приспособления к ним. Для этого было два пути: (1) вернуть полную функциональность триаде MVC, позволяющей создавать многоплатформенный пользовательский интерфейс, одинаково функционирующий под различными оконными системами; (2) приспособить язык и систему классов к особенностям оконных систем персональных компьютеров.

Фирма Digitalk выбрала второй путь, который позволил создать весьма компактные и эффективные системы Smalltalk/V for Windows и Smalltalk/V for OS/2. Эти системы используют совершенно другую идеологию [2]: они полностью опираются на *события* (в смысле Windows – OS/2) и поддерживают отличную от других систем программную среду, которую, из-за отсутствия подходящего названия, часто называют программной средой, управляемой событиями.

Именно последнюю систему управления окнами, поддерживаемую Smalltalk/V for Windows и SmalltalkExpress for Windows мы далее опишем и приведём примеры. Для этого надо понимать, что такое событие окна среды Smalltalk, знать небольшую часть иерархии классов, связанную с окнами, иметь общее представление о функционировании операционной системы Windows и о механизмах программной среды, управляемой событиями.

## 1.1 Классы ViewManager и Window

Чтобы писать интерактивные приложения в системе Smalltalk/V, надо понять технологию построения окон системы. Smalltalk/V управляет интерфейсом совместно с оконной системой (Windows или Presentation Manager/Workplace Shell), благодаря администратору окон WindowManager. До широкого распространения доступных графических интерфейсов пользователя на персональных компьютерах, базирующих на Intel-процессорах, система Smalltalk/V внутри своей среды разработки полностью реализовывала все необходимое для построения интерфейса. С появлением интерфейса пользователя MS Windows, системы Smalltalk/V стали использовать средства его графического интерфейса, позволяя программировать Windows-приложения внутри полностью объектно-ориентированной оболочки.

Из всех объектов, заполняющих мир Windows, видимыми и, возможно наиболее важными, является окна. Окна (Window) — «точки соприкосновения» внешнего мира с приложениями. В то время как приложение создаёт окно для взаимодействия с пользователем, его поведение и функции управления, кооперируясь, управляют взаимодействием между приложением и операционной системой. Многое из общего поведения окон — место расположения на экране, действие полосы прокрутки, изменение размеров, и т.д. — автоматически обрабатывается объектом **WindowManager.** Основная забота разработчика — «полировка алмаза», то есть создание специального окна, которое реализует в себе объекты, обеспечивающие информационное взаимодействие с вашим уникальным приложением.

В системе **Smalltalk/V**, приложение с графическим интерфейсом пользователя обычно включает экземпляры подклассов представляющих окна классов: **ViewManager** (**АдминистраторОкна**) и **Window** (**Окно**). Как правило, создавая новое приложения, программист, помимо классов, описывающих объекты приложения, должен создать подкласс класса **ViewManager,** связанный с приложением. Экземпляр этого класса — окно приложения, ответственен за создание окна в целом (например, строки заголовка окна, другой связанной с окном информацией, подобной системному меню), управляет изменениями размеров окна, а также следит за панелями, содержащимися в окне и отображающими информацию, связанную с приложением. В свою очередь, связанное с окном приложение, полностью ответственно за предоставляемую интерфейсу информацию и за координацию между внутренними панелями. Например, если произведён выбор записи в списковой панели, приложение может отреагировать на это, изменением содержания связанной со списковой панелью текстовой панели. Эта координация между внутренними панелями окна достигается через *события* окна среды **Smalltalk**. Вот два простых примера: щелчок по кнопке мыши порождает **#clicked**-событие (событие щелчка по кнопке); выбор в списковой панели генерирует **#select**-событие (событие выбора элемента списка). Для обработки событий, классы самого приложения и класс окна приложения должны действовать совместно.

Стоит отметить, что классы приложения «знают» только об абстрактных аспектах решаемой проблемы. Слово «абстрактный» означает, что прикладные классы описывают структуру и поведение сущностей, которые относятся к вашему приложению, а не то, как эти сущности представляются в окне или как эти представления ведут себя в окне. Класс окна приложения создаётся главным образом для того, чтобы определить структуру и поведение окна, которое обеспечивает интерактивное, визуальное представление объектов, моделируемых прикладными классами. Как правило связь между классами приложения и окна приложения реализуется через переменные экземпляра. Например, экземпляр системного класса **ClassBrowser** имеет следующие переменные экземпляра:

**browsedClass** — класс, который в настоящее время просматривается;

**selectedDictionary** — текущий словарь сообщений (или класса или экземпляра) просматриваемого класса;

**selectedMethod** — в настоящее время выбранный метод из выбранного словаря сообщений.

Класс **Window** и его подклассы ответственны за низкоуровневое взаимодействие с окнами системы Windows. Сам класс **Window** — это абстрактный класс, обеспечивающий общие данные и протокол, наследуемые всеми подклассами, представляющими в **Smalltalk** Express окна MS Windows. В частности, они ответственны за обработку сообщений Windows, понимаемых **Smalltalk** (т.е. экземпляр **Window** является объектно-ориентированной оболочкой функции окна Windows). Методы, обрабатывающие сообщения Windows, по соглашению имеют имена, совпадающие с мнемоническим обозначением этого сообщения в файле заголовков `windows.h` или в заголовочном файле какой-либо подсистемы MS Windows. Связь этих методов **Smalltalk** с сообщениями Windows устанавливается с помощью глобального массива **WinEvents**, если код сообщения $< 1024$, и с помощью словаря **ExtWinEvents** в противном случае. Эти наборы хранят селекторы методов, обрабатывающих сообщения Windows с кодами, равными индексу массива (ключу словаря). Если для какого-то кода нет сопоставленного ему метода, то такое сообщение будет обрабатывать функция окна по умолчанию, предоставляемая самой системой Windows. Фактически создание нового подкласса **Window** имеет смысл при создании новых виджетов, а также создание такого подкласса и/или расширение самого класса **Window** требуется, если необходимо научить окна обрабатывать новые сообщения, например, связанные с реализацией доступа к новым программным интерфейсам (API), не предусмотренных в самой системе **Smalltalk** Express, например, к берклеевским гнёздам (winsocks) или для работы с нестандартной аппаратурой.

Заметим, что добавлению нового обработчика сообщения в массив **WinEvents** или словарь **ExtWinEvents** должно предшествовать добавление соответствующего метода в класс **Window**.

В **Smalltalk** Express предусмотрена возможность создавать приложения, включая новый класс в иерархию класса **ApplicationWindow** — подкласса класса **Window**. Этот класс предоставляет общий протокол для окон приложений и других окон верхнего уровня. Такое приложение имеет смысл создавать если необходимо добиться глубокой интеграции с Windows и нет необходимости поддерживать сложную модель и многооконный интерфейс.

Стандартными строительными блоками создаваемого интерфейса служат подклассы класса **Window,** в основном классы **TopPane, SubPane** и их подклассы.

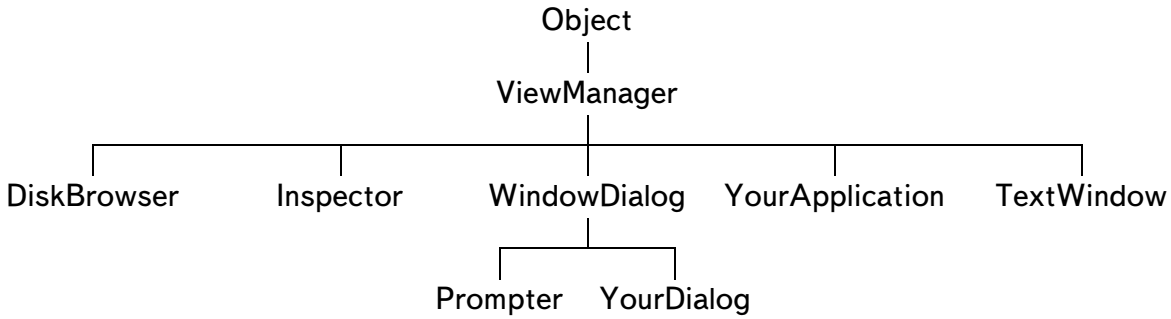


Рис. 1: Некоторые администраторы окон в системе Smalltalk/V

Используемые в комбинации, эти классы облегчают разработку и реализацию сложных прикладных окон.

При построении сложных приложений (особенно если они не требуют специальных возможностей Windows) лучше создавать их как подклассы класса ViewManager. Класс ViewManager — абстрактный класс, реализующий общий протокол для той части приложения, отвечающей за интерфейс с пользователем (или шире — с внешним миром). Он может управлять многими окнами (представлениями) одного и того же приложения. Такое представление не обязано быть визуальным (окном системы Windows). Хотя каждое приложение в младших версиях Windows должно иметь окно, класс ViewManager поддерживает вместе с многооконным интерфейсом и невизуальные представления: звуковые, клиент-серверные и др.

Итак, как мы видим, класс окна приложения тесно связан с его прикладными классами, но он отличается от них. Сохранение в разрабатываемом приложении явного разделения между классами моделируемых объектов и классом или классами окна приложения, которые обеспечивают взаимодействие с этими объектами, помогает сохранять организованными и представляемыми в реализации приложения любой сложности. Если же приложение небольшое, то вполне возможно решить задачу с помощью только одного класса — подкласса класса ViewManager.

В дополнение к создаваемым новым классам окон, проектировщики должны создавать и окна диалога, которые позволяли бы обрабатывать в интерактивном режиме запросы пользователя к приложению и запросы приложения к пользователю. Примером диалогового окна в Smalltalk/V является экземпляр класса Prompter (Промптер) — подсказчик. При создании нового окна для диалога, проектировщик должен создавать новый подкласс класса WindowDialog. Диалоговые окна (экземпляры класса WindowDialog), в отличии от окон приложений, являются модальными окнами — окно его вызвавшее не может продолжить свою работу до тех пор, пока модальное окно не будет закрыто и не обеспечит необходимой информацией первоначальное окно.

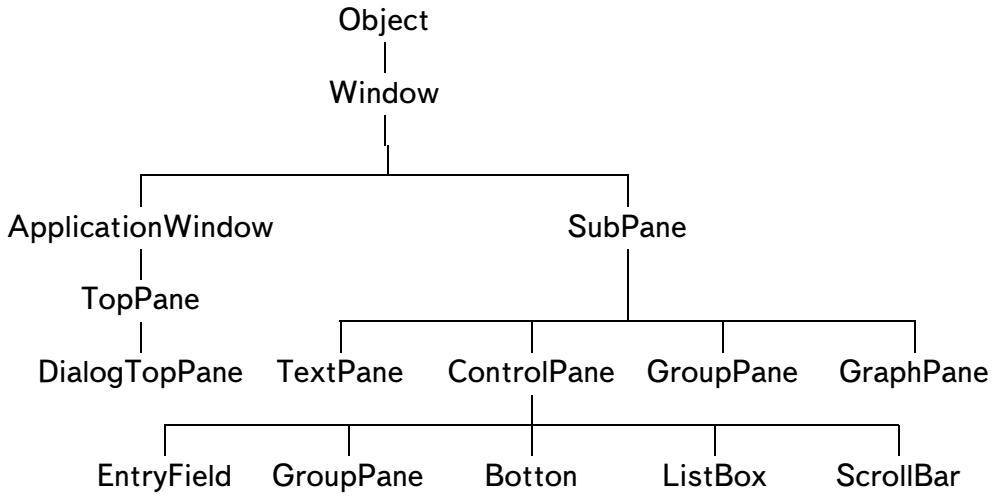


Рис. 2: Часть иерархии классов окон системы Smalltalk/V

Часть иерархии классов, которая включает некоторые основные классы окон (панелей), показана на рисунке 2.

В качестве примера, рассмотрим создание окна, имеющего только одну текстовую панель. Чтобы создать простейшее окно такого вида, достаточно вычислить следующий код:

```
| learnWindow |
learnWindow := TextWindow windowLabeled: 'Learning Status'
                    frame: (100 @ 100 extentFromLeftTop: 400 @ 200)
```

TextWindow — один из подклассов класса ViewManager, экземпляр которого уже содержит текстовую панель (экземпляр класса TextPane), обеспечивая все возможности по интерактивному редактированию текста, производимому пользователем с помощью клавиатуры и мыши. Посылка классу сообщения windowLabeled:frame: создаёт на экране окно, занимающее прямоугольную область с началом в точке экрана 100 @ 100 и экстентом 400 @ 200, с задаваемым аргументом-строкой заголовком, расположенным по центру в поле рамки окна. Чтобы закрыть окно, достаточно выбрать опцию Close (Закрыть) в системном меню окна.

Создавая окно, можно задать информацию о цветовой гамме окна, устанавливая цвета символа (foreColor:) и цвет фона (backColor:). Например, то же окно, что и в предыдущем примере, но с жёлтым цветом фона и синими символами при наборе в нем текста, будет создано, если вычислить следующий код:

```
| learnWindow |
learnWindow := TextWindow windowLabeled: 'Learning Status'
                    frame: (100 @ 100 extentFromLeftTop: 400 @ 200)
                    foreColor: ClrBlue
                    backColor: ClrYellow
```

Чтобы реально построить окно для приложения и вывести его на экран, в специально созданном для этого приложения подклассе класса ViewManager достаточно создать метод экземпляра open, который полностью сгенерирует внутреннюю структуру окна приложения и, в заключение, выполнит сообщение openWindow, отображая его. Метод open будет выглядеть примерно так:

```
open
  "Открыть окно для приложения."
  self labelWithoutPrefix: 'Sample Application'.
  self addSubpane: (PaneClass new
                  ... specifications ...
                  yourself).
  self addSubpane: ...
      ...
  self addSubpane: ...
  self openWindow
```

## 1.2 Постановка задачи учебного приложения

Для того, чтобы приводить примеры и в конце концов построить полное приложение, мы будем разрабатывать проект приложения под названием “Телефонная книга”. Постановка задачи следующая: есть много людей, которым я регулярно звоню по телефону, и я хочу создать приложение, в окне которого

- будет отображаться список имен таких людей и их телефонных номеров;
- список имен будет сортироваться в алфавитном порядке;
- при выборе имени из списка должен отображаться связанный и ним номер телефона;
- имена и номера телефонов могут добавляться в список и удаляться из него.

Размышляя над проблемой, можно возвращаться к исходной постановке задачи, изменять или расширять её, совершенствуя цели, и тем самым продвигаясь вперёд в её решении. Цель должна состоять в краткой и понятной формулировке задачи, что затем позволит правильно определить тот способ её решения, который в наибольшей степени соответствует предъявляемым требованиям.

Опираясь на формулировку задачи, давайте представим, как должно выглядеть то новое окно, которое поможет её решить, при этом не будем пока беспокоится о внутренних механизмах приложения и включаемых структурах данных. Сконцентрируем внимание на внешнем виде и взаимодействии. На что похоже

приложение? Как оно себя ведёт? Какие их существующих окон системы Smalltalk/V подходят в качестве первого приближения?

Как представляется, окно телефонной книги должно состоять, во-первых, из панели, отображающей список лиц, в котором можно выбирать одно из них, в который можно добавлять новых, и из которого не нужную больше информацию можно удалять. Во-вторых, нужна панель, в которой в ответ на сделанный в первой панели выбор будет отображаться соответствующий телефонный номер, если он уже существует, а если его ещё нет, панель должна позволять его ввести и отредактировать. Таким образом, с каждой из панелей надо связать определённые действия, которые полезно представлять как пункты меню. Наличие идей относительно числа, размера и содержимого панелей окна вашего приложения безусловно полезно, но размышление в терминах выборок из меню все ставит на свои места. Пункты меню — посредники, используемые пользователем для того, чтобы посылать сообщения приложению.

Итак, мы преобразовали сформулированную задачу в эскиз конкретного окна, которое надо будет сформировать. Создание рисунка — спецификация прикладного интерфейса пользователя и отправная точка для определения остальной части приложения.

Зная столько, сколько возможно, о решаемой проблеме, и имея в наличии эскиз прикладного интерфейса, можно идентифицировать классы объектов, которые и реализуют приложение. Перечислим и новые классы, определяемые для приложения, и существующие классы из иерархии классов системы Smalltalk/V, которые нужно будет использовать. Существующие классы, используемые в данном приложении, будут подклассами классов интерфейса пользователя ViewManager, Menu, Window, и подклассами базовых классов системы Collection, Magnitude, String и Stream.

Итак, прежде всего нужен новый подкласс класса ViewManager, назовём его PhoneBook (ТелефоннаяКнига), экземпляры которого являются полно-функциональными телефонными книгами. Ввиду простоты поставленной задачи, для её решения никаких других новых классов не потребуется. Поскольку в вашей модели окна появятся меню и панели, нам понадобятся экземпляры классов Menu (Меню), ListPane (СписковаяПанель) и TextPane (ТекстоваяПанель). После некоторых рассуждений, легко прийти к выводу, что словарь (экземпляр класса Dictionary) более всего подходит для связи между именами людей и их номерами телефонов, а сами имена и номера телефонов будут представляться как строки (экземпляры класса String).

Идентифицированные до сих пор классы отражают естественное представление сформулированной задачи и её изображения в виде окна. Но что, если выбор не столь очевиден? Важно сделать некоторый выбор, даже если он и не самый лучший. Можно и ошибиться, но в среде, которая легко позволяет делать измене-

ния, можно позже произвести улучшающие приложение изменения. Любой прогресс в разработке приложения расширит понимание того, какие классы следует использовать.

Когда задача сформулирована, есть рисунок окна и выделены используемые классы, нужно определить необходимы переменные в новых классах. В данном примере, единственный новый класс — класс PhoneBook — является визуализирующим классом для нашего приложения. Стандартное окно и классы панелей уже следят за большинством состояний, связанных с окном. Наше окно, экземпляр класса PhoneBook, нуждается только в конкретных данных приложения, которыми в нашем случае являются

- словарь лиц и их телефонных номеров, назовём его phones;
- имя из допускающей выбор списковой панели, хранящееся в переменной selectedName.

Таким образом, определение нового класса должно быть таким

```
ViewManager subclass: #PhoneBook
  instanceVariableNames:
    'phones selectedName '
  classVariableNames: ''
  poolDictionaries: ''
```

Есть ли общие правила, позволяющие решить, какие переменные экземпляра следует использовать? Самый лучший учебник — обращение к интерфейсу объекта. Какие вопросы (сообщения) будут посылаться объекту? Каким будет поведение объекта?

Интерфейс объекта — те сообщения, на которые объект отвечает. Всегда полезно перечислить все те сообщения, которые должны реализовывать новые классы. Хорошая отправная точка для этого — расширения рисунка окна посредством внесения в него тех сообщений, которые будут использоваться для каждой панели.

Сообщения используются для

- инициализации переменных;
- обращения к меню, определяемому приложением;
- доступа к содержимому панелей в целях его отображения;
- сохранения содержимого изменённых панелей;
- выборки элементов из конкретного меню;

- открытия окна.

В нашем примере надо реализовать сообщения, позволяющие:

**setDictionary** — инициализировать переменные экземпляра;

**add** — добавить новое имя в телефонную книгу;

**list: aListPane** — заполнить aListPane именами людей из телефонной книги;

**listMenu: aListPane** — задать имя меню списка;

**nameSelected: aListPane** — отобразить номер телефона для выбранного имени из aListPane.

**open** — открыть окно телефонной книги;

**remove** — удалить выбранное имя из телефонной книги;

**text: aTextPane** — установить содержимое aTextPane в качестве номера телефона для выбранного имени.

**textFrom: aTextPane** — получить содержимое текстовой панели как номер телефона для выбранного имени и занести его в словарь.

Теперь наш проект полностью описан и единственное, что остаётся сделать — реализовать методы для сообщений. Хорошая отправная точка для оконного приложения — создать метод, открывающий окно (в нашем случае, метод open). С него и начнём, по ходу дела изучая возможности среды, управляемой событиями. Но прежде всего, нужно написать единственный в нашем случае метод класса (**new**), создающий новый экземпляр класса, и единственный частный метод среди методов экземпляра, инициализирующий переменные экземпляра (**setDictionary**).

```
PhoneBook class methods

new
   "Создать новый экземпляр класса."
   ↑ super new setDictionary


PhoneBook methods

setDictionary
   "Инициализировать словарь."
   phones := Dictionary new
```

## 1.3 Первые реализации метода open

Метод open в вашем приложении самый большой и самый сложный. Создавая его, будем двигаться постепенно, начиная с простейшей модели, по ходу дела усложняя её и изучая возможности классов окон системы. Следующее выражение вводит первую версию метода open, которое создаёт и открывает окно, полностью соответствующее по строению сделанному ранее эскизу, но в котором пока ещё нет ничего, связанного с приложением:

```
open
  "Создать окно для телефонной книги, определяя
  размеры панелей окна, а затем открыть окно."
self label: 'Phone Book '.
self addSubpane: (ListPane new
                  framingRatio: (0@0 rightBottom: 1/3 @ 1)).
self addSubpane: (TextPane new
                  framingRatio:( 1/3 @ 0) rightBottom: 1 @ 1)).
self openWindow
```

Метод open использует специальную переменную **self,** чтобы обратиться к новому экземпляру класса **PhoneBook,** приёмнику сообщения **open,** который по построению является экземпляром класса **ViewManager.** Экземпляр **ViewManager** — "скелет" окна, реализующий рамку окна, заголовок, иконки, стандартную для окон строчку меню и иконку системного меню. Первое, что происходит в методе — объекту сообщают, что надо установить метку окна равной строке **'Phone Book'.** Затем в окно добавляются (**addSubpane:**) две панели: экземпляры классов **ListPane** и **TextPane.**

Последняя строка метода посылает объекту сообщение **openWindow,** реализованное в классе **ViewManager,** что приводит к созданию окна и его появлению на экране. (Остаётся за кадром то, что **Smalltalk/V** связывается с главным администратором полиэкранного режима, экземпляром класса **WindowManager,** сообщая ему, как должно выглядеть и каким должно быть поведение прикладного окна; именно администратор полиэкранного режима выполнит все необходимое, создавая окно и под управлением среды **Smalltalk/V** помещая его на экран.)

Давайте внимательно посмотрим на определение панелей окна, например на выражение

```
self addSubpane: (ListPane new
                  framingRatio: (0@0 rightBottom: 1/3 @ 1)).
```

Именно здесь определяется, какую часть окна будет занимать создаваемая в окне панель. Когда ранее мы создавали окно **learnWindow** как экземпляр класса

TextWindow, исходный размер окна мы определяли абсолютно (frame: (100 @ 100 extentFromLeftTop: 400 @ 200)), что, впрочем, не совсем корректно, поскольку при этом никак не учитываются реальные размеры экрана. Но когда создаётся панель внутри окна, размеры которого могут меняться пользователем достаточно произвольно, занимаемая панелью часть окна должна определяться относительно размеров самого окна, то есть заданием её расположения относительно исходного прямоугольника. Например, выражение

```
aPane framingRatio: (Rectangle leftTop: 0@0 extent: (1/3)@(1/2)))
```

указывает, что создаваемая панель должна занять 1/3 окна в горизонтальном направлении и 1/2 окна в вертикальном направлении. В качестве альтернативы, размещение панели может определятся, используя блоки, посредством вычисления абсолютного размера панели через абсолютные размеры прямоугольника всего окна. Например, выражение

```
aPane framingBlock: [:box |
    (Rectangle leftTop: box leftTop extent: box extent * (1/3)@(1/2))]
```

определит то же расположение панели в окне, что и предыдущее.

Но созданное окно ничего не умеет делать по отношению к своей модели — словарю с именами людей (ключами словаря) и их телефонами (значениями ключей), за исключением некоторого интерактивного редактирования текста. Оно даже не умеет отображать содержимое словаря phones, связанного с окном, поскольку мы не определили механизмы, с помощью которых словарь мог бы связываться с окном. Кроме того, чтобы сделать окно способным выполнять ещё что-нибудь полезное, окну во время создания должны быть посланы дополнительные сообщения, которые определяют, что надо делать в ответ на производимые пользователем действия. Дадим вторую редакцию метода open.

```
open
 "Создать окно для телефонной книги, определяя размеры
  и поведение панелей окна, а затем открыть окно."
self label: 'Phone Book '.
self addSubpane: (ListPane new
                    owner: self;
                    when: #getContents perform: #list: ;
                    framingRatio: (0@0 rightBottom: 1/3 @ 1)).
self addSubpane: (TextPane new
                    owner: self;
                    when: #getContents perform: #text: ;
                    framingRatio:( 1/3 @ 0) rightBottom: 1 @ 1)).
self openWindow
```

В этом варианте метода, панелям посылаются по два дополнительных сообщения. Первое сообщение и там и там одно и то же:
owner: self — сообщает панели о её подконтрольности содержащему её окну; это означает, что панель может посылать сообщения “в” и получать сообщения “из” окна приложения.

Вторые сообщения имеют один и тот же селектор, но разные аргументы. Шаблон этого сообщения имеет вид when: #eventSymbol perform: #eventHandlerSelector:, и сообщает панели, что, когда в ней происходит событие, описываемое символом #eventSymbol, она должна послать сообщение eventHandlerSelector: (обработчик события) владельцу панели, в данном случае экземпляру класса PhoneBook. Такое сообщение должно иметь точно один аргумент — панель, в которой произошло событие.

Прежде, чем показать как работает написанный код и разрабатывать дальше метод open, рассмотрим подробнее события в окнах системы Smalltalk.

## 1.4 События системы Smalltalk

Итак, каждая панель имеет владельца, который, как правило, является окном приложения, содержащим панель. Когда панель нуждается в некоторой информации, относящейся к приложению, она запрашивает её косвенно через своего владельца, опираясь на механизм *события* языка Smalltalk. Событие окна системы Smalltalk (не путать с событиями, генерируемыми операционной системой) генерируется панелью и тогда, когда требуется некоторая информация от приложения, и тогда, когда требуется сообщить приложению о том, что произошло нечто существенное. Событие окна — не объект, события представляются именами; например, #clicked. Для того, чтобы сгенерировать (породить) событие в панели, надо выполнить выражение вида

```
aPane event: #eventSymbol.
```

Для того, чтобы приложение (или владелец) имели возможность ответить на это событие, приложение должно следующим образом обеспечить панель селектором для обработки события:

```
aPane when: #eventSymbol perform: #eventHandlerSelector:.
```

Селектор обработчика события, под которым скрывается стандартный метод языка Smalltalk, должен иметь точно один параметр — панель, в которой событие возникло. Следовательно, для обработки события приложение должно иметь метод следующего вида:

```
eventHandlerSelector: aPane
"Код ответа на событие #eventSymbol."
```

После чего приложение может явно связываться с панелью для того, чтобы или получить или изменить некоторый атрибут панели. Панель может связываться и с другими панелями, явно посылая им сообщения или генерируя для них события.

Панели получают имена при выполнении выражений типа:

```
aPane paneName: 'aName'.
```

Если панель названа, она может быть вызвана приложением посредством выполнения сообщения вида:

```
anApplication paneNamed: 'aName'.
```

Каждый вид панели может генерировать свой набор событий. Но есть события, которые генерируют все панели. Вот список наиболее важных из них вместе с ожидаемыми ответами приложения.

### Window Events (События окна)

#### #opened

**Событие возникает:** Как результат открытия прикладного окна прежде, чем реальное окно появится на экране.

**Ответ приложения:** Выполняет инициализацию, которая может быть выполнена только после того, как сформированы панели окна; этого нельзя сделать в методе initialize, поскольку панели в то время ещё не существуют.

#### #close

**Событие возникает:** Как результат попытки закрыть прикладное окно.

**Ответ приложения:** Завершает всю работу до исчезновения окна с экрана; может подтверждать закрытие окна и при этом выполнять “self close”, чтобы действительно закрыть окно, или ничего не делать и отменить закрытие.

## Common Event (Общие События)

### #getPopUpMenu

**Событие возникает:** Пользователь пытается обратиться к всплывающему меню.

**Ответ приложения:** Установить в панель всплывающее меню (экземпляр класса Menu) через сообщение setPopupMenu:.

### #getContents

**Событие возникает:** Событие #getContents генерируется каждый раз, когда панели посылается модификационное сообщение update. Кроме того, оно же генерируется (но только один раз), когда первоначально открывается окно приложения. Более точно: когда сообщение open посылается приложению, для всех панелей окна генерируется #getContents-событие, затем порождается #opened-событие, и, в заключение, окно появляется на экране.

**Ответ приложения:** Содержимое панели устарело. Что-то необходимо сделать, чтобы панель полностью обновилась; например,

- установить радио-кнопку так, чтобы она принимала значение “on” (включено);
- обеспечить списковую панель списком, который нужно отобразить на экране, и элементом, который должен быть выбран;
- обеспечить текстовую панель, текстом, который нужно отобразить на экране.

## Text Pane Events (События Текстовой Панели)

### #save

**Событие возникает:** Когда выбирается элемент save в всплывающем меню.

**Ответ приложения:** Получить текст из текстовой панели, и обработать его. Сообщить об этом текстовой панели (используя сообщение modified:) для того, чтобы панель знала, что содержащийся текст сохранён.

### List Pane Events (События Списковой Панели)

#select

**Событие возникает:** Когда пользователь любым образом выбирает элемент из списка.

**Ответ приложения:** Получить выбранный в панели элемент и как ответ возвратить его.

#doubleClickSelect

**Событие возникает:** Когда происходит двойной щелчок пользователя на элементе списка.

**Ответ приложения:** Получить выбранный в панели элемент и как ответ возвратить его.

### Graph Pane Events (События Графической Панели)

#button1Down нажата первая кнопка

#button1DownShift

#button1Up отпущена первая кнопка

#button1DoubleClick двойной щелчок первой кнопкой

#button1Move перемещение мыши с нажатой первой кнопкой

#button2Down нажата вторая кнопка

#button2Up отпущена вторая кнопка

#button2DoubleClick двойной щелчок второй кнопкой

#button2Move перемещение мыши с нажатой второй кнопкой

#mouseMove перемещение мыши

**Событие возникает:** Когда с мышью происходит указанное событие (обычно первая кнопка — левая, вторая — правая кнопки мыши, но в настройках Windows их можно поменять местами).

**Ответ приложения:** Отреагировать на движение мыши или нажатие кнопки мыши. Панель может запрашиваться о расположении мыши в время последнего (прошедшего) события, путём посылки ей сообщение mouseLocation.

### Button Pane Events (События Кнопочной Панели)

#clicked

**Событие возникает:** Когда пользователь нажимает кнопку.

**Ответ приложения:** Отреагировать на нажатие кнопки выполнением предписанных действий.

## 1.5 Протокол панелей системы Smalltalk/V

Приведём некоторые простые и наиболее интересные сообщения, понятные панелям.

### Text Panes (Текстовые Панели)

aTextPane contents — возвращает строку, отражающую содержимое панели.

aTextPane contents: aString — для панели-приёмника определяет её содержимое как строку aString.

aTextPane modified: aBoolean — если в текстовой панели сделаны изменения содержимого, значение переменной экземпляра modified становится равным true (истина); этот метод полезен для сообщения приложению о том, что содержимое панели было сохранено (как реакция на #save-событие), устанавливая значение переменной равным false.

### List Panes (Списковые Панели)

aListPane contents — возвращает массив (или упорядоченный набор) строк, отражающих содержимое панели.

aListPane contents: aIndexedCollection — для панели, приёмника сообщения, определяет как её содержимое набор aIndexedCollection, элементы которого должны преобразовываться (с помощью printString или другого сообщения, имя которого сообщается списку сообщением ) в экземпляры классов String или Symbol.

aListPane selectedItem — возвращает выбранную в списковой панели строку (если она существует) или **nil,** если ничего не было выбрано.

aListPane selection: aStringOrNil — определяет элемент, который будет выбранным элементом в панели (выделенным), равным аргументу **aStringOrNil;** при задание аргумента равным **nil** ничего не выбирается.

**Bottons (Кнопки)**

**aButton contents** — возвращает строку, которая является меткой (**label**) кнопки.

**aButton contents: aString** — устанавливает метку кнопки равной строке **aString.**

**aButton selection** — возвращает **true,** если кнопка нажата, иначе возвращает **false.**

**aButton setection: aBoolean** — устанавливает переменную состояния кнопки (**value**) равным аргументу **aBoolean** (**true** — нажата, **false** — не нажата.

**Graph Panes (Графические панели)**

**aGraphPane mouseLocation** — возвращает текущее положение курсора мыши в координатах панели.

Теперь, давайте посмотрим, как перечисленные понятия могут быть собраны в методе **open** так, чтобы обеспечить поведение приложения в ответ на происходящие события.

## 1.6 Завершение построения приложения PhoneBook

Вернёмся ко второму варианту метода **open** класса **PhoneBook** и рассмотрим сообщение **when: #getContents perform: #list:,** сначала посылаемое списковой панели. В том случае, когда необходимо получить содержимое панели (событие **#getContents**), списковая панель должна послать владельцу сообщение вида **list: aPane**, которое и позволит определить содержимое. Поскольку моделью приложения у нас является словарь **phones**, а отображать в списковой панели надо ключи словаря, обработчик данного события должен выглядеть так:

```
list: aListPane
    "Отобразить в списковой панели aListPane
    имена людей из телефонной книги."
    aListPane contents: phones keys asSortedCollection
```

Рассмотрим сообщение when: #getContents perform: #text:, посылаемое текстовой панели. Обработчик text: должен отображать в текстовой панели номер телефона лица, выбранного в списковой панели, и ничего не отображать, если выбора не сделан.

```
text: aTextPane
   "Определить содержимое текстовой панели как номер телефона
   того лица, которое выбрано в списковой панели."
   aTextPane contents: (phones at: selectedName
                        ifAbsent: [''])
```

Напомним, что каждая панель обязательно получает и отображает своё содержимое и тогда, когда им владеющее окно первоначально открывается, и тогда, когда её содержимое изменяется.

Чтобы завершить разработку метода open согласно сделанного ранее проекта, нам необходимо в списковой панели обрабатывать событие выбора элемента (#select), и встроить в неё меню, посредством которого можно было бы добавлять в список новые лица и удалять ненужные. В текстовой панели нам нужно обрабатывать событие #save, сохраняя введённый номер телефона. Таким образом текст метода open должен выглядеть так:

```
open
  "Создать окно для телефонной книги, определяя размеры
  панелей окна, их поведение; затем открыть окно."
self label: 'Phone Book '.
self addSubpane: (ListPane new
                  owner: self;
                  when: #getContents perform: #list: ;
                  when: #select perform: #nameSelected: ;
                  when: #getMenu perform: #listMenu: ;
                  framingRatio: (0@0 rightBottom: 1/3 @ 1)).
self addSubpane: (TextPane new
                  owner: self;
                  when: #getContents perform: #text: ;
                  when: #save perform: #textFrom: ;
                  framingRatio:( 1/3 @ 0) rightBottom: 1 @ 1)).
self openWindow
```

Реализация обработчика события #save для текстовой панели должна сохранить новое содержимое текстовой панели в словаре phones в качестве значения для выбранного в списковой панели ключа (лица) и сообщить об этом текстовой панели:

```
textFrom: aTextPane
    "Сохранить в словаре phones содержимое aTextPane как
    номер телефона выбранного в списковой панели лица."
    selectedName isNil ifTrue: [ ↑true].
    phones at: selectedName put: aTextPane contents.
    aTextPane modified: false
```

Обработчик события **#select** списковой панели должен отображать в текстовой панели номера телефона выбранного лица. Значит должна быть связь между этими двумя панелями. Для этого существуют специальные механизмы. Отвлечёмся от построения примера и коротко остановимся на этих механизмах. Есть два механизма для связи с панелями окна: явное, пользуясь сообщением-приказом панели модифицировать себя, и неявное через механизм change/update (изменить/модифицировать). Механизм change/update сам несёт ответственность за поиск панели с конкретным обработчиком события **#getContents.** Сравнение между двумя подходами можно отобразить следующей таблицей (под именем **#handler:** скрывается обработчик события **#getContents**):

| Прямое сообщение | Механизм change/update |
|---|---|
| `aPane update` | `self changed: #handler:` |
| `aPane selector` | `self changed: #handler:`<br>`with: #selector` |
| `aPane selector: parameter` | `self changed: #handler:`<br>`with: #selector`<br>`with: parameter` |

**Smalltalk/V** все время использует смесь из этих двух механизмов. Есть выражения подобные

```
aPane selection = 1
 ifTrue: [self changed: #handler with: #restoreSelected with:1].
```

Следующее выражение работает точно также:

```
aPane selection = 1 ifTrue: [aPane restoreSelected: 1]
```

Надо отметить, что возможность именовать панели — относительно недавнее добавление в **Smalltalk**. Инструментальные средства среды программирования, подобно браузерам иерархии классов, браузерам дисков и отладчикам, были

реализованы намного раньше, используя механизм change/update, и не изменились до сих пор. Что касается пользователя, он сам должен решать, использовать ли ему прямую передачу сообщений или механизм change/update. Прямая передача сообщений требует панели или имени панели, в то время как механизм change/update требует имени обработчика события **#getContents.**

Таким образом, возвращаясь к построению приложения **PhoneBook,** в обработчике события **#select** можно использовать сообщение **changed:**, которое отыщет в окне панель с методом **#text:** в качестве обработчика события **#getContents** и, тем самым, выполнит в ней необходимые операции.

```
nameSelected: aListPane
    "Отобразить в текстовой панели номер телефона
    в соответствии с выбором в списковой панели."
    selectedName := aListPane selectedItem.
    self changed: #text:
```

Остаётся только, в соответствии с сообщением **when: #getMenu perform: #inputMenu:,** написать метод, создающий и включающий специальное меню для списковой панели, и методы обработки выбора каждой из строк меню.

```
listMenu: aListPane
 "Определить всплывающее меню для aListPane."
 aListPane setMenu: ((Menu
                labels: '~Add\~Remove' withCrs
                lines: Array new
                selectors: #(add remove))
                title: '~Phones' )
```

Текст метода **textMenu:** требует пояснений. Прежде всего, элементы меню определяются строкой-аргументом ключевого слова **labels:** и отделяется друг от друга символом “backslash” (\), который в конце списка заменяются символом возврата каретки (благодаря сообщению **withCrs,** посылаемому строке). Символ ~ объявляет следующий после него символ подчёркнутым и одновременно определяет его пригодным для использования в качестве горячей клавиши. Массив-аргумент ключевого слова **lines:** — либо пустой массив, либо содержит номера строк меню, после которых надо провести в меню разделительную горизонтальную линию. Селекторы, перечисленные в массиве после ключевого слова **selectors:** — имена методов, которые будут выполнены тогда, когда будет выбран соответствующий пункт меню. Такие сообщение всегда посылается владельцу меню (в этом случае **self** — экземпляр класса **PhoneBook**). Ключевое слово **title:** со строкой в качестве аргумента, определяет имя данного меню и под этим именем устанавливает его в строку меню окна. Тем самым доступ к меню становиться возможным и через

строку меню окна, и через событие **#getMenu,** возникающее при нажатии в списковой панели правой кнопки мыши. Если в подобном методе заголовок меню не определяется, **Smalltalk/V** создаст его сам в виде '**Untitled**', и добавит меню с таким именем к строке меню, напоминая программисту, что необходимо обеспечить более описательный заголовок.

Наконец, чтобы сделать такое меню работающим, реализуем в классе **PhoneBook** следующие два новых метода:

```
add
 "Добавить новое имя в телефонную книгу."
 | key |
 self textModified
        ifTrue: [ ↑self].
 key := Prompter prompt: 'new name'
                  default: String new.
 (key isNil or: [phones includesKey: key ])
        ifTrue: [ ↑self].
 selectedName := key.
 phones at: key put: nil.
 self
   changed: #list:; "изменить список"
   changed: #list: with: #selection: with: key; "выбрать новый элемент"
   changed: #text:  "изменить текст"

remove
    "Удалить выделенное имя из телефонной книги."
    phones removeKey: selectedName
           ifAbsent: [ ].
    selectedName := nil.
    self
        changed: #list:;
        changed: #text:
```

Здесь есть несколько новых моментов. Прежде всего, обратим внимание на выражение **self textModified ifTrue: [self]** из метода **add,** которое проверяет остались ли в окне текстовые панели, содержимое которых не было сохранено, и, если это так, создаёт диалоговое окно, требуя от пользователя инструкций по отношению к несохраненной информации. Самое интересное в методе **add** — второе сообщения с ключевым словом **changed:**. Благодаря ему, по **#list** будет найдена та панель, которая понимает этот обработчик, в данном случае — списковая панель, и ей будет послано сообщение **selection: key,** выделяющее только что введённый

новый элемент.

В методе remove нет ничего неизвестного. Стоит обратить внимание только на то, что после удаления выделенного имени из списковой панели, нового выделенного имени не определяется (selectedName := nil), а потому и текстовая панель будет чистой.

Построение завершено. Чтобы начать работать можно, например, определить глобальную переменную MyPhoneBook, вычисляя выражение

```
MyPhoneBook := PhoneBook new
```

а затем открыть эту книгу для работы с ней:

```
MyPhoneBook open.
```

Чтобы сохранить всю введённую в телефонную книгу информацию, выходя из системы надо сохранить её образ.

## 1.7 О цикле разработки приложения

Если проанализировать то, что было нами сделано при построении класса PhoneBook, можно отметить несколько характерных моментов в разработке приложения на языке Smalltalk/V, среди них:

1. формулировка проблемы;
2. изображение окна;
3. идентификация классов;
4. описание состояний объектов;
5. описание интерфейса объектов;
6. реализация методов.

Как правило, первоначальная формулировка задачи, подобно самому приложению на языке Smalltalk, будет постепенно развиваться. Действия по претворению первоначальных идей проекта в жизнь, будут порождать новое понимание, способствовать расширению восприятия задачи и, в конце концов может изменить первоначальную оценку исходной проблемы и методы её решения. Все этапы разработки связаны друг с другом и, хотя они записаны последовательно, реализуются они, что и более правильно и более продуктивно, параллельно или итерационно (особенно этапы 3-5).

В течение первого круга разработки нового проекта, проходя через все выше упомянутые шаги, мы получаем, как правило, только “первый срез” разрабатываемого приложения. Полученный опыт его разработки, и первый опыт эксплуатации приложения, принесут возросшее понимание проблемы, породят новые подходы к её решению. И вновь, пройдя те же шесть шагов, уже не обязательно последовательно, позволит получить “лучшее” решение.

Возможности системы Smalltalk/V поддерживают (и даже, провоцирует) такой усложняющийся и эволюционный подход к разработке программного продукта, ведь Smalltalk/V — открытая система, предоставляющая множество возможностей для расширения. На практике, разработка приложения может оказаться очень соблазнительной. Всегда просто придумать и добавить всего несколько строк кода, которые сделают более изящным интерфейс приложения или повысят эффективность. Именно поэтому очень важно вспомнить о первоначальных целях проекта разрабатываемого Smalltalk/V-приложения. Если созданный прототип удовлетворяет заданным критериям, стоит остановится. Приобрести опыт работы с приложением, и это приведёт к последующим расширениям.

Возвращаясь к классу PhoneBook, нетрудно заметить, что есть в нашем проекте небольшой дефект. Внося в словарь новое лицо, можно и ошибиться. Если ошибку заметили во время набора, до внесения в словарь — нет проблем в её исправлении, как впрочем и в исправлении ошибочно введённого номера телефона. Если же ошибочный ключ уже внесён в словарь, то единственный способ исправить её — удалить и ввести заново, но при этом потеряется номер телефона. Поэтому стоит изменить метод listMenu, добавив в меню строку “Correct”, и написать связанный с ней новый метод. Метод listMenu примет вид:

```
listMenu: aListPane
 "Определить всплывающее меню для aListPane."
 aListPane setMenu: ((Menu
    labels: '~Add\~Remove\~Correct' withCrs
    lines: Array new
    selectors: \#(add remove correct))
    title: '~Phones' )
```

А новый метод будет иметь следующий код, надеемся понятный без всяких пояснений:

```
correct
 "Исправить ошибку в имени из телефонной книги."
 |old key |
 self textModified
        ifTrue: [ ↑self].
```

```
old := phones associationAt: selectedName.
key := Prompter prompt: 'Correct name:' default: selectedName.
((key isNil or: [phones includesKey: key ]) or: [key == selectedName])
        ifTrue: [ ↑ self].
phones removeKey: selectedName.
phones at: key put: old value.
selectedName := key.
self
    changed: #list:;
    changed: #list: with: #selection: with: key;
    changed: #text:
```

# 2 Приложение с кнопками

Проиллюстрируем базисные понятия, описанные при построении предыдущего примера, несколько расширяя их, ещё на одном примере, реализуя простое окно "перевода" слов с английского языка на русский. Постановка задачи почти ясна из формулировки: построить приложение, которое позволяет выбирать в любой из панелей существующее слово и получать в другой панели его перевод, если он ранее в неё был введён. Очевидный эскиз окна такого переводчика приведён на рисунке ниже.

При реализации этого приложения, в отличии от предыдущего, обработку списков будем проводить, используя кнопки, а не меню, для чего определим две списковые панели и три кнопки. Левая списковая панель — панель английских слов (ключей словаря, в котором будут храниться ассоциативные пары), в то время как правая — панель трансляции (панель русских слов). Когда слово выделяется в английской панели, соответствующий перевод выделяется в панели трансляции, и наоборот. Новые слова могут добавляться и удаляться из английской панели при использовании кнопок AddWord и DelWord, соответственно. Перевод может быть отредактирован при нажатии на кнопку Edit Translation.

Стратегия, используемая при реализации приложений, должна гарантировать, что полное состояние приложения поддерживается переменными экземпляра. Нам потребуются всего две переменные:

**translation** — содержит словарь с английскими словами в качестве ключей, и их перевод на русский в качестве значений;

**englishSelection** — определяет выбранное английское слово; равно **nil**, если выбор не сделан.

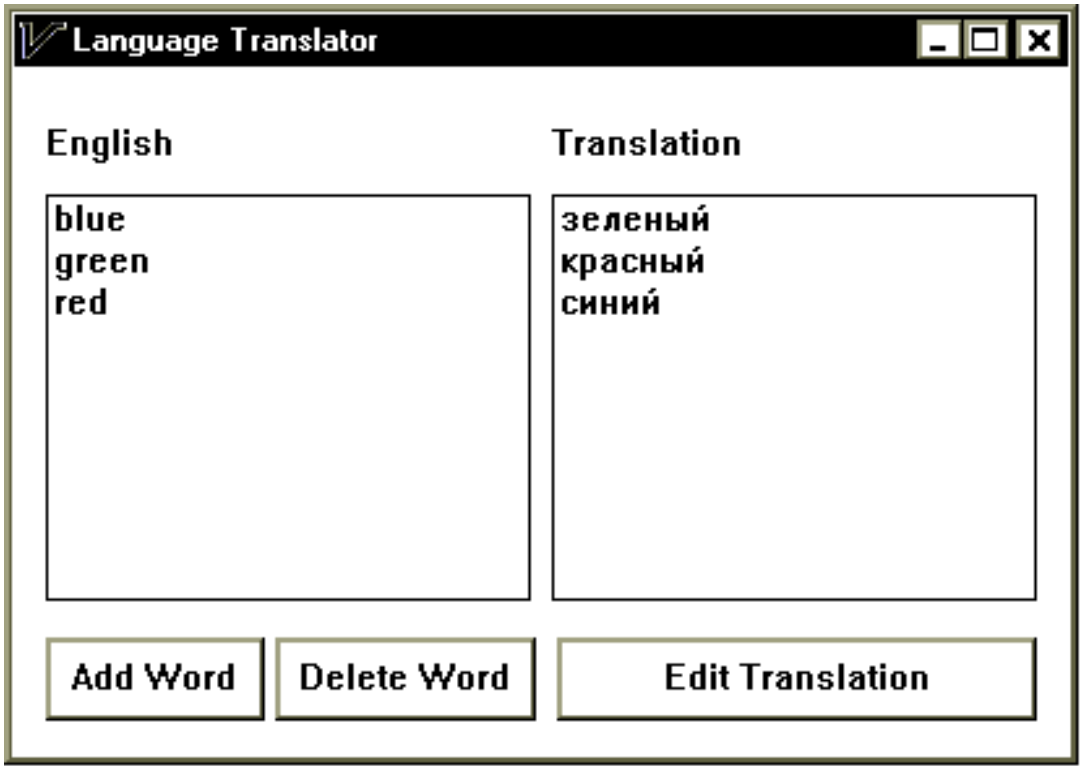


Рис. 3: Окно переводчика слов с одного языка на другой

По выбранному английскому слову, всегда можно определить соответствующее русское слово. Исходя из этой информации, можно полностью модифицировать приложение. Когда в приложении произошли изменения (например, произвели выбор в одной из панелей), надо вызвать соответствующий обработчик, который должен гарантировать, что состояние приложения в соответствии с происшедшими изменениями будет обновлено.

Перечислим все обработчики событий строящегося приложения:

**Окно LanguageTranslator**

- #opened $\Longrightarrow$ open

**Панель English**

- #getContents $\Longrightarrow$ updateEnglishList:
- #select $\Longrightarrow$ selectEnglishItem:

**Панель Translation**

- #getContents $\Longrightarrow$ updateTranslationList:

- #select ⟹ selectTranslationItem:

**Кнопка AddWord**

- #clicked ⟹ clickedAddWord:

**Кнопка DeleteWord**

- #clicked ⟹ clickedDeleteWord:

**Кнопка EditTranslation**

- #clicked ⟹ clickedEditTranslation:

Определение самого класса LanguageTranslator, определение метода класса new и метода экземпляра setDictionary, необходимых для создания нового экземпляра класса, очевидны.

```
ViewManager subclass: #LanguageTranslator
  instanceVariableNames: 'translation englishSelection'
  classVariableNames: ''
  poolDictionaries: 'ColorConstants'

LanguageTranslator class methods

new
  "Создать новый экземпляр."
  ↑ super new setDictionary


LanguageTranslator methods

setDictionary
  "Инициализировать переменную экземпляра."
  translation := Dictionary new.
```

Поскольку придётся искать по английскому слову его русский перевод, а по переводу, переведённое слово, нам потребуются два простых метода экземпляра, позволяющие это делать.

```
englishWordFor: aWord
   "Возвращает английское слово по его переводу aWord,
```

```
    если такое есть, иначе возвращает nil."
    ↑ translation keyAtValue: aWord ifAbsent: [nil]

translatedWordFor: anEnglishWord
    "Возвращает перевод на русский английского слова
    anEnglishWord, если таковой есть, иначе возвращает nil."
    ↑ translation at: anEnglishWord ifAbsent: [nil]
```

Метод open такого окна записывается почти аналогично предыдущему приложению. Различие состоит в том, что вместо меню используются кнопки. Обычно в системе Smalltalk Express метод open не пишется вручную, для его построения используется программа построения окон WindowBuilder Pro (которая, как мы предполагаем, станет темой следующей методической разработки). А пока обойдёмся без неё, не очень заботясь о "красоте" создаваемого окна, сосредоточим своё внимание на его функциональности.

```
open
    "Создать окно для переводчика, определяя размеры
    панелей окна, их поведение; затем открыть окно."
  self labelWithoutPrefix: 'Language Translator';
       noSmalltalkMenuBar.  "окно без стандартной строки меню"
  self addSubpane: (StaticText new
                owner: self;
                framingRatio:((1/20)@(1/20) corner: (9/20)@(3/20));
                contents: 'English:';
                leftJustified);
       addSubpane: (StaticText new
                owner: self;
                framingRatio:((11/20)@(1/20) corner: (19/20)@(3/20));
                contents: 'Translation:';
                leftJustified);
       addSubpane: (ListPane new
                owner: self;
                paneName: 'englishPane';    "логическое имя панели"
                framingRatio:((1/20)@(4/20) corner: (9/20)@(15/20));
                when: #getContents perform: #updateEnglishList:;
                when: #select perform: #selectEnglishItem:);
       addSubpane: (ListPane new
              owner: self;
              framingRatio:((11/20)@(4/20) corner: (19/20)@(15/20));
              paneName: 'translationPane';
```

```
            when: #getContents perform: #updateTranslationList:;
            when: #select perform: #selectTranslationItem:);
        addSubpane: (Button new
            owner: self;
            framingRatio:((1/20)@(16/20) corner: (4/20)@(19/20));
            paneName: 'addWordButton';
            when: #clicked perform: #clickedAddWord:;
            contents: 'AddWord');
        addSubpane: (Button new
            owner: self;
            framingRatio:((6/20)@(16/20) corner: (9/20)@(19/20));
            paneName: 'deleteWordButton';
            when: #clicked perform: #clickedDeleteWord:;
            contents: 'DelWord';);
        addSubpane: (Button new
            owner: self;
            framingRatio:((11/20)@(16/20) corner: (19/20)@(19/20));
            paneName: 'editTranslationButton';
            when: #clicked perform: #clickedEditTranslation:;
            contents: 'Edit Translation').
    self openWindow
```

Ниже для каждого обработчика, как помощь в понимании аспектов реализации, даны детализированные комментарии. В отличии от предыдущего приложения, обработчики событий будут активно использовать модифицирующее сообщение update, как то, которое реализовано в самом классе LanguageTranslator среди методов экземпляра, так и то, которое посылается панелям окна и реализовано в классе SubPane, и которое, в свою очередь, вызывает обработчик события #getContents.

```
update
    "Модифицировать все приложение, для чего модифицировать
    панели трансляции и английского языка. Посылка панелям
    сообщений update вызовет в каждой панели событие
    #getContents, в результате чего будут выполнены сообщения
    updateEnglishList: и updateTranslationList: - соответствующие
    обработчики событий #getContents каждой панели."
    (self paneNamed: 'englishPane') update.
    (self paneNamed: 'translationPane') update
```

Перечислим обработчики событий, упомянутые в методе open.

- *Обработчик события #getContents панели 'englishPane'*

```
updateEnglishList: aPane
  "Обработчик события #getContents списковой панели с
  именем 'englishPane'. Отображает в панели текущий список
  английских слов и выбирает в ней слово из списка,
  соответствующее значению переменной englishSelection."
  aPane contents: translation keys asSortedCollection;
        selection: englishSelection
```

- *Обработчик события #getContents панели 'translationPane'*

```
updateTranslationList: aPane
  "Обработчик события #getContents списковой панели
  с именем 'translationPane'. Отображает в панели текущий
  список русских слов и выбирает в ней слово из списка,
  соответствующее значению переменной englishSelection."
  aPane contents: translation values asSortedCollection;
       selection: (translation at:  englishSelection ifAbsent: [nil])
```

- *Обработчик события #select панели 'englishPane'*

```
selectEnglishItem: aPane
  "Обработчик события #select списковой панели с именем
  'englishPane'. Когда в панели английских слов выбран некоторый
  элемент, он запоминается в переменной englishSelection. Затем
  панели трансляции посылают модифицирующее сообщение, чтобы
  определить соответствующий перевод и отобразить его, выбирая
  в отображаемом панелью списке."
 englishSelection := aPane selectedItem.
 (self paneNamed: 'translationPane') update
```

- *Обработчик события* **#select** *панели 'translationPane'*

```
selectTranslationItem: aPane
  "Обработчик события #select списковой панели с именем
  'translationPane'. Когда в панели трансляции выбирается элемент,
  в переменной englishSelection запоминается соответствующее слово
  английского языка. Затем панели английских слов посылают
  модифицирующее сообщение, чтобы отобразить её, выбирая в
  отображаемом списке найденный первоисточник."
  englishSelection := translation
          keyAtValue: aPane selectedItem ifAbsent: [nil].
  (self paneNamed: 'englishPane') update
```

- *Обработчик события #clicked кнопки AddWord*

```
clickedAddWord: aPane
  "Обработчик событие #clicked кнопки 'addWordButton'.
  Вызывает диалоговое окно для ввода нового английского слова,
  и если новое слово является допустимым, добавляет его в словарь
  translation и определяет как выбираемое из списка. После чего
  окно приложения модифицируется."
  | response |
  response := Prompter prompt: 'New Word:' default: ''.
  response isNil | (response = ' ') "nil - как отказ от ввода"
          ifTrue: [ ↑self].
  translation at: response put: (translation at: response
                        ifAbsent: [response,'translation']).
  englishSelection := response.
  self update
```

- *Обработчик события #clicked кнопки DelWord*

```
clickedDeleteWord: aPane
  "Обработчик событие #clicked кнопки 'deleteWordButton'.
  Просит пользователя подтвердить удаление элемента, и, если
  подтверждение получено, удаляет выбранный элемент и его
  перевод из словаря translator."
  englishSelection isNil
    ifTrue: [MessageBox message:
          'You must select the English word to be deleted.']
    ifFalse: [(MessageBox confirm: 'Are you sure you want to delete',
                                  englishSelection,'?')
            ifTrue: [translation removeKey: englishSelection.
                    englishSelection := nil.
                    self update]]
```

- *Обработчик события #clicked кнопки Edit Translator*

```
clickedEditTranslation: aPane
  "Обработчик событие #clicked кнопки 'editTranslationButton'.
  Если выбор был сделан, открывает диалоговое окно, предлагая
  пользователю отредактировать существующий перевод."
  | newTranslation |
  englishSelection isNil ifTrue: [↑MessageBox message:
            'You must select the translation to be edited.'].
```

```
(newTranslation := Prompter prompt: 'Edit the translation:'
            default:(translation at: englishSelection)) isNil
    ifTrue: [ ↑self]. "nil - как отказ от ввода"
translation at: englishSelection put: newTranslation.
(self paneNamed:'translationPane') update
```

Добавим ещё метод экземпляра, позволяющий открыть наш переводчик на словаре, который неким образом создан заранее.

```
openOn: aDictionary
  "Установить словарь translation равным
   aDictionary и открыть приложение."
  translation := aDictionary.
  self open
```

Заключительный штрих: чтобы иметь возможность достаточно просто протестировать работу созданного приложения, напишем простой *метод класса*, который позволяет открыть окно приложения и поэкспериментировать с ним.

```
example1
  "Метод класса для LanguageTranslator, используемый
   при тестировании работы приложения."
  self new openOn: (Dictionary new at: 'red' put:'красный';
                                  at: 'green' put: 'зелёный';
                                  at: 'blue' put: 'синий';
                                  yourself)
```

Теперь для начала работы достаточно вычислить выражение

```
LanguageTranslator example1.
```

# 3 Заключительные замечания

## 3.1 Класс ViewManager

Подводя итог, сделаем некоторые выводы. Прежде всего отметим, что подклассы ViewManager выполняют пять главных функций:

*Помнят текущее состояние окна*. Для этого обычно используются переменные экземпляра приложения.

*Создают панели*. Создание прикладного окна обычно выполняется при посылке сообщения openOn: или open новому экземпляру подкласса класса ViewManager. Выбор между openOn: и open в значительной степени зависит от того,

необходим ли методу, открывающему окно, параметр. Метод, открывающий окно, инициализирует метку окна и создаёт панели окна, обеспечивая для каждого панели следующие элементы:

- метод для заполнения панели её содержимым (обработчик события **#getContents**), по этому методу всегда можно идентифицировать панель (определить её имя), чтобы использовать её в сообщении **changed:,** когда приложение должно модифицировать панель;
- метод для построения меню каждой конкретной панели;
- выражение для построения или вычисления рамки области, занимаемой панелью в окне приложения;
- один или более методов, которые нужно выполнять тогда, когда изменения в панели воздействуют на все приложение;
- владельца панели, обычно это сам экземпляр создаваемого подкласса класса **ViewManager.**

*Обеспечивают содержимое панелей.* Приложение должно предусмотреть для каждой панели метод, который определяет содержимое панели.

*Выполняют связь и синхронизацию.* Когда делается выбор или изменяются содержащиеся в панели данные, это может иметь локальные или глобальные последствия. Глобальные воздействуют и на само приложение, и на другие панели. Все остальное — локальные последствия. Если, например, редактируется текст в текстовой панели окна, такие изменения локальны, поскольку не затрагивают другие панели и само приложение. Однако, когда эти изменения сохраняются, текст должен быть или откомпилирован в выбранном классе или сохранён в файле, и зарегистрирован в файле регистрации изменений. Это может сделать только само приложением, так что последствия сохранения текста уже глобальны.

Определение **when:perform:** в методе, открывающем окно приложения, обеспечивает каждую панель сообщением, которое должно посылаться, когда происходят глобальные последствия. Создавая метод для **perform:,** надо помнить, что его единственным параметром всегда будет сама панель. Когда глобальные последствия требуют модификаций других панелей, для передачи сообщений об изменениях соответствующим панелям приложения, могут использоваться определённые в классе **ViewManager** методы **changed:, changed:with:, changed:with:with:.** Когда приложение получает подобное сообщение, по его первой части (**changed: #message**), оно пытается определить панель, содержимое которой надо обновить. Это та панель, для которой **#message** выступает как второй параметр со-

общения when: #getContents perform: #message, посылаемого панели во время выполнения открывающего метода.

*Определяют меню для панелей.* Когда во время создания панели ей посылается сообщение when: #getMenu perform: #..., в приложении должен быть определён метод с тем же самым именем, как у второго параметра. Этот метод устанавливает меню панели, создавая экземпляр класса Menu (Меню), содержащий желаемые опции. Обратите внимание, что метод, который устанавливает меню панели, всегда должен создавать новый экземпляр класса Menu. Меню будут работать не правильно, если, например, метод создаёт один экземпляр класса Menu, сохраняет его в переменной класса или в глобальной переменной, а затем пытается его использовать для того, чтобы определить меню для нескольких различных панелей.

## 3.2 Класс SubPane

Обратимся к подклассам касса SubPane. Каждый экземпляр класса SubPane обладает следующими характеристиками:

- Имеет владельца, обычно подкласс класса ViewManager.
- Имеет набор событий, о которых сообщает владельцу. Если владельцу надо сообщить о специфическом событии, владелец посылает панели сообщение when:perform:, обеспечивая событие (экземпляр класса Symbol (Символ)) в качестве первого параметра, а имя метода (экземпляр класса Symbol) в качестве второго параметра. Когда происходит событие, панель посылает второй параметр как сообщение владельцу панели, передавая саму панель как единственный аргумент метода.
- Сообщает о событии #getContents, которое является запросом к владельцу, установить содержимое панели. Дополнительно, имя метода для события #getContents служит для определения панели при посылке сообщений с changed:.
- Для многих панелей характерно событие #getMenu, которое является запросом к владельцу установить меню панели.
- Каждый тип панелей имеет специфический набор событий и реализует метод класса с селектором supportedEvents, возвращающий весь набор событий, о которых сообщает панель.
- Понимает сообщения framingBlock: и framingRatio:, которые определяют в окне приложения занимаемую панелью область.

- Понимает модифицирующие сообщения update:, update:with:, update:with:with:, которые, в свою очередь, используются при посылке владельцу панели сообщений changed:, changed:with:, changed:with:with:, соответственно.
- Понимает сообщение, определяющее стиль отображения панели (style:), который задаёт внешний вид панели и определяет необходимы ли в ней полосы прокрутки.

Каждый подкласс класса SubPane обрабатывает свои события, свои общие сообщения и сообщения модификации присущим ему способом, имеет дополнительные уникальные характеристики. Перечислим некоторые основные подклассы класса SubPane с кратким описанием особенностей каждого из них.

**Класс TextPane.** Метод contents: для экземпляра класса TextPane требует в качестве параметра строку. Сообщает о событии #save, когда пользователь пытается сохранить содержимое панели.

**Класс GraphPane.** Событие #getContents для экземпляра класса GraphPane предполагает, что владелец создаёт (рисует) содержимое панели, используя экземпляр класса GraphicsTool, связанный с данной панелью, который доступен при посылке панели сообщения pen. Экземпляр GraphPane сообщает о многих событиях, по одному для каждого события мыши. Текущая позиция мыши в панели может быть получена путём посылки панели сообщения mouseLocation.

**Класс AnimationPane.** Метод contents: для экземпляра класса AnimationPane требует в качестве параметра набор из экземпляров класса AnimatedObject. Чтобы "оживить" объекты, надо посылать сообщения самим объектам анимации, а не анимационной панели.

**Класс ControlPane.** Класс ControlPane — абстрактный класс с подклассами, которые реализуют средства управления, поддерживаемые системой Windows.

**Класс ListBox.** Метод contents: для экземпляра ListBox в качестве параметра требует экземпляра класса IndexedCollection, String или Symbol. Когда в панели выбирается элемент, сообщается о событии #select. Метод selection возвращает индекс выбранного элемента, а метод selectedItem возвращает строку или символ для выбранного элемента.

**Класс GroupPane.** Первичная цель экземпляра класса GroupPane — корректно хранить в себе экземпляры других подклассов класса SubPane, представляя их как единое целое.

**Классы Button и RadioButton.** Метод contents: для экземпляра Button или RadioButton требует, чтобы его параметром был экземпляр класса String — метка кнопки. Метод selection: требует в качестве параметра логического объекта. О событии #clicked сообщается всякий раз, когда пользователь нажимает «на» кнопку. Экземпляры RadioButton, которые находятся внутри объекта GroupPane не должны иметь явного владельца, и даже метода для события #clicked. Сам экземпляр GroupPane имеет владельца и сообщает владельцу о нажатиях на расположенные внутри него кнопки.

**Класс Prompter.** Класс Prompter (Подсказчик) реализует простой механизм, способный задать вопрос и добиться ответа на него. Его экземпляр — модальное диалоговое окно, которое реализует такой тип взаимодействия, который обязательно требует ответа. Диалоговое окно отображает подсказку в форме вопроса, и предоставляет текстовую панель для редактирования ответа.

Чтобы открыть подсказчик, можно послать классу Prompter одно из следующих двух сообщений:

```
Prompter prompt: question default: answer
Prompter prompt: question defaultExpression: answer
```

в котором каждый параметр — строка. После того, как окно откроется, строка answer будет отображаться в текстовой панели. Первое сообщение возвращает введённую пользователем приложения строку, в то время как второе сообщение возвращает объект, который является результатом вычисления введённого ответа. Если ответ не вводится, оба сообщения в качестве ответа возвращают nil. В любом случае при закрытии подсказчика, управление потоком данных передаётся объекту, вызвавшему его.

**Класс MessageBox.** Класс MessageBox (ОкноСообщения) позволяет получить от пользователя быстрый ответ, типа «да/нет». Чтобы открыть окно сообщения, можно послать классу MessageBox одно из следующих двух сообщений:

```
MessageBox confirm: aString
MessageBox message: aString
```

Оба отображают окно сообщения с строкой aString в качестве заголовка. Но первое позволит выбрать либо “Yes” либо “No”, возвращая соответствующий логи-

ческий объект, а второе — всегда возвращает **false**, после нажатия на единственную кнопку “OK”.

# Список литературы


[1] Goldberg A., Robson D., *Smalltalk Express. The language.* — Addison-Wesley Publishing Company, 1988.

[2] Lalonde W., Pugh J., **Smalltalk/V**: *Practice and Experience*, Prentice-Hall, 1994

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

[4] Буч Г. *Объектно-ориентированное проектирование с примерами применения.* — М.: Конкорд, 1992.

[5] *Смолток. Объектно-ориентированная система программирования. Руководство пользователя, часть 1, 2, 3* — М.: Ин-т проблем информатики РАН, 1995

[6] Иванов А., Кремер Ю., *Язык **Smalltalk**: концепция объектно-ориентированного программирования* // КомпьютерПресс — 1992, № 4 — С. 21–31

[7] Фути К., Судзуки Н., *Языки программирования и схемотехника СБИС*: пер. с япон. — М.: «Мир», 1988

[8] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык **Smalltalk**. Общие концепции и синтаксис.* — Ростов-на-Дону: УПЛ РГУ, 1995

[9] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык **Smalltalk**. Интерфейс пользователя и среда программирования.* — Ростов-на-Дону: УПЛ РГУ, 1995

[10] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программмирование и язык **Smalltalk**. Протокол поддержки всех объектов системы.* — Ростов-на-Дону: УПЛ РГУ, 1997

[11] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык **Smalltalk**. Ядро графики: классы* **Point, Rectangle, Form**. — Ростов-на-Дону: УПЛ РГУ, 1997

[12] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык Smalltalk. Графика в Smalltalk/V for Windows.* — Ростов-на-Дону: УПЛ РГУ, 1997

# Содержание