Ю. А. Кирютенко

Объектно-ориентированное программирование. Среда VisualWorks

Кирютенко Ю. А.

Объектно-ориентированное программирование: Среда VisualWorks — Ростов-на-Дону: «ЮФУ», 2008. — 163 с.: ил.

Учебник по объектно-ориентированному программированию, языку программирования Smalltalk и их реализации в среде VisualWorks. В книге рассматривается многопользовательская среда разработки приложений VisualWorks 7.4.1. Основу книги составили материалы лекций, читавшихся автором в Южном федеральном университете.

Для студентов и преподавателей вузов, практикующих программистов и всех желающих изучить язык Smalltalk и объектно-ориентированное программирование и их реализации.

Учебник написан при поддержке гранта ЮФУ 2007.

© Ю. А. Кирютенко, 2008

© ЮФУ, оформление, 2008

Предисловие

История VisualWorks началась в 1989 году со среды *ObjectWorks* и успешно завершилась первым выпуском VisualWorks в 1991 году. Серия продуктов VisualWorks обладает большими возможностями, а после слияния ParcPlace и Digitalk стала одним из основных продуктов сначала объединенной компании ObjectShare, а затем компании Cincom

Сегодня VisualWorks — одна из самых распространённых смолтоковских сред. Её библиотека классов восходит к библиотеке Smalltalk-80, удивительно стабильна, а инструменты, связанные с разработкой приложений, поддержкой баз данных, жизненного цикла программ, и прочие утилиты, не уступают аналогичным инструментам среды IBM VisualAge.

VisualWorks является комплексом приложений и состоит из ядра (собственно VisualWorks) и множества подключаемых дополнений. Среди них

VisualWave — содержит инструменты разработки интерактивных web-приложений;

Distributed Smalltalk — содержит классы и средства разработки распределенных приложений;

DLL & C Connect — обеспечивает доступ из VisualWorks к функциям разделяемых библиотек и позволяет создавать новые примитивы;

Database Connect — обеспечивает интерфейсы к СУБД IBM DB2 и Oracle; в архивах общедоступного кода есть пакеты, обеспечивающие доступ к базам данных через ODBC, интерфейс к СУБД MySQL и непосредственную работу с файлами dBase;

GemBuilder for Smalltalk — клиентская программа, поддерживающая в среде VisualWorks и VisualAge работу с объектно—ориентированной базой данных GemStone.

COM Connect — поддерживает работу с OLE-совместимыми приложениями.

В 2006 году компанией Cincom выпущена версия VisualWorks 7.4.1, которая работает на платформах Apple PowerMac-MacOS 8.x, HP/UX, IBM AIX, SGI Irix, Sun SPARC-Solaris, Compaq Alpha-Digital Unix, Linux-i386

и Windows 9x/NT/XP. Её некомерческую версию для Windows 95/NT, Linux i386 и Power Macintosh можно найти на сайтах

http://www.cincom.com/ и http://www.redhat.com/

Версия 7.4.1 представляет существенно измененный Смолток, поддерживающий пространства имён, константные объекты и ограничения доступа. Но сохранена обычная для VisualWorks организация GUI, и сделано все, чтобы ядро (как описано в [1], [3]) осталось неизменным.

Некоммерческая версия является полнофункциональной системой, отличающаяся от обычной только тем, что её можно использовать только для знакомства и обучения, но нельзя использовать для создания коммерческих приложений. Кроме того, не включена часть пакетов, в частности те, которые поставляются на условиях сублицензирования. В остальном (и прежде всего в формате образа) система полностью совместима с коммерческой версией.

В поставку среды также входят разнообразные приложения и утилиты из общедоступных смолтоковских архивов, адаптированные под данную версию VisualWorks.

В учебнике описывается среда VisualWorks 7.4.1, но часть материала справедлива и для более ранних версий. Предполагается, что читатель знаком с основными идеями и синтаксисом языка Смолток в рамках материала, изложенного, например, в [15, глава 1–2, 4–9]. Такой читатель может пропустить первую главу и начинать с главы 2. Для остальных в первой главе кратко и с небольшими изменениями, отмечающими особенности VisualWorks, повторяется материал глав 1, 2 из [15].

Благодарности

Благодарю профессоров Я.М. Ерусалимского — декана факультета математики, механики и компьютерных наук Южного федерального университета, и А.В. Абанина — заведующего кафедрой математического анализа этого же факультета, за поддержку и помощь. Благодарю специализировавшихся у меня студентов за их явную и неявную помощь в создании учебника.

Ю.А. Кирютенко.

Глава 1

Структура классического Смолтока

Классическим Смолтоком мы называем реализации, ориентированные на разработку приложения одним программистом (например, Smalltalk-80, VisualWorks-3.0, Visual Smalltalk, Smalltalk Express, VisualAge for Smalltalk 4.0, ...). Данная глава кратко и с небольшими изменениями повторяет материал глав 1, 2 из [15]. Те, кто читал эту или другие книги по смолтоковским системам (например, книги [2], [3], [4], [8]), работал в смолтоковских средах, могут эту главу пропустить, перейти к следующим главам, в которых описываются существенные новшества среды VisualWorks 7.4.1[1], и использовать первую главу только как справочник по объектно-ориентированной терминологии.

1.1. Основные определения и термины

В основе объектно-ориентированного программирования лежит подход, согласно которому структура программы следует структуре прикладной предметной области и опирается на различимые сущности — *объекты*. Каждый объект обладает индивидуальностью, проявляемой в их состоянии и поведении, а взаимодействия между объектами происходит посредством пересылки *сообщений*.

Совокупность состояния и поведения объектов позволяет различать объекты и формирует *индивидуальность* объекта. *Состояние* объекта определяется всеми его возможными свойствами или, иначе, атрибутами (как правило, статическими), и текущими значениями каждого из этих свойств (обычно динамическими, изменяющимися во время работы программы). Значения атрибутов могут вычисляться или запрашиваться объектом у других программ или аппаратных систем.

[1] Всюду далее рассматривается некомерческая версия VisualWorks 7.4.1.

Поведение характеризует то, как объект отвечает на посылаемые ему *сообщения*: или изменением своего состояния и/или воздействием на другие объекты с помощью передачи сообщений. Совокупность всех сообщений, понимаемых объектом, называется его *интерфейсом*. Состояние объекта доступно только через его сообщения.

Поведение объекта реализуется в виде подпрограмм, вызываемых объектом в ответ на полученное им сообщение. Такие подпрограммы называются *методами*. Разные объекты могут использовать разные методы для ответа на одно и то же сообщение. Объекты должны иметь функционально полные и в то же время максимально простые и стабильные интерфейсы.

В правильно созданном объектно-ориентированном приложении в любой момент можно изменить внутреннюю структуру объекта и реализацию его методов и это не скажется на работе остальных частей приложения. Такое объединение в объекте его структуры и поведения называется *инкапсуляцией* и позволяет скрыть структуру и данные внутри объекта, делая их невидимыми для всех, за исключением методов самого объекта, тем самым снижая сложность использующей объект программы. Инкапсуляция позволяет рассматривать объекты как изолированные «черные ящики», которые знают определенные действия и умеют их выполнять, функционируя независимо друг от друга и скрывая за интерфейсом детали реализации. Поэтому объекты в объектно-ориентированных системах можно рассматривать как минимальные (в смысле самодостаточности) единицы инкапсуляции, позволяющие навести должный порядок среди данных и обрабатывающего эти данные кода.

К объекту может обращаться и программист, и любой объект системы, посылая нужному объекту *сообщение,* которое представляет предписание (просьбу, приказ, требование) выполнить некоторые действия. Если предписание может быть выполнено получившим сообщение объектом (его называют объектом-получателем, или просто получателем), то оно выполняется и, как результат, объекту, пославшему сообщение, может возвращаться некоторый объект (возвращаемый объект). Если по какой-либо причине объект, получивший сообщение, не может его выполнить, он в какой-то форме информирует об этом пославший сообщение объект (или систему времени выполнения). В других терминах: объект-получатель — это *сервер,* предоставляющий обслуживание *клиенту* — объекту-отправителю.

Таким образом, объектно-ориентированное программирование сводится к моделированию некоторого числа объектов, которые для решения поставленной задачи взаимодействуют друг с другом, посылая сообщения, в ответ на которые выполняют действия, описанные в методах, соответствующих сообщениям.

В объектно-ориентированном программировании объекты, имеющие общие свойства и поведение, используют общее определение состояния и поведения, которое осуществляется классами. На класс возлагается обязанность по «хранению» поведения и информации о внутренней структуре объектов. Наследование позволяет разделять между несколькими классами поведение, внутреннюю структуру и, возможно, данные. Другими словами, класс — это шаблон, описывающий объекты определенной структуры и поведения и хранящий информацию, общую для всех таких объектов.

Каждый класс имеет уникальное идентифицирующее его имя. Имена классов в Смолтоке всегда начинаются с прописной буквы, поскольку классы являются объектами, доступными для многих других объектов системы. В классическом Смолтоке — для всех объектов системы. Имена всех таких объектов содержатся в системном словаре с именем Smalltalk, и эти объекты могут вызываться по имени в любое время и в любом месте. В системе VisualWorks это не так. Подробно это будет рассматриваться в главе 3.

Класс через переменные описывает структуру объекта, а через методы — поведение объекта, и определяет механизмы создания «своего» объекта, который представляет собой *экземпляр класса*. Таким образом, методы и переменные определяются в классе, в то время как значения переменных определяются в экземпляре, отражая индивидуальные характеристики объекта. Каждый метод имеет имя, состоящее из селектора сообщения и, возможно, параметров сообщения.

Взаимодействие классов и порядок среди классов достигается с помощью введения механизма *наследования*. По своей сути механизм наследования прост: один класс, называемый в рамках этих отношений *суперклассом*, полностью передает другому классу, который называется его *подклассом,* свою структуру и поведение, то есть все свои переменные и все методы. Что дальше делать с «этим богатством» определяет подкласс: он может добавить в структуру и поведение что-либо свое, что-то не использовать вообще, что-то использовать без изменений, а что-то изменить. Таким образом, класс с помощью подклассов «уточняет» поведение экземпляров, и, как результат, создаваемые объекты становятся более специализированными.

Поскольку поведение объектов определяется методами, в результате переопределений в классах, связанных наследованием, могут возникать (и возникают) методы с одним и тем же именем, но с разными реализациями. Возможность единообразного обращения к объектам (посылки им одноименных сообщений) при сохранении их уникального поведения называется *полиморфизмом*. Классы, расположенные по принципу наследования, начиная с самого общего, базового класса, образуют древовидную структуру,

которая называется *иерархией классов*.

В Смолтоке существует ограничение на построение иерархии классов: у каждого класса может быть только один непосредственный суперкласс. Такое наследование называется *одиночным наследованием*.

Все классы системы являются подклассами класса Object[2], в котором определены структура и поведение, общие для всех объектов системы. Object является базовым классом иерархии и единственным классом, не имеющим суперкласса. Цепочка класс-суперкласс всегда завершается классом Object.

1.2. Метаклассы

В однородном объектно-ориентированном языке программирования, каковым является Смолток, любой объект системы является экземпляром некоторого класса. Следовательно, сами классы выступают как объекты системы. И, подобно тому, как объекты создаются из классов, так и сами классы являются объектами, создаваемыми в соответствии с шаблоном, заключенном в *метаклассе* данного класса. При этом класс — *единственный* экземпляр своего метакласса. В языке Смолток метакласс, в отличие от класса, не имеет имени, и доступ к нему осуществляется посылкой классу сообщения class.

Иерархия метаклассов повторяет иерархию их классов и ее базовым классом является метакласс класса Object.

Каждый метакласс в системе, являясь ее объектом, является экземпляром класса Metaclass, который, являясь шаблоном для метаклассов, описывает общие свойства тех объектов, которые умеют создавать единственный объект-класс.

Метакласс класса Object в системе Смолток является подклассом класса с именем Class,. Таким образом, все метаклассы являются подклассами класса с именем Class, одновременно являясь экземплярами класса Metaclass.

Реализованная в Смолтоке иерархия классов имеет одно очень важное следствие: *цепочки суперклассов для самого класса и для его метакласса разные*. Например, класс Integer в VisualWorks имеет цепочку суперклассов

Integer, Number, ArithmeticValue, Magnitude, Object.

А его метакласс (класс Integer class) имеет цепочку суперклассов

Integer class, Number class, ArithmeticValue class, Magnitude class, Object class, Class, ClassDescription, Behavior, Object.

Итак, в Смолтоке есть два вида объектов: те, которые могут создавать экземпляры, и те, которые не могут этого делать. Классы Class, Metaclass,

[2] Смолтоковский код далее всегда будет печататься рубленым шрифтом.

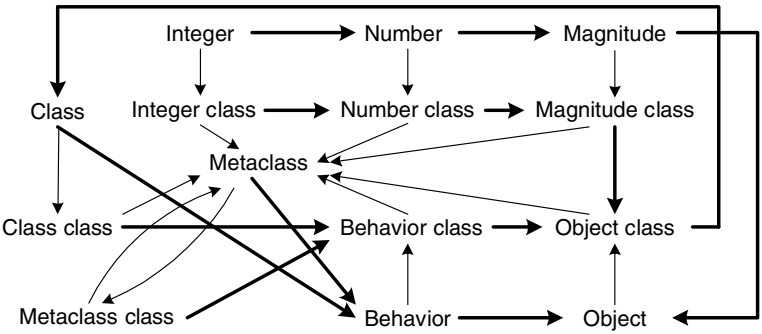


Рис. 1.1: Двойная иерархия класс/метакласс

ClassDescription и Behavior описывают структуру и поведение тех объектов, которые могут создавать экземпляры.

Из сказанного выше следует, что класс Metaclass занимает в системе особое место, поскольку порождает как свои экземпляры все метаклассы системы, в том числе и собственный метакласс Metaclass class, и, с другой стороны, сам является экземпляром собственного метакласса Metaclass class. В этом месте в системе специально создана «особенность», позволяющая избежать бесконечной (reductio ad infinitum) последовательности метаклассов, «отождествляющая» классы Metaclass и Metaclass class (см. рисунок 1.1).

В силу того, что иерархия метаклассов автоматически определяется по иерархии классов, иметь дело приходится в основном с иерархией классов. Метаклассы для классов система создает и устанавливает в иерархию автоматически.

Как итог, сформулируем те основные правила, которым подчиняется иерархия классов системы Смолток:

1) Каждый класс, кроме самого класса Object, не имеющего суперкласса, имеет один непосредственный суперкласс и, в конечном счете, каждый класс является подклассом класса Object.
2) Каждый класс является единственным экземпляром своего метакласса.
3) Иерархия метаклассов подобна иерархии классов.
4) Класс Object class, и все метаклассы являются подклассами класса Class.
5) Каждый метакласс является экземпляром класса MetaClass.

1.3. Посылка сообщений

В Смолтоке единственным способом управления объектами является посылка объектам сообщений. Запись по синтаксическим правилам языка Смолток операции посылки объекту сообщения (последовательности сообщений) будем называть выражением посылки сообщения или просто *выражением*. Каждое выражение отделяется от следующего выражения точкой. В VisualWorks если выражение единственное или последнее, точку можно не ставить (граница выражения ясна и так). Процесс реакции объекта на полученное им сообщение, включающий поиск и выполнение метода, а также возврат некоторого объекта, будем называть *выполнением выражения*.

Выражение сообщения состоит их трех компонентов: приемника сообщения, селектора сообщения и нескольких (если они нужны) вспомогательных объектов — аргументов. Термин *сообщение* технически относится только к имени метода (состоящего из селектора сообщения и аргументов, если они есть), в то время как термин *выражение посылки сообщения* включает ещё и приемник сообщения.

Метод (все равно, класса или экземпляра) состоит из имени метода и тела метода. Имя метода состоит из селектора сообщения) и формальных параметров, если они нужны. Например, один из методов класса в классе Association имеет имя key: aKey value: anObject, который состоит из селектора сообщения key:value: и двух формальных параметров aKey и anObject. Все, что после него, — тело метода, составляющее программу, выполняемую объектом-получателем в ответ на сообщение с селектором key:value:. В любом месте программы в нее могут добавляться комментарии — строки, заключенные в двойные кавычки. Традиционно тело метода начинают с комментария, поясняющего назначение и особенности применения метода.

Методы экземпляра или методы класса выполняются в ответ на посылку сообщений, соответственно, экземплярам класса, или классу. Получив сообщение, получатель ищет метод с соответствующим сообщению селектором (шаблоном), начиная поиск обычно со своего класса. Если объект — класс, то метод ищется среди методов класса, а если объект — экземпляр класса, то — среди методов экземпляра класса. Если метод с соответствующим шаблоном находится в классе получателя, то он и выполняется.

Если метода с нужным шаблоном в классе нет, то метод ищется по цепочке суперклассов для класса объекта-получателя, начиная с его ближайшего суперкласса. Если нужного метода нет в первом суперклассе, то поиск продолжается в следующем по иерархии суперклассов. Если нужный метод не будет найден вплоть до класса Object, то система отобразит сообщение об ошибке времени выполнения Message not understood:. Соответствующий

метод определен в классе Object.

После выполнения сообщения отправителю сообщения возвращается некоторый возвращаемый объект. Есть три способа указать на объект, который будет возвращаться методом:

- по умолчанию, тогда отправителю возвращается приемник сообщения; (self).
- оператором возврата объекта (^), предшествующим имени переменной, хранящей этот объект, тогда возвращается значение переменной;
- оператором возврата объекта (^), предшествующим выражению сообщения, тогда возвращается результат выполнения сообщения.

Когда в теле метода встречается оператор возврата объекта, то объект выражения возвращается как результат выполнения метода, а дальнейшее выполнение метода прекращается.

Таким образом, посылка сообщения включает в себя:

- определение объекта, которому посылается сообщение;
- определение, если они нужны, объектов-аргументов сообщения;
- определение нужного метода, который ищется в классе или суперклассах;
- выполнение метода;
- возвращение некоторого объекта.

1.4. Определение объектов

1.4.1. Литеральное определение объектов

Объекты языка, которые можно определить, просто записывая их, называются *литеральными объектами (литералами)*. Таким образом, литерал — смолтоковское выражение, которое всегда ссылается на один и тот же объект, на самого себя. Все литералы в VisualWorks не могут изменяться!

Особенность литеральных объектов состоит в том что для создания их достаточно «написать», то есть представить в литеральной форме[3].

Символы

Чтобы что-то написать, надо иметь алфавит. Алфавит системы Смолток ничем не отличается от алфавита многих других языков программирования и состоит из букв, цифр, символов пунктуации и так далее. Но в языке

[3] С точки зрения организации системы это почти соответствует посылке сообщения соответствующему классу, в результате чего и порождается новый экземпляр класса.

Смолток это, как и все в системе, объекты — экземпляры класса Character, которые мы будем называть символами.

```
Object ()
  Magnitude ()
    Character()
```

Экземпляр класса Character не надо создавать, он всегда присутствует в системе после её запуска, а чтобы им воспользоваться, его надо только записать тем или иным образом. Печатаемый объект-символ можно задать в виде символа $ и последующего литерала. Например, $A — экземпляр класса Character, представляющий латинскую заглавную букву A, $, — экземпляр класса Character, представляющий запятую, $ж — экземпляр класса Character, представляющий строчную русскую букву ж, $7 — экземпляр класса Character, представляющий цифру 7.

Экземпляры класса Character еще называются символьными константами, поскольку структуру данного объекта изменить нельзя.

В VisualWorks символы идентифицируются неотрицательным целым числом. Коды 0—127 определяют символы согласно стандарту ASCII. Символы уникальны в том смысле, что все символы с одним и тем же кодом идентичны (==).

Для символьных кодов между 0 и 65535 (16rFFFF), используется Unicode Character Code Standard — 16-битный стандарт кодирования символов, позволяющий представлять алфавиты всех существующих в мире языков. Символы с кодами между 0 и 255 совпадают со стандартом ISO 8859—1. В настоящее время, отображения для символов с кодом, больше чем 65535, не определены, и такие символы не поддерживаются. Буквы русского алфавита от А до я (кроме букв Ё,ё) имеют код от 1040 до 1103. Ё имеет код 1025, ё — 1105.

Те символы, которые нельзя ввести с клавиатуры (непечатаемые символы), типа del, cr, esc, tab можно получить, посылая соответствующие сообщения классу Character. В дополнение к наследуемым из класса Magnitude методам сравнения, символы можно сравнивать по их кодам, определять, является ли символ алфавитно-цифровым, является ли буква гласной или согласной, строчной или прописной, можно переводит строчную букву в прописную и наоборот, определять код символа, по коду определять символ.

Строки

Строка — последовательность символов, заключенная в одинарные кавычки. Как и для символов, чтобы создать новую строку, достаточно «написать» нужную строку. Например, 'Bold', 'seed', 'Hello word', 'Привет мир!'.

Когда одинарная кавычка является частью строки, она должна удваиваться, чтобы синтаксический анализатор среды, понимал, что это не конец строки. Например, 'don''t'.

Любая строка является экземпляром некоторого конкретного подкласса абстрактного класса String (Строка).

```
Object
  Collection
    Bag
    SequenceableCollection
      ArrayedCollection
        Array
        CharacterArray
          String
            ByteEncodedString
              ByteString
            Symbol
              ByteSymbol
              FourByteSymbol
              TwoByteSymbol
            TwoByteString
```

В большинстве приложений, строки являются экземплярами класса ByteString, если используются только символы с однобайтовыми кодами ASCII. Если в строке используется хотя бы один символ, требующий для представления двухбайтового кода, строка реализуется как экземпляр класса TwoByteString. Поскольку разные операционные системы кодируют однобайтовые символы по-разному, различие между такими строками определяется абстрактным классом ByteEncodedString и его конкретными подклассами, связанными с конкретными платформами: ISO8859L1String для X Window и MSWindows, MacString, OS2String, MSCP1252String для кодовой таблицы 1252 Microsoft, а так же общий подкласс ByteString. Однако программисту беспокоиться о внутреннем представлении строк не стоит, система производит все преобразования между строковыми объектами автоматически.

Большая часть полезного протокола для строк определена в классе CharacterArray и включает в себя доступ (замену и поиск), преобразование (в число, текст, символ и другие родственные объекты), копирование, сравнение (алфавитные отношения), и отображение на экран. Так как строка — набор, она также наследует богатый протокол из всех суперклассов наборов. В частности, запятая часто используется для объединения двух строк (string3 := string1, string2). Стоит отметить, что такая операция объединения строк неэффективна. Когда важна скорость, следует связывать строки, используя потоки.

Наиболее полезные методы, определенные в самом классе String, включают сравнение и преобразование строк в другие строковые объекты.

Системные имена

Системное имя — уникальная последовательность символов одного из следующих видов:

- последовательность букв и цифр, начинающаяся с буквы (идентификатор);
- последовательность их одного или более ключевых слов (идентификаторов, в конце которых стоит двоеточие);
- последовательность из одного или двух специальных символов

$$+ \; - \; / \; \backslash \; * \; = \; < \; = \; > \; \sim \; @ \; | \; \% \; \& \; ? \; ! \; ,$$

Каждое системное имя — экземпляр одного из подклассов класса Symbol (СистемноеИмя). Когда в программе записывается системное имя, перед ним ставится символ #. Например, #Red, #Green, #at:put:, #size, #,. Использованная выше фраза «уникальная последовательность», означает, что в системе не могут существовать два системных имени, которые имеют одну и ту же последовательность символов. Система следит за уникальностью системных имен. Системные имена используются для именования уникальных объектов системы.

Экземпляр класса Symbol часто вводится в систему и выводится на печать без префикса #. Однако при работе с системным именем как таковым или при определении нового имени обязательно надо явно указывать префикс #.

Системное имя не должно включать символов типа пробела, табуляции, перевода строки или #, если строка после # не является литеральной строкой. Например, #ab cd не является единым системным именем, а #'ab cd', #'a#b' — являются, причём апострофы не его часть. Поскольку системные имена уникальны, для их сравнение используется эквивалентность, которая является более быстрой операцией, та как она требует только сравнения указателей. Строки обычно сравниваются операцией равенством, которое может потребовать много времени.

Числа

Все классы чисел в VisualWorks определены в иерархии классов Magnitude – ArithmeticValue. Структура этой части иерархии следующая:

```
Object ()
  Magnitude ()
    ArithmeticValue ()
      Number ()
```

```
FixedPoint ('numerator' 'denominator' 'scale')
Fraction ('numerator' 'denominator')
Integer ()
  LargeInteger ()
    LargeNegativeInteger ()
    LargePositiveInteger ()
  SmallInteger ()
LimitedPrecisionReal ()
  Double ()
  Float ()
  SmallDouble ()
```

В смолтоковской среде могут использоваться целые числа (экземпляры подклассов класса Integer), рациональные дроби (отношения двух целых — экземпляры класса Fraction), рациональные числа в форме числа с плавающей точкой (экземпляры класса Float), рациональные числа в форме числа с фиксированной точкой (экземпляры класса FixedPoint). В приведенной выше иерархии против классов FixedPoint и Fraction указаны определяемые ими переменные экземпляра. Из них наиболее важными являются классы Fraction, LargeNegativeInteger, LargePositiveInteger, SmallInteger, Point, а классы FixedPoint, Double используются редко.

Экземпляры всех подклассов класса Number неизменяемы, поэтому их ещё называют числовыми константами,.

Целые числа (Integer) — числа без десятичной точки, например, 13, -4570089, 0. В VisualWorks целые числа, непосредственно обрабатываемые компьютерными аппаратными средствами реализованы классом SmallInteger и их максимальные и минимальные значения могут быть получены при выполнении следующих выражений:

```
SmallInteger minVal  → -536870912
SmallInteger maxVal →  536870911
```

Объекты SmallInteger представляются целочисленным значением в двоичном коде, что делает доступ к ним очень эффективным.

Диапазон для чисел SmallInteger далеко не маленький, а название SmallInteger возникло вследствие того, что Смолток реализует другие, по существу неограниченные целые числа, используя два класса «больших целых чисел». Выделяемая под большие числа память ограничивается только памятью вашего компьютера. Размер большого числа, в принципе, неограничен. Но, в отличие от объектов SmallInteger, такие числа не могут обрабатываться одной командой CPU, и требуют специальной обработки, которая и осуществляется классами LargeNegativeInteger и LargePositiveInteger, являющимися подклассами класса LargeInteger. Как пример, класс LargePositiveInteger

позволяет вычислять такие числа как 2^{64}, сумму членов геометрической прогрессии $\sum_{i=1}^{64} 2^i$, 100!:

```
2 raisedToInteger: 64      →  18446744073709551616

(1 to: 64)
   inject: 0
   into: [:s :i | s + (2 raisedToInteger: i)]
                           →  36893488147419103230

100 factorial              →
      933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

Эти результаты нелегко получить в других языках программирования. Конечно, цена за эту вычислительную мощь — значительно большее время вычислений, чем при работе с объектами SmallInteger. Но в Смолтоке не надо следить за тем, с какими, «маленькими» или «большими» целыми числами вы работаете, не надо следить за числами во время выполнения арифметических операций, поскольку Смолток выполняет преобразование между этими тремя классами целых чисел автоматически, и можно даже не знать, что эти три класса существуют.

Числа с плавающей точкой — числа с десятичной точкой (например, 3.14, -7890.123, 0.00341, 0.0, 1.33185e17 = $1.33185 \cdot 10^{17}$). Подобно другим языкам программирования, Смолток имеет два представления для чисел с плавающей точкой, реализуемых двумя классами Float (одинарной точности) и Double (удвоенной точности). Различие между ними в том, сколько байтов они используют и какую точность и в каком диапазоне они обеспечивают. В отличие от автоматического преобразования между SmallInteger и LargeInteger, преобразование между числами с одинарной точностью и удвоенной точностью не происходит автоматически и требует конверсионных сообщений asFloat и asDouble.

Экземпляры класса Float представляют числа с плавающей точкой в диапазоне $\pm 10^{38}$ с точностью вычислений приблизительно в $8 - 9$ цифр. В литеральном представлении формат числа не допускает никаких пробелов, содержит e, как основание степени равное десяти, цифры с обеих сторон от e и десятичную точку. Вот несколько примеров чисел с плавающей точкой класса Float: 8.0e 13.3e 0.3e 2.5e6 1.27e-30 1.27e-31 -12.987654e12.

Экземпляры класса Double представляют числа с плавающей точкой в 64-разрядном IEEE-формате в диапазоне между $\pm 10^{307}$ с точностью вычислений приблизительно в $14 - 15$ цифр. В литеральном представлении

формат числа не допускает никаких пробелов, содержит d, как основание степени равное десяти, цифры с обеих сторон от d и десятичную точку: 8.0d 13.3d 0.3d 2.5d6 1.27d-30 1.27d-31 -12.987654d12

Если в литеральном представлении числа с плавающей точкой не указан спецификатор e или d, система по умолчанию генерирует экземпляр класса Float.

Экземпляры класса SmallDouble представляют числа с плавающей точкой в 64-разрядном IEEE-формате. Ненулевые числа SmallDoubles имеют экспоненту от −127 до 127, не создаются непосредственно, а создаются как результат системных примитивов.

Важная особенность представления чисел с плавающей точкой состоит в том, что отображаемое значение не точно соответствует тому, что храниться в памяти компьютера из-за несовместимости внутреннего двоичного представления числа и его десятичного представления на экране. Число цифр, отображаемых на экране, управляется значением, возвращаемым в ответ на сообщение defaultNumberOfDigits. Например,

2.3e-2 defaultNumberOfDigits → 6
2.3d-2 defaultNumberOfDigits → 14

Таким образом числа с плавающей точкой отображаются с меньшей точностью, по отношению к их полной точности. Но независимо от того, сколько цифр печатается, внутреннее представление чисел не меняется и не меняется точность выполнения арифметических операций.

Рациональные дроби (Fraction**)** — экземпляры класса Fraction, представляют рациональные дроби и литерально записываются в виде числитель/знаменатель, например, −3/5 14/27. Каждая дробь внутренне представляется двумя целыми числами — целочисленным числителем и натуральным знаменателем, поэтому не происходит никаких преобразований и потери точности вычислений при выполнении арифметических операций. Арифметические операции с дробями используют известные из математики правила преобразований, основанные на числителях и знаменателях операндов, и создают новую дробь, то есть создают новый экземпляр класса Fraction. Поскольку при работе с дробями требуется больше операций с целыми числами, операции с дробями требуют больше времени.

Числа с фиксированной точкой (FixedPoint**)** — представляют «числа для бизнеса», то есть числа с произвольным числом цифр перед десятичной точкой, но ограниченным числом цифр после десятичной точки. Их иногда называют масштабируемыми десятичными числами — Scaled Decimal. Поэтому при их литеральном представлении используется символ s: 123s, 23.56s. Основная область их использования — представление денежных сумм, которые всегда округляются до сотых, но которые могут иметь пе-

ред десятичной точкой большее цифр, чем, например, числа с плавающей точкой.

Приведем еще несколько примеров литерального представления чисел, одновременно объясняя особенности их представления:

16r2FA1, −16rFF, 8r55425, −2r10011, −23, 36r2AZ7 — примеры разных целых чисел. Число перед 'r' (всегда в десятичной форме) указывает основание системы счисления. Если 'r' перед числом отсутствует, число записано в десятичной системе счисления. Для обозначения недостающих цифр, используются буквы латинского алфавита, поэтому возможное наибольшее основание счисления — 36. Первые два числа записаны в 16-ричной системе счисления, затем числа в восьмеричной, двоичной, десятичной, и 36-ричной системах счисления.

−1/2, 34/55, −8r5366/8r571, 2r1001101/2r1011, 8r34/55 — рациональные числа (дроби) в которых числитель и знаменатель представлены в разных системах счисления. Системы счисления для представления чисел в числителе и знаменателе дроби могут не совпадать.

В Смолтоке большинство математических операций над числами можно выполнять на всех типах чисел с обычными ограничениями, такими как, недопустимость деления на 0, вычисление логарифмов и квадратных корней только для положительных чисел, и так далее. Но следует признать, что Смолток — не идеальный язык для больших математических вычислений (особенно на числах с плавающей точкой), поскольку принцип трактовки всего в виде объектов, замедляет выполнение арифметических операций. Для таких задач следует использовать другие языки программирования (например, Си, Фортран, ассемблер).

Есть в Смолтоке и проблемы с точностью вычислений, особенно при выполнении преобразовании чисел с плавающей точкой. Например, метод asRational преобразует числа в экземпляр класса Fraction (в рациональную дробь). Методы asFloat, asDouble преобразуют числа в число с плавающей точкой соответствующего класса. Вот что может получиться:

```
12345678.1476 asRational asFloat - 12345678.1476 → 0.0
12345678.76 asRational asFloat = 12345678.76 → true.
31.76 asRational asFloat - 31.76 → 1.90735e-6
31.76 asRational asFloat = 31.76 → false.
31.76 asRational asDouble - 31.76 → -1.833446461319d-8
31.76 asRational asDouble - 31.76d → 2.1054736976112d-7
31.76d asRational asDouble - 31.76d → 0.0d
1.75e-1 asDouble - 1.75d-1 → -2.9802322554229d-9
```

Арифметика для чисел с плавающей точкой не точна и может приводить к неожиданным результатам даже в простых ситуациях. Например, напечатаем таблицу логарифмов для всех чисел от 1 до 10 с шагом 0.1. Задача легко решается следующим выражением:

```
Transcript clear.
1 to: 10 by: 0.1 do: [:number |
        Transcript show: number asFloat printString;
                tab;
                show: number log printString;
                cr].
```

К сожалению, код останавливает вычисление на значении 9.9, поскольку накапливается погрешность вычислений от прибавления числа с плавающей точкой 0.1, и log 10.0 уже не считается. Чтобы исправить ситуацию, в качестве шага приращения используем рациональную дробь и получим желаемый результат:

```
Transcript clear.
1 to: 10 by: 1/10 do:[:number |
        Transcript nextPutAll: number asFloat printString;
                tab;
                nextPutAll: number log printString;
                cr].
```

Массивы

Еще один объект, который можно задавать литерально — массив. Массив — экземпляр класса Array, содержащий упорядоченный набор объектов, каждому из которых, в соответствии с занимаемым им местом, приписывается индекс. Например, запись #($A $B $C) порождает массив из трех объектов, на первом месте стоит символ $A, на втором — символ $B, на третьем — символ $C.

Элементы массива не обязаны быть экземплярами одного класса. Они могут быть любыми объектами. Запись #(1 'two' $D Green) порождает экземпляр класса Array с четырьмя объектами, каждый из которых литерал. Последний объект в массиве — системное имя, но поскольку символ # уже использован перед определением самого массива, перед системным именем его можно опустить.

Если литеральный массив является элементом в другом массиве, он должен предваряться литерой #, как в следующем примере:

```
#(1586.01 $a 'sales tax' #January #($x $y $z)).
```

Часто массив используется в тех ситуациях, когда надо выполнить несколько альтернативных сообщений на одном наборе данных. Например, напе-

чатаем в окне Transcript значения функций sin, cos, ln, exp, $\sqrt{}$ в точках из промежутка от 1 до 10 с шагом 1/2. Рассмотрим массив функций #(#sin #cos #ln #exp #sqrt), который будем использовать при перечислении и воспользуемся сообщением perform:, которое требует аргумента в виде системного имени, представляющего имя метода, и выполняет его. Конечно, не стоит использовать perform:, если точно известно выполняемое сообщение, но если сообщение заранее на известно, как в примере, perform: — единственный способ решить такую задачу:

```
1 to: 10 by: 1/2 do: [:x | Transcript cr; show: x printString.
#(#sin #cos #ln #exp #sqrt) do:
        [:message | | value |
            value := x perform: message.
            Transcript tab; show: value printString]]
```

Литерально можно создавать экземпляры класса ByteArray — массивы байтов. Эти объекты, в отличие от массивов, могут хранить только целые числа от 0 до 255, которые отделяются друг от друга пробелами.

```
Object
  Collection
    SequenceableCollection
      ArrayedCollection
        IntegerArray
          ByteArray
            BinaryStorageBytes
            BOSSBytes
          DwordArray
          WordArray
```

Литерально массив байтов записывается в квадратных скобках и ему предшествует символ #. Например, #[255 0 0 7]. Массивы байтов обычно получают из строк, посылая им сообщение asByteArray:

```
'assa' asByteArray  → #[97 115 115 97].
```

Строку (экземпляр класса ByteString) можно получить, посылая массиву байтов сообщение asByteString.

Экземпляры класса WordArray могут хранить только целые числа от 0 до 65535, а экземпляры класса DwordArray — от 0 до $2^{32} - 1$. По своему внешнему виду они ничем не отличаются от обычных массивов, но не имеют литерального задания. Их экземпляры надо создавать посредством посылки классу сообщений создания экземпляра.

1.4.2. Определение объектов посылкой сообщения

Литеральных объектов очень мало, а классов в системе очень много, так что первый способ определения объектов не универсален и доступен только для наиболее простых и часто используемых объектов. Все остальные объекты создаются посредством посылки классу сообщения. Да и экземпляры классов String, Symbol, Array, asByteArray можно создавать не только литерально.

Метод, создающий новый экземпляр класса, определяется или самим классом, как, например, метод с именем key:value: в классе Association, или наследуется классом у одного из его суперклассов.

Например, чтобы создать экземпляр класса Date, который представляет сегодняшнюю дату, следует послать классу Date сообщение today. Аналогично, чтобы создать экземпляр класса Time, представляющий текущее время, требуется послать классу Time сообщение now. Будет возвращено время (по часам компьютера) посылки сообщения.

Существуют универсальные сообщения создания экземпляров. Это сообщения new и new:. Первое сообщение просто создает экземпляр класса по содержащемуся в классе описанию, ничего в экземпляре конкретно не определяя. А второе используется аналогичным образом для тех классов, экземпляры которых имеют индексированные переменные; в качестве аргумента выступает число индексированных переменных. Например,

Association new — порождает экземпляр класса Association, его переменные key и value не имеют разумных значений. Точнее, системой им автоматически присваивается значение nil— неопределенный объект. В системе это единственный экземпляр класса UndefinedObject.

Bag new — порождает экземпляр класса Bag без элементов.

Array new: 2 — порождает экземпляр класса Array с двумя элементами, каждый из которых nil.

String new — порождает экземпляр класса String без символов.

Изучая сообщения класса, определяемые и наследуемые классом в иерархии, в категории instance creation можно найти методы создания его экземпляра. Формально такие методы всегда есть, поскольку все метаклассы наследуют из класса Behavior методы new и new:, но протоколы некоторых классов недостаточны для описания объекта с полностью определенными структурой и поведением. Например, в системе Смолток не существует набора вообще, а есть массивы, множества, упорядоченные наборы и так далее. И хотя синтаксически выражение Collection new правильно, и его можно выполнить, создавая экземпляр класса Collection, такой объект бесполезен. А вот выполнение выражения Set new создаст полностью работоспособный экземпляр, поскольку протокол класса Set полон.

Классы, протоколы которых неполны (что не позволяет им создавать полноценные экземпляры) создаются для того, чтобы описывать те общие характеристики, которые будут наследоваться и конкретизироваться их подклассами. Такие классы называются *абстрактными классами*. К ним относятся классы Object, Collection, Magnitude, Window, ...

Классы, создающие полнофункциональные экземпляры (таких классов большинство), полностью реализуют протоколы своих абстрактных суперклассов и называются *конкретными классами*. Примерами таких классов являются классы Set, Point, Float, ...

1.5. Типы сообщений и их приоритеты

Чтобы заставить объект работать, ему надо послать сообщение. Например, 10 factorial или #(1 2 3) size. Сообщения, для выполнения которых требуется только получатель, называются *унарными*. Встретившись в одном выражении, унарные сообщения выполняются слева направо.

```
#(1 2 3) reversed   →  (3 2 1)    (Набор в обратном порядке)
65 asCharacter      →  $A         "16r0041"
Set name size       →  3          (Длина имени класса Set)
```

Сообщение с именем, состоящим из одного или более ключевых слов (ключевое слово — идентификатор, завершающийся двоеточием), называется *ключевым сообщением*. В выражении за каждым ключевым словом следует аргумент:

```
#(1 2 3 4 5) includes: 4          →  true   (истина)
#(1 2 3 4 5) includes: 7          →  false  (ложь)
#(9 8 7 6 5) copyFrom: 2 to: 4    →  (8 7 6)
'I love you' copyFrom: 8 to: 10   →  'you'
```

Сообщения с одним аргументом и селектором, состоящим из одного или двух специальных (не алфавитно-цифровых) символов, называются *бинарными сообщениями*. Бинарные сообщения всегда выполняются строго слева направо. Арифметические сообщения — частный случай бинарных сообщений. Поэтому, чтобы правильно вычислять арифметические выражения с соблюдением общепринятых математических приоритетов, надо использовать круглые скобки. Существует много и других бинарных сообщений.

```
#(1 2 3), #(4 5 6)   →  (1 2 3 4 5 6)   (конкатенация)
51 < 100             →  true            (сравнение)
2 + 3 * 5            →  25              (2 + 3)* 5
2 + (3 * 5)          →  17
2 @ 3                →  2 @ 3           (создание точки)
```

Рассматривая выражения, в которых происходит сложная посылка сообщений, то есть в которых сообщения посылаются результату выполнения другого сообщения, а аргументы, возможно, получаются как результат посылки некоторых сообщений, надо использовать следующие правила:

1) Унарные сообщения имеют приоритет над бинарными.
2) Бинарные сообщения имеют приоритет над ключевыми.
3) Все сообщения одинакового приоритета выполняются строго слева направо.
4) Круглые скобки имеют наивысший приоритет, и прежде всего выполняются выражения, стоящие в скобках, после чего скобки заменяются на возвращаемые объекты.

Например, первое из нижеследующих выражений правильно интерпретируется, а второе приводит к сообщению об ошибке, поскольку интерпретируется компилятором, как ключевое сообщение с селектором readFrom:on:, которого не существует.

```
Time readFrom: (ReadStream on: '10:00:00 pm').
Time readFrom: ReadStream on: '10:00:00 pm'.
```

При необходимости выполнить несколько выражений, надо каждое выражение завершать точкой. Если одному и тому же объекту надо послать по очереди несколько разных сообщений, следует воспользоваться каскадным сообщением. Каскадное сообщение — стенографический способ записи нескольких сообщений, посылаемых одному и тому же получателю. В таком сообщении объект-получатель указывается один раз, а посылаемые ему сообщения разделяются точкой с запятой. Например,

```
(Array new: 3)
        at: 1 put: $A;
        at: 2 put: $B;
        at: 3 put: $C.
```

1.6. Блоки

Блок в VisualWorks — экземпляр класса BlockClosure. Блок представляет отложенную (не выполняемую) последовательность выражений, описываемых выражениями языка внутри блока. Блок, как и любой другой объект, может присваиваться в качестве значения переменным, передаваться как аргумент при посылке сообщения или возвращаться в качестве результата выполнения выражения. Все выражения, составляющие блок, заключаются в квадратные скобки и разделяются точками. Блок может иметь переменные (параметры), которые стоят в начале блока и отделяются от

выражений блока символом |. Имени каждого параметра предшествует двоеточие :. Блок может иметь временные переменные, которые стоят перед выражениями блока после символа |, если у блока есть переменные блока, или сразу после открывающей квадратной скобки, если переменных блока нет. Временные переменные блока располагаются между двумя символами | ... |. В VisualWorks блоки являются *замыканиями*. Это означает, что их переменные и временные переменные локальны для блока и вне блока не видны.

Блоки можно вкладывать один в другой. Блок выполняется только при посылке ему соответствующего сообщения, вид которого зависит от числа переменных блока. При выполнении блока, если не предписано что-то другое, возвращаемый блоком результат является значением последнего выполненного в блоке выражения (это правило отличается от правила, применяемого при выполнении метода!).

Блок может содержать выражение, перед которым стоит оператор возврата значения ^. Тогда значение, полученное при выполнение этого выражения, возвращается как результат выполнения блока и дальнейшее выполнение блока прекращается. Если блок с оператором возврата значения ^ стоит в теле метода, то вместе с завершением выполнения блока, прекращается выполнение метода, а в качестве возвращаемого методом объекта используется значение, возвращенное блоком.

```
[2 + 3]                              →  [2 + 3]
[2 + 3] value                        →  5
[:letter | letter isVowel]           →  [:letter | letter isVowel]
[:letter | letter isVowel] value: $A →  true
[:a :b | a < b]                      →  [:a :b | a < b]
[:a :b | a < b] value: 5 value: -2   →  false
```

Блок с переменными можно трактовать как своеобразную функцию указанных переменных, тело которой расположено после символа |. И, как для всякой функции, чтобы ее вычислить для конкретных значений переменных, ей следует передать нужное число аргументов и «приказать» произвести вычисления.

Как понятно из примеров, блок без переменных выполняется при посылке ему унарного сообщения value. Блок с одной переменной вычисляется при посылке ему ключевого сообщения value: anObject, аргумент которого подставляется в блок на место его переменной. Блок с двумя переменными выполняется при посылке ему ключевого сообщения value: anObject1 value: anObject2, аргументы которого по порядку подставляются в блок на место его переменных. Блок с тремя переменными выполняется при по-

сылке ему ключевого сообщения value: anObject1 value: anObject2 value: anObject3. Блок с большим числом переменных выполняется при посылке ему ключевого сообщения valueWithArguments: anArray, в котором число элементов массива anArray должно совпадать с числом переменных блока.

1.7. Переменные

Структура любого объекта описывается переменными. Большинство таких переменных имеют имена. Каждая переменная запоминает один объект (своё значение), и имя переменной — просто ссылка на этот объект. Имя переменной используется как ссылка на значение в пределах окружения видимости имени переменной. Таким образом, переменная определяет зависимость между именем и изменяемым значением. Поэтому переменные в системе Смолток не имеют типа (другими словами, смолтоковские переменные имеют динамический тип). Переменные только указывают на объект, и присвоение переменной некоторого значения создаёт новый указатель на объект (новый псевдоним объекта), а не создает копию объекта. Для копирования объектов в системе существуют специальные методы. Объекты, к которым можно получить доступ из конкретного места программы, определяются через доступные в этом месте имена переменных.

Имена переменных состоят из букв, цифр и могут включать символ подчеркивания (_). Имя должно начаться или с литеры или с символа подчеркивания. Имена смолтоковских объектов обычно длиннее, чем в большинстве других языков, что делать смолтоковский код более понятным. Для наглядности, имена часто составляются из двух или более слов, при этом первая буква каждого внутреннего слова печатается с прописной буквы. Это соглашение не предписано правилами языка или его инструментами, но улучшает удобочитаемость кода.

В соответствии с ANSI-стандартом, в смолтоковских идентификаторах нельзя использовать точку. Однако, в VisualWorks используются специальные имена с точками (точечная нотация) для ссылки на разделяемую переменную, класс и пространство имён) (см. главу 3).

Переменные различаются временем своего существования и областью действия (областью видимости). В классических смолтоковских системах определяются следующие типы переменных:

- общие переменные;
- переменные пула;
- переменные класса;
- экземплярные переменные класса;
- переменные экземпляра;
- временные (локальные) переменные метода;
- переменные блоков;
- временные (локальные) пере-

менные блока;
- псевдопеременные,
- аргументные переменные.

1.7.1. Общие переменные классического Смолтока

В классическом Смолтоке общие переменные, называемые еще глобальными переменными, доступны любому объекту системы и все они находятся, как уже отмечалось, в словаре системы с именем Smalltalk. Переменные, содержащиеся в словарях, называемых пулами, в классическом Смолтоке доступны всем экземплярам классов, в которых пул объявлен при определении класса, а также всем экземплярам их подклассов. Все эти переменные «живут» в системе до тех пор, пока их явно не удалят.

Переменные класса доступны этому классу, его подклассам, всем экземплярам класса и его подклассов. Сам класс и все его подклассы разделяют одно значение такой переменной. Эти переменные существуют в системе до тех пор, пока существует определяющий их класс. Имена всех этих переменных, так же как и имена пулов, начинаются с прописной буквы.

Все эти определения в среде VisualWorks теряют смысл, поскольку вместо единого словаря системы с именем Smalltalk вводятся пространства имен. Этому определению посвящена глава 3.

1.7.2. Локальные переменные

В отличие от общих переменных, остальные переменные хранятся внутри каждого объекта, доступны только ему и исчезают вместе с ним. Они относятся к короткоживущим переменным, которые называются локальными или частными переменными. Их имена пишутся с маленькой буквы. Ссылки на эти переменные возможны только внутри того объекта, которому они принадлежат.

Экземплярная переменная класса

Экземплярные переменные класса впервые появились в *IBM Smalltalk* и, после включения в проект стандарта, добавлены во все современные реализации Смолтока. Это переменные класса, рассматриваемого как экземпляр своего метакласса, и которые аналогичны по своим свойствам переменным экземпляра любого класса системы. Например, подклассы могут иметь переменные с такими именами как и их суперкласс, но это другие переменные, значения которых определяются и используются только этими подклассами. Другими словами, экземплярная переменная класса хранит данные, которые изменяются каждым подклассом по иерархии. Она объявляется как часть определения класса, и к ней можно обращаться только методами класса. Такую переменную надо инициализировать один раз, а

класс и все его подклассы могут использовать одни и те же методы для работы с ней. Имена экземплярных переменных класса указываются в определении класса в строке classInstanceVariableNames: '...' и начинаются со строчной буквы.

Переменная экземпляра

Переменная экземпляра содержат данные, которые определяются в каждом конкретном экземпляре класса. Значение такой переменной описывает состояние или атрибут экземпляра. Переменная экземпляра создается, когда создаётся экземпляр, и существует, пока жив сам экземпляр. Окружение видимости такого имени — экземпляр, который является единственным объектом, указывающим на данную переменную.

Есть два вида переменных экземпляра, именованные (имеющие имя) и индексированные (оба типа переменных экземпляра определяются в определении класса). Индексированные переменные экземпляра не называются, а нумеруются, используя целочисленный индекс. Все индексированные переменные экземпляра содержат один и тот же тип объектов, который указывается при определении класса (см. раздел 6.6). Разные экземпляры класса могут иметь разное число индексированных переменных экземпляра. С индексированными переменными работают, используя сообщения at: и at:put:. Класс может определять как именованные, так и индексированные переменные экземпляра. Имена переменных экземпляра указываются при определении класса в строке instanceVariableNames: ' ... '.

Переменные экземпляра наследуются, так что экземпляр класса имеет собственную копию всех переменных экземпляра, объявленных всеми его суперклассами.

Именованные переменные экземпляра доступны по своим именам в методах экземпляра. Как правило, значение именованной переменной можно узнать, посылая экземпляру сообщение из протокола доступа к переменным (accessing), селектор которого совпадают с именем переменной, Чтобы установить новое значение именованной переменной, следует послать экземпляру ключевое сообщение, с селектором, представляющим собой имя переменной с конечным двоеточием, и объект, представляющий новое значение переменной.

Временные переменные метода и блока

К локальным переменным относятся временные переменные, определяемые внутри методов и временные переменные блока, рассмотренные ранее. Все эти переменные создаются в момент вызова метода или блока и уничтожаются по окончании выполнения метода или блока.

Чтобы использовать временные переменные в методах и блоках, их сна-

чала надо описать. Описываются он точно так же, как описываются временные переменные в блоках: между вертикальными линиями-ограничителями | *имя1 имя2 ...* | и обычно располагаются в теле метода сразу за комментарием метода. В качестве примера рассмотрим метод класса из системного класса Time среды VisualWorks:

```
millisecondsToRun: timedBlock
    "Возвращает число миллисекунд, необходимых для
    выполнения блока timedBlock."
    | initialMicroseconds |
    initialMicroseconds := self microsecondClock.
    timedBlock value.
    ^ self microsecondClock – initialMicroseconds
```

Здесь := — оператор присваивания значения, который переменной initialMicrosec присваивает значение в виде объекта, возвращаемого методом класса microsecondClo

Первая строка — millisecondsToRun: timedBlock — это шаблон сообщения, который представлен ключевым сообщением, состоящим из одного ключевого слова millisecondsToRun: (селектора сообщения) и аргумента timedBlock). Сразу после шаблона сообщения в кавычках находится комментарий к определяемому методу. Затем строкой | initialMicroseconds | вводится временная переменная метода с именем initialMicroseconds, с которой могут работать все выражения в теле метода. Далее в теле метода находятся выражения, которые определяют производимые методом операции.

Выражение initialMicroseconds := self microsecondClock производит присваивание значения временной переменной initialMicroseconds. В этом выражении используется псевдопеременная self, которая ссылается на получателя сообщения. В данном случае, поскольку рассматривается метод класса, псевдопеременная self ссылается на класс Time. Сообщение microsecondCloc вызывает метод класса Time, который вычисляет время в миллисекундах, прошедшее с 0 часов 1-го января 1901 года до момента вызова метода; на это число и будет ссылаться переменная initialMicroseconds.

Выражение timedBlock value выполняет блок timedBlock. Здесь timedBlock — имя параметра метода, который используется для ссылки на аргумент сообщения в теле метода и по смыслу должен являться блоком.

Выражение ^ self microsecondClock – initialMicroseconds сначала вычисляет текущее время, из которого затем вычитается время initialMicroseconds и разность возвращается как результат выполнения метода. После этого временная переменная initialMicroseconds прекращает свое существование.

Временные переменные блока описываются и ведут себя также, как и временные переменные метода.

Аргументные переменные

Аргументная переменная — специальный вид временной переменной, которая объявляется в имени бинарных или ключевых методов. Такая переменная получает своё значение от аргументов, передаваемых с посылаемым сообщением. Например, класс Time определяет метод экземпляра

```
hours: hourInteger minutes: minInteger seconds: secInteger
    "Инициализировать все переменные экземпляра."
    hours := hourInteger.
    minutes := minInteger.
    seconds := secInteger
```

Этот метод объявляет в имени метода три временных переменных с именами hourInteger, minInteger и secInteger. Когда объект-клиент посылает это сообщение экземпляру класса Time, к которому он мог бы обратиться как к aTime, с сообщением передаются соответствующие целые числа. Например: aTime hours: 11 minutes: 42 seconds: 15 Когда метод вызывается, указанные значения назначаются соответствующим аргументным переменным, так hourInteger получает значение 11, minInteger — 42, secInteger — 15. Аргументные переменные, в отличие от других временных переменных не принимают новых значений, так что первоначальные назначения не изменяются во время жизни такой переменной.

Как соглашение, имя аргументной переменной должно указывать на объект, тип которого передается (например, aSet, aString, anInteger). Однако, это только соглашение. Если метод не может обработать переданный объект, во время выполнения возникает сообщение об ошибке.

1.7.3. Псевдопеременные

В Смолтоке есть еще и псевдопеременные. Их имена начинаются со строчной буквы, но они доступны всем объектам системы. Псевдопеременные — это имена-указатели специальных объектов системы, и, в отличие от переменных, их значения не могут изменяться. Псевдопеременными в системе Смолток являются nil, true, false, self, super.

Псевдопеременная nil

Псевдопеременная nil указывает на специальный объект — единственный экземпляр класса UndefinedObject, используемый, когда нет другого подходящего объекта. Например, при создании нового экземпляра класса, если не указаны значения его переменных, все они получат значение nil. Кроме того, объектом nil часто представляют бессмысленный результат.

Псевдопеременные true и false

Псевдопеременные true и false представляют логическую истину и логическую ложь и являются единственными экземплярами классов True (Истина) и False (Ложь), соответственно. Сами классы True и False — подклассы класса Boolean, в котором описан общий протокол поведения этих двух логических объектов. В классах True и False реализованы следующие сообщения, выполняющие логические операции:

& aBoolean — возвращает true, если и объект, получивший сообщение, и аргумент aBoolean истинны (операция вычисляющего "И", в которой аргумент aBoolean всегда вычисляется, независимо от получателя сообщения).

| aBoolean — возвращает true, если или получатель сообщения, или аргумент aBoolean истинны (операция вычисляющего "ИЛИ").

not — отрицание: если получатель сообщения false, возвращает true, если получатель сообщения true, возвращает false.

eqv: aBoolean — возвращает true, если получатель сообщения эквивалентен (тождествен) аргументу aBoolean, иначе возвращает false.

xor: aBoolean — исключающее "ИЛИ": возвращает true, если получатель сообщения не эквивалентен aBoolean, иначе возвращает false.

and: alternativeBlock — если получателем сообщения является true, возвращает результат выполнения блока alternativeBlock, который не должен иметь аргументов; если получателем сообщения является false, возвращает false без выполнения блока.

or: alternativeBlock — если получателем сообщения является false, возвращает результат выполнения блока alternativeBlock, который не должен иметь аргументов; если получателем сообщения является true, возвращает true без выполнения блока.

Обычно логические объекты возвращаются после посылок сообщений, которые требуют простого ответа типа «да—нет» (например, сообщений для сравнения величин: <, <=, >, >=).

Не пытайтесь создавать дополнительные экземпляры классов UndefinedObject, True, False! Хотя это и можное сделать, используя сообщение basicNew, но созданные так объекты вызовут крах системы VisualWorks!

Псевдопеременные self и super

Особое место в системе занимают псевдопеременные self и super, которые используются в теле методов и указывают на объект, который вызвал данный метод. Их значение изменяется в соответствии с контекстом выполнения кода. Различаются они способом поиска того метода, который необходимо выполнить объекту: self начинает поиск метода в классе, которому

принадлежит объект, получивший сообщение, а super — в суперклассе того класса, в котором находится метод, содержащий псевдопеременную super (см. [3, Глава 4] или [15, раздел 2.5]).

Использование псевдопеременной super не в качестве получателя сообщения (а, например, в качестве аргумента) полностью совпадает с использованием псевдопеременной self.

Псевдопеременная thisContext

Псевдопеременная — thisContext, является ссылкой на контекст стека текущего процесса. В то время как псевдопеременные self и super часто используются при программировании на Смолтоке, псевдопеременная thisContext редко используется в приложениях. С ней в основном работают обработчик особых ситуаций системы и отладчик.

1.7.4. Назначение значения переменной

Значение по умолчанию для любой переменной — nil. Чтобы назначить новое значение переменной, следует использовать встроенную в систему операцию назначения значения :=, как, например, в выражениях:

```
prompt := 'Enter your name'.
flavors := #('chocolate' 'vanilla' 'mint chip').
ramdomNumber := (Random new next) * 3.
```

Назначения могут выстраиваться в цепочку, когда две или более переменных должны хранить одно и то же значение, как, например, в выражении majorLoopCounter := minorLoopCounter := 1. Назначения по цепочке должны использоваться только с литералами или только для чтения, иначе, изменение значения одной переменной, как побочный эффект, изменит значение и другой переменной.

Выражение назначения значения := не является селектором метода, хотя по форме напоминает бинарный селектор.

1.8. Методы и примитивные методы

Вся работа осуществляется объектами, которым посылаются сообщения. В ответ объекты вызывают для исполнения методы, в которых в свою очередь посылаются сообщения другим объектам, и так далее. Но только пересылая сообщения от одного объекта к другому, толку не добьешься. Кто-то должен взаимодействовать со средой, обращаться к операционной системе. Но кто и как?

В методах многих классов встречается строка вида <primitive: *nn*>, где *nn* — некоторое число. Такое обозначение имеют *примитивные методы* системы, реализованные в виртуальной машине, а число *nn* идентифицирует примитивный метод в выражениях, написанных на языке Смолток. В теле

метода такая конструкция располагается самой первой. Примитивные методы, как правило, выполняют операции низкого уровня (такие как арифметические операции, выделение памяти) или операции, критичные по производительности.

Обычно такой метод состоит из двух частей: примитивного метода с номером, и части, написанной на Смолтоке. Сначала выполняется примитивный метод. Его выполнение заканчивается либо успехом (возвращением некоторого объекта), либо неудачей. В первом случае, выполнение метода прекращается и возвращенный примитивом объект возвращается как результат выполнения всего метода. Код на языке Смолток в этой ситуации выполняться на будет. Если примитивный метод не может быть выполнен, то выполняется часть метода, написанная на языке Смолток. Например,

```
+ aNumber
    <primitive: 41>
    ^aNumber sumFromFloat: self
```

Такое разделение ответственности очень эффективно работает, поскольку примитивы управляют наиболее часто используемыми, но простыми случаями.

1.9. Соглашения о форматировании кода

Компилятор игнорирует в коде символы табуляции, перевода каретки и дополнительные пробелы. Форматирование кода не влияет на его выполнение, но для достижения его удобочитаемости, следует придерживаться следующих простых соглашений.

1) Начинать определение выражения посылки сообщения с левого поля, и выравнивать всё содержание метода на одном уровне.
2) Оставлять пустую строку ниже комментария метода и как разделитель между разделами длинного метода.
3) После каждой точки, заканчивающей выражение, переходить на новую строку.
4) Отступать вправо при необходимости, чтобы визуально идентифицировать каждый зависимый раздел кода.

Браузер кода, используемый в VisualWorks обеспечивает команду Format для автоматического применения этих правил.

1.10. Контрольные вопросы

1) Что такое объект?
2) Что такое структура и поведение объекта?
3) Что такое интерфейс объекта?

4) Что такое инкапсуляция?
5) Что такое сообщение, посылка сообщения, селектор сообщения?
6) Что такое метод, имя метода?
7) Что представляет собой возвращаемый методом объект?
8) Что такое класс? Что такое экземпляр класса?
9) Что такое наследование, подкласс и суперкласс? Что такое иерархия классов?
10) Что такое полимозфизм?
11) Являются ли в Смотоке классы объектами?
12) Что такое метакласс? Что является его экземплярами? Сколько экземпляров может иметь каждый метакласс?
13) Экземплярами какого класса является каждый метакласс?
14) Подклассами какого класса является каждый метакласс? Как строится иерархия метаклассов?
15) Как используется иерархия для поиска метода, выполняемого по сообщению?
16) Что такое оператор возврата объекта?
17) Что такое смолтоковское выражение?
18) Что такое литеральный объект? Какие объекты в Смолтоке задаются литерально?
19) Экземплярами каких классов являются символы, числа, системные имена, строки, массивы?
20) Посылкой каких сообщений определяется новый объект?
21) Что такое абстрактные и конкретные классы?
22) Какие типы сообщений используются в Смолтоке? Какв их приоритет?
23) Что такое именованные и индексированные переменные объекта?
24) Что такое блок? Какие типы блоков встречаются в Смолтоке?
25) Что такое временные переменные метода и блока?
26) Что такое аргументная переменная?
27) Что такое псевдопеременная?
28) Чем схожи и чем отличаются друг от друга псевдопеременные self и super?
29) Что такое операция назначения значения переменной?
30) Что такое примитивные методы и как они используются?

Глава 2

Прогулка по *VisualWorks*

VisualWorks— многоплатформенная, функционально полная среда разработки приложений, включающая

- реализацию языка Смолток (Smalltalk),
- виртуальную машину (или двигатель объектов) для выполнения смолтоковского кода,
- обширную библиотеку классов,
- широкий набор инструментов для создания приложений.

Цель этой главы — на примере разработки простого приложения показать некоторые основные инструменты среды VisualWorks 7.4.1, особенности среды и языка Смолток. После первого и быстрого обзора, следующие главы будут содержать подробное изложение большинства из рассмотренных в этой главе тем.

2.1. Установка и запуск *VisualWorks*

Установка VisualWorks на платформе MS Windows происходит точно так же, как и любого другого Windows-приложения. После стандартной инсталляции, запуск приложения производится двойным щелчком на иконке приложения, расположенной на рабочем столе, или через кнопку Пуск рабочего стола Windows:

Пуск → Программы → VisualWorks → VisualNC

Как результат, на экране откроются два окна (см. рис. 2.1). Одно из которых — стартовое окно системы (VisualWorks Launcher). Это центр управления средой, позволяющий выполнять в ней все необходимые операции. Второе окно — рабочее окно с кратким описанием среды VisualWorks на нескольких страницах. Чтобы открыть страницу, следует щелкнуть на её закладке (ярлыке) левой кнопкой мыши.

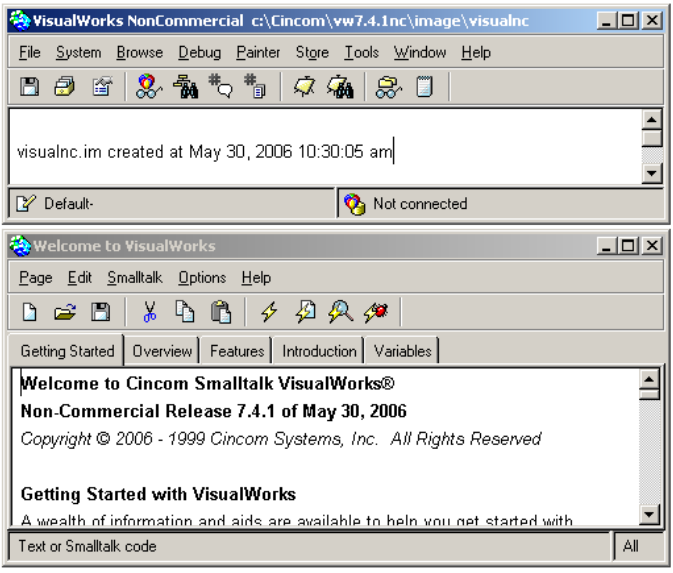


Рис. 2.1: Окна, возникающие при первом запуске VisualWorks.

За этими простыми действиями скрывается своеобразный механизм запуска среды VisualWorks, которая выполняется как виртуальная машина, обрабатывающая данные в смолтоковском образе. Виртуальная машина — исполняемый файл с расширением `*.exe`, который интерпретирует и выполняет смолтоковский байт-код, хранящийся в файле образа среды с расширением `*.im`.

Для каждой операционной системы, поддерживаемой VisualWorks, существует своя виртуальная машина, имя файла которой указывает ту операционную систему, в которой она выполняется. Так, например, для запуска среды разработки приложений VisualWorks используются двигатели объектов, имена которых имеют формат vw<platform>, например vwnt.exe для Microsoft Windows NT или vwlinux86 для Linux. По умолчанию, виртуальная машина среды устанавливается в подкаталог `\bin\<platform>` корневого каталога инсталляции VisualWorks.

Формат файла образа среды не зависит от платформы. Если при запуске файл образа не задан, то по умолчанию виртуальная машина в своём каталоге ищет файл образа с тем же именем, что и файл виртуальной машины. Например, если загружаемая виртуальная машина — `visual.exe`, то ищется файл образа `visual.im`. Но для запуска образа с именем, от-

личным от имени виртуальной машины, последней следует указать путь к нужному файлу образа, и запустить двигатель объектов с этим файлом образа в качестве аргумента. Например, можно создать `*.bat`-файл запуска с таким текстом (если среда VisualWorks инсталлирована в каталог `vw` на диске C:):

```
c:\vw\bin\win\visual.exe c:\vw\image\visual.im
```

При одновременной работе над несколькими приложениями, можно для каждого приложения иметь собственный файл образа и отдельный `*.bat`-файл для его запуска.

Можно производить запуск среды VisualWorks из командной строки, используя при этом опции, которые позволяют уточнить то, как происходит запуск системы. Описание различных виртуальных машин и опций приведено в [9, C Virtual Machines].

2.2. Настройка среды

Стартовое окно даёт возможность запускать множество инструментов VisualWorks, либо выбирая соответствующие пункты меню, либо щелкая на кнопках из панели инструментов. Кроме этого, в стартовом окне находится текстовая панель, которая называется Transcript. Она отображает информационные сообщения, сгенерированные средой VisualWorks или кодом приложений. Transcript — экземпляр класс TextCollector и потому отображает только строковые данные.

Для получения справочной информации о системе, следует щелкнуть в панели инструментов стартового окна на иконке со знаком вопроса (первой справа, если она есть). Для получения справочной информации о конкретном окне, следует сделать это окно активным, а затем либо воспользоваться первой командой из меню Help этого окна, либо нажать клавишу F1. Например, чтобы отобразить справочную информацию о стартовом окне Launcher следует воспользоваться его командой меню Help → Launcher Help.

Прежде чем работать в среде, её нужно должным образом настроить. Для этого нужно открыть окно установки параметров среды (Settings), либо выбирая в основном окне команду меню System → Settings, либо нажимая в панели инструментов на кнопку окна установок (третью слева). Выбор строки в левой списковой панели окна, открывает страницу для установки соответствующего параметра.

Пользуясь окном Settings, настроим характер операций, выполняемых с помощью кнопок мыши. Есть три общих набора первичных операций, соответствующих трем кнопкам мыши:

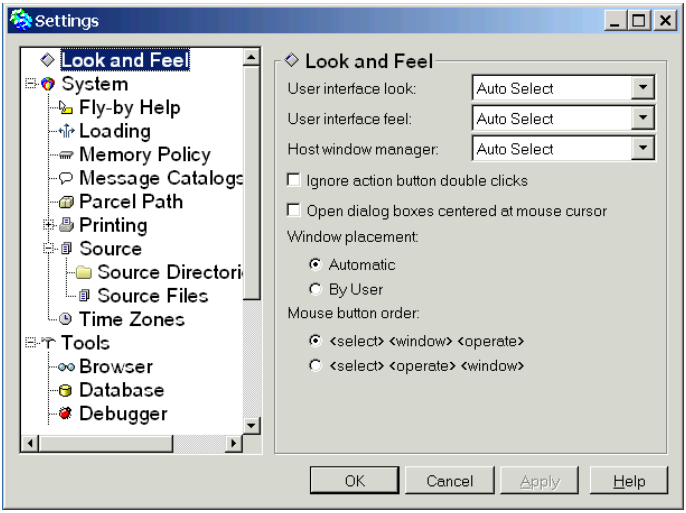


Рис. 2.2: Окно установки параметров среды.

- Кнопка <Select> выбирает объекты и текст.
- Кнопка <Operate> открывает всплывающее меню операций, содержащее команды меню, соответствующие текущей панели окна. В зависимости от типа панели, меню меняется.
- Кнопка <Window> открывает оконное меню, содержащее команды для работы с текущим окном.

В случае 3-кнопочной мыши можно использовать два разных порядка связывания кнопок мыши с этими операциями (слева направо):

<Select> <Window> <Operate> или <Select> <Operate> <Window>.

Выбор нужного порядка надо провести на странице Look and Feel окна Settings. Мы рекомендуем выбрать первый вариант, когда основными кнопками являются левая и правая кнопки мыши, меню <Window> используется не часто.

В случае двухкнопочной мыши VisualWorks позволяет связать операции с копками мыши следующим образом: <Select> — левая кнопка, <Operate> — правая кнопка <Window> — <Ctrl>+<Select>.

Меню (Operate) — наиболее важное меню в VisualWorks. Многие операции, описанные далее, фактически выполняются при выборе команды из всплывающего меню Operate соответствующей панели. Команды этого меню объясняются по мере необходимости.

Проверим правильность установки системной переменной $(VISUALWORKS), которую VisualWorks устанавливает при инсталляции и которая хранит домашний каталог инсталляции. Если по каким-то причинам это не сделало или сделано не верно, некоторые инструменты среды будут работать не правильно. Например, нельзя будет увидеть исходный текст из библиотеки классов или файлы загрузки дополнительных инструментов. Следует проверить правильность определения домашнего каталога, для чего выбрать в левой списковой панели строку System, в поле ввода VisualWorks home directory: ввести или выбрать (нажимая кнопку Browse. . .) путь к домашнему каталогу (если он установлен не правильно), и, наконец, сохранить изменения. На эту же страницу окна установок параметров среды можно попасть, выбирая в стартовом окне команду меню File → Set VisualWorks Home. . . .

На странице Source полезно выбрать одну из двух радио-кнопок, определяя в каком формате будет сохраняться смолтоковский код в текстовых файлах при выполнении команд File Out и File Out As. . . : в «старом» формате порций данных (Chunk Format), или в «новом» формате XML (XML Format), который используется по умолчанию. Формат порций необходим только тогда, когда предстоит переносить код в другие смолтоковские среды, не понимающие файлов в XML-формате.

Установки среды можно сохранить, выбирая сначала команду Save. . . из всплывающего меню операций списковой панели окна установок, а, затем, в открывшемся диалоговом окне, определяя имя и каталог для файла установок параметров среды и, наконец, щелкая на кнопке Save. Сохраненные ранее установки можно загрузить из файла, выбирая команду Load. . . из всплывающего меню операций списковой панели.

2.3. Разработка простого приложения

Уже стало традицией начинать изучение нового языка или знакомство с новой средой примером Hello, World! (Привет, Мир!). Не будем нарушать традиций!

Самый простой способ решить такую задачу в VisualWorks— открыть новое рабочее окно командой меню Tools → Workspace, в его текстовой панели обычным образом ввести выражение (см. рис. 2.3)

```
Transcript cr; show:'Hello World!'; cr.
```

Затем выбрать в рабочем окне команду меню Smalltalk → Do It. . . , отображая Hello World! в текстовой панели Transcript стартового окна.

Однако, по аналогии с другими языками программирования, большинство пользователей хотело бы видеть автономное приложение, а не решающий задачу смолтоковский код в среде разработки приложений. Это по-

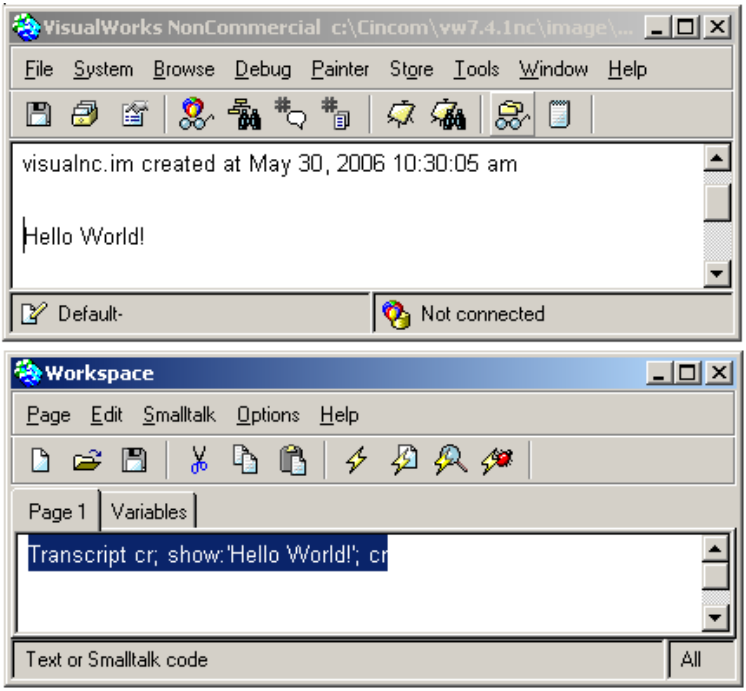


Рис. 2.3: Простой способ вывода строки Hello, World!.

требует, во-первых, создания одного класса и одного метода в этом классе, и во-вторых, создания загрузочного модуля для автономного запуска приложения.

Создание класса

Чтобы написать программу, воспользуемся основным инструментом среды VisualWorks— системным браузером, окно которого открывается командой меню Browse → System) основного окна среды. Системный браузер позволяет работать с кодом, определяющим все объекты среды (см. главу 6). Объём уже включенного в среду исходного текста огромен. В нём легко заблудиться! Наша задача: сосредоточиться на тех частях системы, которые нужны для разработки приложения, и проигнорировать остальное. Для этого воспользуемся инструментами, позволяющие организовать создаваемый код и удобно работать с ним.

В левой верхней панели браузера системы в виде дерева перечисляются пакеты (или, по-другому, категории) и связки пакетов, содержащие

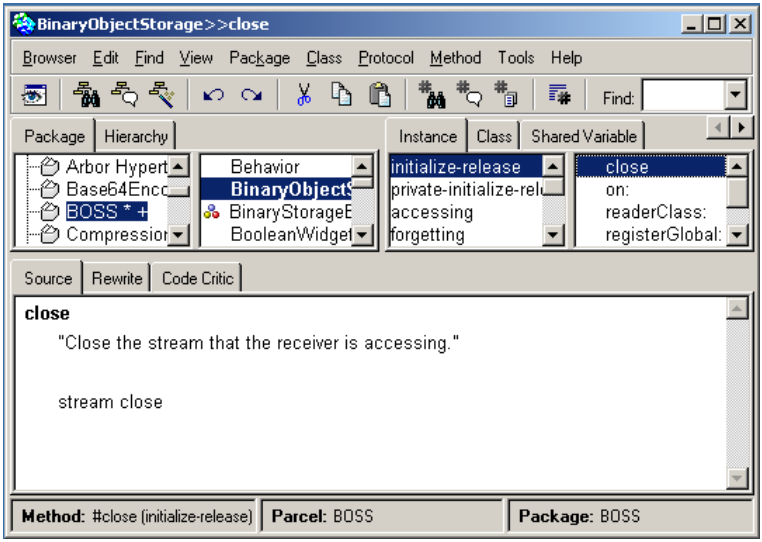


Рис. 2.4: Окно системного браузера.

код среды. Чтобы создаваемая программа не затерялась, отделим её от исходного текста среды, создавая собственный пакет. Для этого выберем в системном браузере команду меню Package → New Package. . . , в появившемся диалоговом окне введём имя пакета, скажем, Hello World и нажмём кнопку OK. Определённый пакет окажется выбранным в дереве пакетов и связок пакетов. Это то место, где мы будем работать.

Чтобы сосредоточиться на создаваемой программе, можно открыть новый браузер — браузер пакета, отображающий содержимое только данного пакета. При выбранном пакете Hello World, выберем в системном браузере команду меню Package → Spawn. Панели нового браузера пусты, но сам браузер имеет заголовок, указывающий, что он относится к пакету Hello World (см. рис. 2.5).

Теперь создадим новый класс. В браузере пакета, выберем пункт меню Class → New Class. . . , открывая диалоговое окно создания класса.

Знак «Внимание!» (желтый треугольник с восклицательным знаком) в диалоговом окне определения любого объекта указывает на поле ввода, требующее обязательного заполнения.

В данном случае — это поле ввода имени класса. Напечатаем в нём имя HelloWorld (в отличие от имени пакета, в имя класса пробел вставлять нельзя!). Остальные поля оставим со значениями по умолчанию и нажмём кнопку OK. Класс будет создан (скомпилирован), добавлен в пакет, а его имя

отобразится в браузере пакета (и в системном браузере, если он остался открытым).

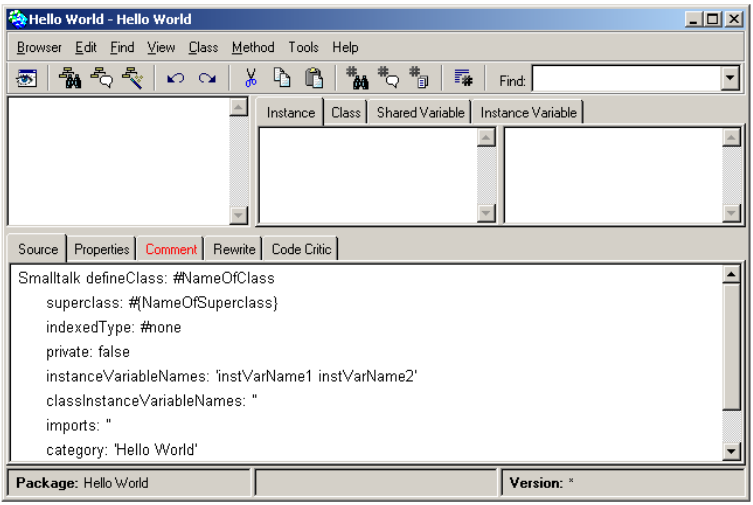


Рис. 2.5: Окно браузера для пакета HelloWorld.

Создадим метод, выполняющий основную работу. Код метода должен открывать диалоговое окно, отображающее строку 'Hello, World!', а затем закрывать его, когда пользователь щелкает на кнопке Hello. Воспользуемся методом класса choose:lables:values:default: из класса Dialog, уже определенных в среде в одном из системных пакетов. В браузере пакета выберем класс HelloWorld, затем категорию (или, другими словами, протокол) методов экземпляра initialize-release и, наконец, метод с именем initialize. Всё это было создано средой в момент создания класса. Заменим код метода, предлагаемый средой, на следующий

```
initialize
    Dialog choose: 'Hello, World!'
            labels: (Array with: 'Hello')
            values: #(nil)
            default: nil.
```

Выбирая в браузере пакета команду меню Edit → Accept или нажимая клавиши Ctrl + S, сохраним изменения. Можно проверить, как код работает, для чего напечатаем в рабочем окне текст HelloWorld new, выберем его и выполним, выбирая команду меню Smalltalk → Do It или нажимая клавиши Ctrl + D (см. рис. 2.6).

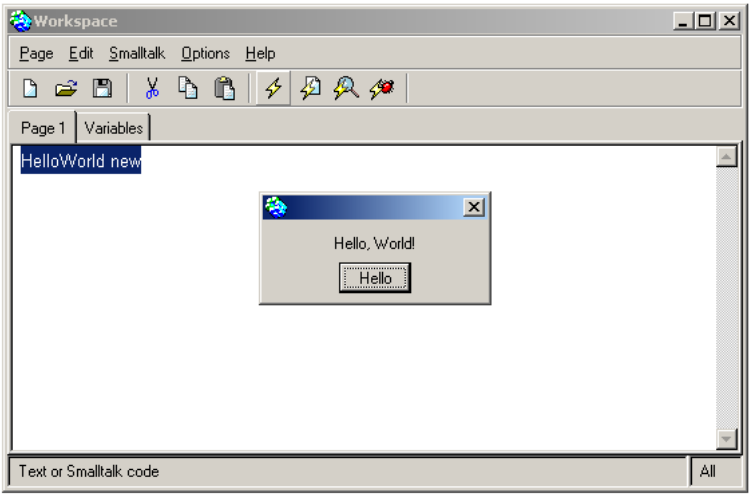


Рис. 2.6: Вывод строки Hello, World! в диалоговом окне.

2.4. Сохранение созданного кода

Сохранение в файле образа

В большинстве сред программирования созданный код сохраняется в редактируемом файле. Затем такой файл компилируется и создается исполняемая версия программы. В VisualWorks созданный код традиционно сохраняется в файле образа среды. При этом можно изменять имя файла, сохраняя свой образ для каждого из создаваемых приложений. Файл образа — "мгновенный снимок" VisualWorks (бинарный файл), включающий весь код, который составляет среду разработки: библиотеку классов, инструменты и создаваемое приложение. Когда для сохранения работы используется именно этот путь, следует сохранять образ достаточно часто, и обязательно в конце каждой сессии разработки.

Чтобы сохранить текущее состояние образа в том же файле образа, который использовался для запуска среды надо выбрать в стартовом окне системы команду меню File → Save Image. В этом случае содержание текущего файла образа заменяется текущим состоянием образа системы. При следующем запуске среда VisualWorks будет иметь то состояние, которое было сохранено в файле образа. Именно так и поступим, чтобы сохранить приложение Hello World.

Чтобы сохранить текущее состояние образа а файле с новым именем, надо выбрать команду File → Save Image As. . . , открывая диалоговое ок-

но, запрашивающее имя (без расширения `*.im`) нового файла образа (по умолчанию, отображается имя файла текущего образа).

Настоятельно рекомендуется, первоначальный образ VisualWorks со всеми нужными инструментами и пакетами (или его копию) хранить без изменений в отдельном файле. Это обеспечит возможность начинать всё с «чистого листа» всякий раз, когда необходимо. Ради удобства пользователя, такой «дополнительный» файл включен в поставку системы и находится в каталоге `image` в виде файла `image.im.zip`

Отметим важную особенность файла образа: он тесно связан с двумя файлами — файлом исходного кода (sources) и файлом изменений (changes). Эти три файла синхронизированы и должны копироваться и сохраняться вместе, создавая полную запись среды. Файл исходного кода содержит в формате XML исходный текст образа первоначальной среды VisualWorks без всех последующих изменений. По умолчанию его имя `visual.sou`. Имя файла исходного кода никогда не меняется.

Файл изменений, который обычно имеет то же самое имя, что и файл образа, но с расширением `cha`, содержит исходный текст всех изменений, сделанных в системе, независимо от того, сохраняется образ или нет. По этому файле можно восстановить потерянную информацию в случае краха среды. Файл изменений со временем может стать очень большим, и должен иногда сжиматься с помощью команды меню System → Changes → Condense Changes стартового окна VisualWorks. При этом из файла изменений удаляется всё, кроме последних изменений.

Сохранение кода в текстовом файле

Как и в любой смолтоковской среде, в инструментах, работающих с исходным текстом, код можно сохранить в текстовом файле (в файле с расширением *.st) командой File Out As... и позже загрузить в среду командой File In... (см. раздел 4.3).

Чтобы сохранить в текстовом файле код класса HelloWorld, следует выбрать этот класс, из всплывающего меню этой же панели выбрать команду File Out As..., определить имя файла (согласится с именем, предлагаемым средой) и нажать кнопку Save. Код сохранится в файле HelloWorld.st. По умолчанию файл сохраняется в каталоге \image\.

Команды записи кода в текстовый файл доступны через многие команды меню различных браузеров. В зависимости от меню, команда будет записывать в файл разные наборы определений.

Чтобы восстановить в образ код из файла исходного текста, следует воспользоваться командой File In... инструмента File Browser (браузер файлов), окно которого открывается командой меню File → File Browser старто-

вого окна (см. также раздел 5.3). Когда в панели файлов этого инструмента выбран нужный *.st-файл, команду File In... надо выбрать из всплывающего меню этой панели.

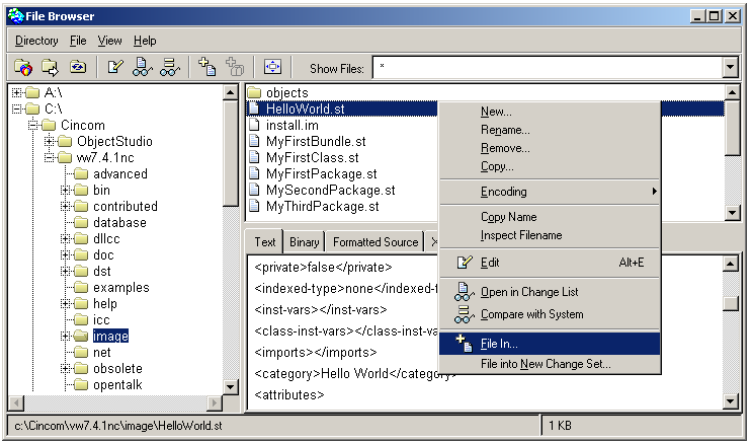


Рис. 2.7: Команда File In... в браузере файлов.

2.5. Создание автономного приложения

Пока программу HelloWorld можно выполнить только в среде разработки VisualWorks. Но программу можно выполнить автономно. Для этого программу надо специально подготовить, используя специальный инструмент Runtime Packager.

Runtime Packager содержится во внешнем программном модуле — парселе. Обычно в парселах хранится редко используемый код, разгружаемый в среду по мере надобности. Самый простой способ это сделать, воспользоваться инструментом Parcel Manager(Администратор Парселов), окно которого открывается командой меню System → Parcel Manager стартового окна (см. рис. 2.8).

В левой списковой панели окна администратора парселов выбрать строку Essentials. В правой верхней списковой панели дважды щелкнуть на строке RuntimePackager (или выбрать парсел и выбрать команду меню Parcel → Load). После чего, закрыть окно Parcel Manager.

Перед тем, как работать с инструментом RuntimePackager закроем все открытые окна среды VisualWorks, кроме окна системного браузера. Чтобы запустить программу RuntimePackager, следует выбрать в стартовом окне команду меню Tools → Runtime Packager. Откроется окно инструмента, в

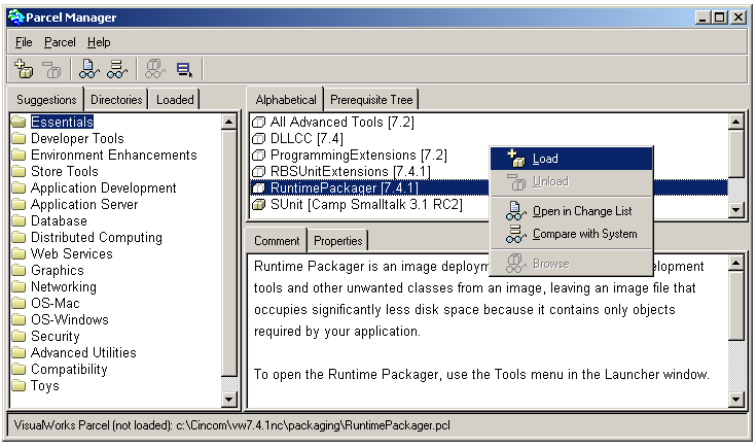


Рис. 2.8: Окно администратора парселов.

левой панели которого отображается текст, объясняющий по-порядку (надо только нажимать клавиши Next и Previous), что надо делать на каждом этапе упаковки приложения (см. рис. 2.9).

В окне инструмента существует множество опций, доступных во время упаковки программы, но в нашем простом случае почти все можно проигнорировать, поскольку подойдут значения по умолчанию.

Итак, дважды щелкая на кнопке Next, перейдем на страницу Set common options (Установить общие опции) и щелкнем не кнопке Do this step (Выполнить этот этап). Откроется блокнот с несколькими страницами настроек. На странице Basics нужно ввести информацию о том, как запустить программу. Нашу программу выполняет выражение HelloWorld new, поэтому в поле ввода StartUp Class введём имя класса HelloWorld, а в поле ввода StartUp Method — имя метода new. В поле ввода Runtime Image Path Name введём имя файла образа для нашего приложения: hello.

На странице Details представлено множество опций. Обратим внимание на раздел Action on last window close (Действие на закрытие последнего окна). Значение по умолчанию — Shutdown image (Закрыть образ) именно то, что нужно: отобразить диалоговое окно со строкой Hello World!, и, после его закрытия, завершить работу приложения. Также обратим внимание на опцию Suppress splash screen and herald sound (Подавить появление заставки и звукового сопровождения). Если она выбрана (по умолчанию), заставка и звуковое сопровождение VisualWorks не будут использоваться при запуске образа приложения.

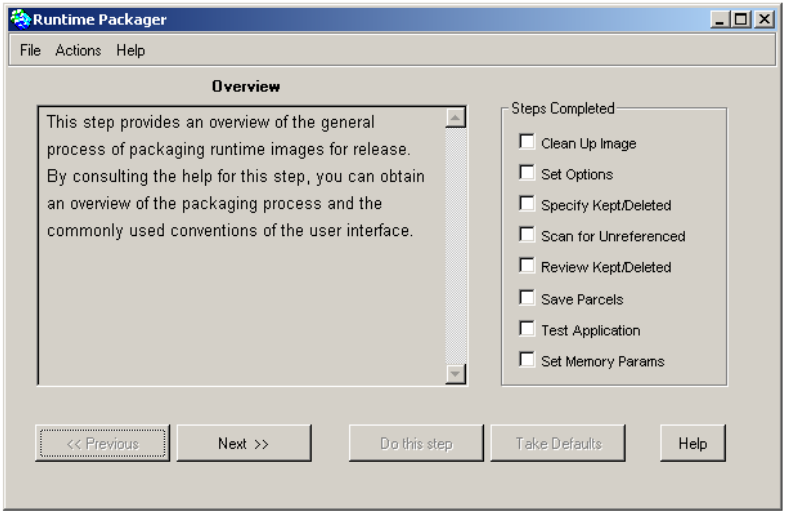


Рис. 2.9: Стартовое окно инструмента Runtime Packager.

Заглянем ещё на страницу Platforms (Платформы). Образы VisualWorks полностью переносимы между платформами операционной системы, Один и тот же образ может выполняться на любой из поддерживаемых VisualWorks платформ. Но, если удалить части поддержки других платформ, то подобное станет невозможно. На этой странице можно выбрать «политику просмотра» и поддержку указанных операционных систем, чтобы включить их в создаваемый образ. По умолчанию, все опции выбраны, всё так и оставим.

Нажмём кнопку OK, чтобы сохранить установленные опции и возвратиться в окно Runtime Packager. Щелкнем на кнопке Next пять раз, чтобы добраться до окна Test application (Протестировать приложение). В этом окне щелкнем на кнопке Begin Test. Отобразится диалоговое окно приложения. Нажмем в нём кнопку Hello, закрывая его. Затем щелкнем на кнопке End Test и кнопке OK, чтобы закрыть окно тестирования.

Дважды щелкая на кнопке Next, доберемся до страницы Strip and save image (Произвести разбор и сохранить образ). Слово «разбор» относится к тому, что будет сделано с образом. Из множества классов, используемых в среде разработки, для выполнения приложения нужны далеко не все, и их следует удалить из образа. Как только это сделано, образ сохраняется в файле образа с именем, определенном на странице Set Common Options (в примере в файле с именем hello).

После щелчка на кнопке Do this step, появиться диалоговое окно, говоря, что все открытые окна, ссылающиеся на классы, будут закрыты (включая окна браузеров). Щелкните на кнопке Yes, закрывая все такие окна. Однако, если остались открытыми рабочие окна или окно браузера файлов, они не будут закрыты. В этом случае надо щелкнуть на кнопке No, закрыть их, и повторить этап, щелкая на кнопке Yes.

Могут появляться и другие диалоговые окна. Читайте их сообщения и отвечайте соответствующим образом. По большей части это будет информация о том, что будет сделано, с просьбой подтвердить продолжение операции создания нового образа. После того, как все изменения сделаны и записаны в файл образа, среда VisualWorks закрывается.

Как результат, существует образ, который можно запускать независимо от среды разработки. Обычно он находится в каталоге \image или корневом каталоге инсталляции VisualWorks. Чтобы запустить его следует выполнить командную строку запуска образа с созданным образом приложения (hello.im). Например,

```
c:\vw\bin\win\vwnt.exe c:\vw\image\hello.im
```

Для конечного пользователя в среде Windows более привычной была бы ситуация с запуском файла `hello.exe`, а не последовательности `visual.exe hello.im`. Это легко сделать. Достаточно, используя инструменты Windows, создать копию файла `visual.exe` с именем `hello.exe` и расположить файлы `hello.exe hello.im` в одном каталоге (не имеет значения в каком). Теперь, чтобы запустить приложение, следуя привычке, нужно выполнить файл `hello.exe`. Напомним, что по умолчанию виртуальная машина ищет в своём каталоге файл образа с тем же именем.

2.6. Выход из среды

Чтобы завершить работу в среде VisualWorks, надо выбрать в стартовом окне команду File → Exit VisualWorks. В открывшемся диалоговом окне нажать кнопку No, чтобы выйти из VisualWorks без сохранения образа, нажать кнопку Yes, чтобы сохранить образ, возможно в файле с новым именем, а затем выйти из VisualWorks, или нажать кнопку Cancel, чтобы остаться в среде разработки.

Если среда VisualWorks прекратила отвечать на ввод пользователя, можно попробовать несколько способов аварийного выхода. Команда <Control>+\ откроет окно Process Monitor, в котором будут перечислены запущенные процессы VisualWorks. При этом все процессы интерфейса пользователя (UI) приостанавливаются. Можно выбрать любой процесс и отладить его, чтобы найти причину возникшей проблемы. Команда <Control>+y от-

крывает отладчик на текущем процессе, минуя окно Process Monitor.

Если и это не поможет, можно попытаться воспользоваться командой экстренного выхода <Shift>+<Control>+y, открывая окно, в котором перечисляются смолтоковские выражения для экстренного выхода, завершаемые нажатием клавиши <Escape>.

Если всё это не даст результата, VisualWorks придётся закрыть средствами операционной системы, но образ сохранить уже не удастся и в следующий раз система запуститься с последним сохраненным образом. Все изменения в среде, сделанные с момента последнего сохранения и до момента принудительного закрытия VisualWorks, будут потеряны, но они будут записаны в журнале — файле с расширением `*.cha`.

2.7. Контрольные вопросы

1) Что такое виртуальная машина (или двигатель объектов) для выполнения смолтоковского кода?
2) Как запустить *VisualWorks*?
3) Что хранить файла образа среды, файл исходного текста и файл изменений? Как они взаимодействуют друг с другом?
4) Как часто следует сохранять файл образа?
5) Какие команды позволяют сжать файл изменений?
6) Для чего используется панель Transcript стартового окна среды?
7) Как происходит настройка среды VisualWorks?
8) Как сохранить и загрузить настройки среды?
9) Как получить справочную информацию о среде и инструментах среды?
10) Какие три системные операции связаны с кнопками мыши? Как связать кнопки мыши с этими операциями?
11) Какой основной инструмент среды VisualWorks используется для создания приложения в среде разработки?
12) Как сохранять созданный код в файле образа?
13) Как сохранить созданный код в текстовом файле? Какие форматы при этом можно использовать?
14) Как подготовить приложение для автономного выполнения? Какой инструмент можно использовать?
15) Как выйти из среды разработки в случае нормального и аварийного завершения работы?

Глава 3

Пространства имён

Пространства имён позволяют справиться в проблемой конфликта имён (совпадения имён). Решение такой сложной задачи требует системного подхода, а не какого-то «обходного маневра» или «заплатки». Поэтому VisualWorks 7.4.1 вводит пространства имён как новую языковую конструкцию, которая позволяет добавлять в VisualWorks компоненты, разработанные разными программистами, не опасаясь совпадения имен разных сущностей.

Решение проблемы конфликта имён в VisualWorks, давно существующее в других средах программирования, состоит в том, чтобы разделить глобальное пространство имён на части (подпространства), после чего имена не обязаны быть уникальными во всей смолтоковской среде, а только в пределах своей части. В результате, имя в одном подпространстве может быть скрыто от другого подпространства, пока явно не потребуется отмена сокрытия. И если два одинаковых имени находятся в различных подпространствах имён, то каждое из них можно точно идентифицировать с помощью имени содержащего его подпространства. Таким образом, устранение неоднозначности происходит за счет квалификации имени.

3.1. Особенности введения пространств имён

Как уже отмечалось, изначально, Смолток всегда имел единственное пространство имен: монолитный пул Smalltalk. Все общие переменные (имена классов, имена глобальных переменных, имена пулов) были определены в пределах единого контекста смолтоковской среды. Соответственно, каждое такое имя должно было быть уникальным.

Всё это прекрасно работало, пока смолтоковские среды использовались либо для создания приложения одним разработчиком, либо для создания приложения, изолированого от других приложений. Поскольку Смолток стал использоваться как инструмент при разработки приложений груп-

пой программистов, роскошью изоляции и единовластного управления средой пришлось пожертвовать.

В версии VisualWorks-7.4.1 универсальное имя Smalltalk сохранено как имя суперпространства имён. Таким образом, ранее единая смолтоковская система разделяется на несколько пространств имён, каждое из которых является контекстом с не конфликтующими именами. Новые пространства имён могут определяться не только в пределах пространства Smalltalk, но и в пределах любого из уже существующих пространств. Поэтому, работая над приложениями, можно, сначала определять все нужные классы в пространстве имён Smalltalk и работать так, как будто множества пространств имён и не существует. Но, как правило, при создании приложения создается новое пространство имён, в котором располагается классы и переменные приложения. Поэтому, для профессиональной работы в среде VisualWorks, знакомство с новшествами, связанными с введением пространств имён жизненно необходимо!

При переходе от единственного пула к множеству пространств имён в смолтоковскую среду VisualWorks изменения надо было внести так, чтобы сохранить преемственность и синтаксис языка Смолток.

Прежде всего, должно измениться определение класса: класс теперь должен определяться сообщением, посылаемым пространству имён, а не его суперклассу. Суперкласс класса должен указываться как часть определения класса.

Глобальные переменные ранее создавались явно, добавлением соответствующих объектов в системный словарь Smalltalk с уникальным именем в качестве ключа. Будучи глобальным, они были доступны всем объектам системы. Пул содержал совместно используемые многими классам переменные, а сам пул определялся глобально. Однако, чтобы обратиться к содержавшимся в нем переменным, пул должен был явно «импортироваться» в класс при его определении. Наконец, переменные класса обеспечивали значения, совместно используемые данным классом, его экземплярами, его подклассами и экземплярами этих подклассов. Такие переменные очень похожи на глобальные переменные за исключением того, что возможная ссылка на них была ограничена указанными классами и экземплярами.

Глобальные переменные, переменные пула и переменные класса имеют это общее, что все они совместно используются (разделяются) различными объектами системы, а их значения логически не зависят от любого отдельного взятого объекта среды. Поэтому имеет смысл объединить их в единый тип объектов, назвать **разделяемыми переменными** (shared variables) и переместить каждое из их из пула Smalltalk в соответствующее пространство имён (см. раздел 3.5).

Чтобы реализовать механизм пространств имён и получить возможность именовать разделяемые переменные, нужны новые объекты. Прежде всего — связывания (bindings), которые являются экземплярами класса VariableBinding (СвязываниеПеременной) или его подкласса InitializedVariableBinding(СвязываниеПерем

Экземпляр класса VariableBinding хранит связанное с ним имя и используется для представления в среде «глобальных» или «полу глобальных» переменных, к которым можно обращаться непосредственно из метода, но которые не являются локальными ни для метода, ни для объекта-получателя. Синтаксически эту возможность обеспечивает точечная нотация — квалификация имени разделяемой переменной именем пространства имён, располагаемым через точку впереди имени разделяемой переменной, а по существу — связывающие ссылки — формируемые на основании точечного имени экземпляры конкретных подклассов BindingReference и LiteralBindingReference класса GeneralBindingReference. В среде с пространствами имён, это позволяет ссылаться на разделяемые переменные, не делая никаких предположений относительно того, в каком пространстве имён содержатся их определения.

3.2. Пространство имён и его содержимое

Пространство имён — это контекст, в пределах которого определена ссылка на объект. Каждая такая ссылка в своем пространстве имён уникальна. Пространство имён само является именуемым объектом, представляющим совокупность разделяемых переменных. Поэтому само пространство имён является значением разделяемой переменной, определенной в другом пространстве имён. Конкретное пространство имён, называемое Root (Корень), является родительским окружением для всех других пространств имён, и позволяет формировать иерархию пространств имён. Иерархия пространств имён в VisualWorks строится по принципу наследования. Характер наследования устанавливается при определении нового пространства имён.

Корневое пространство имён первоначально содержит две разделяемые переменные: Root, значение которого — само пространство имён Root, и Smalltalk, значение которого — пространство имён Smalltalk.

Чтобы увидеть структуру пространств имён в образе среды VisualWorks, достаточно выполнить выражение Root inspect, открывая окно инспектора на содержимом пространства имён Root. При перемещении вниз по пространству Smalltalk, открываются дополнительные разделяемые переменные этого пространства имён, и среди них — пространства имён определенные непосредственно в пространстве Smalltalk. В практических целях, для всех других пространств имён, содержащих смолтоковские определения, иерархия начинается именно с пространства имён Smalltalk.

Приведем фрагмент иерархии пространств имён среды VisualWorks:

```
Root
    Smalltalk
        Core
        Kernel
        OS
            IOConstants
        External
        Graphics
            SymbolicPaintConstants
            TextConstants
        UI
        Tools
        CraftedSmalltalk
        XProgramming
            SUnit
```

Новые пространства имён со смолтоковским кодом должны содержаться в пространстве Smalltalk, либо непосредственно, либо косвенно, но не в Root. Новые пространства имён верхнего уровня, определяемые непосредственно в пространстве Smalltalk, должны иметь в пределах этого пространства уникальные имена. Разработчики среды VisualWorks и другие поставщики пакетов зарезервировали несколько имён верхнего уровня, чтобы уже на этом уровне избежать конфликта имён (их список можно найти на сайте http://www.cincomsmalltalk:8080/).

Примером пространства имён вне пространства Smalltalk могло бы быть пространство, которое содержит программный продукт, поддерживающий разработку и выполнение в пределах смолтоковского образа программ на другом языке программирования, например, на языке Джава (Java). Такая программа могла бы создать пространство имён в Root, например, с именем JavaWorld и содержащимися в нем другими пространствами имён. Конечная иерархия пространств имён могла бы выглядеть примерно так:

```
Root
    Smalltalk
    JavaWorld
        java
            lang
            awt
        COM
            sun
            microsoft
```

При работе в пространстве Smalltalk, поиск ссылки на именованные объекты начинается в пространстве Smalltalk, а не в пространстве Root. Практически, пространство Root можно проигнорировать. Если нужно обратиться к пространству имён Root из пространства Smalltalk то это можно сделать так: Root.Smalltalk.Root. Циклический вызов выглядит пугающе, но такое возможно, поскольку в пространстве Smalltalk определена разделяемая переменная Root, которая ссылается на пространство имён Root. Поскольку предполагается, что начальная часть пути всегда Root.Smalltalk, можно сократить ссылку до Root.

3.3. Ссылка на объекты и импорт

Объекты вновь создаваемых приложений, которые будут располагаться в собственном пространстве имён, должны ссылаться на множество объектов в пространствах имён VisualWorks, и, возможно, на объекты других разработчиков. В пределах естественного окружения ссылка на разделяемый объект происходит, используя его неквалифицированное имя — только имя самого объекта. Для ссылки на разделяемую переменную из другого пространства имён используется точечная нотация, которая производит связывание разделяемой переменной, описывая путь по иерархии пространств имён от пространства Smalltalk до пространства разделяемой переменной. Например, полная ссылка, устанавливающая связывание с системной константой Bold (определенной в пространстве имён TextConstants) такова:

```
Root.Smalltalk.Graphics.TextConstants.Bold
```

Однако, практически, система VisualWorks, когда производит синтаксический анализ точечного имени, по умолчанию считая Root.Smalltalk. начальной частью имени. Поэтому, на практике, ссылка выше сокращается до ссылки Graphics.TextConstants.Bold.

Ссылаться на каждый разделяемый объект явно, описывая путь в иерархии пространств имён от Root или Smalltalk до текущего пространства объекта, неудобно. Вместо этого, при определении нового пространства имён и класса предпочтительнее произвести импорт нужных связываний в про-

странство имён локального объекта, чтобы они могли упоминаться с помощью неквалифицированного имени. Например, определения пространств имён XML и SAX выглядят следующим образом:

```
Smalltalk defineNameSpace: #XML
    private: false
    imports: ' private Smalltalk.*
              XML.SAX.* '
    categoey: 'XML-NameSpace'

Smalltalk.XML defineNameSpace: #SAX      private: false
    imports: 'private Smalltalk.* '
    category: 'XML-SAX'
```

В этих случаях производится общий импорт, использующий звездочку (*) и позволяющий импортировать все связывания, определенные в указанном пространстве имён. В примере, импортируются все связывания из пространств Smalltalk и Smalltalk.XML.SAX.

Обратите внимание, что при импорте перед Smalltalk.* стоит слово private, а перед XML.SAX.* его нет. Если объект импортируется «частным образом» (private), импортированные им связывания не экспортируются этим объектом при его последующем импортировании. Если связывание кем-то определено как частное или импортировано «частным образом», это означает, что оно не должно быть доступно другим пространствам имён при дальнейшем импортировании. И это предписание автора следует уважать!

С другой стороны, если связывание импортируется в объект «публичным образом» (public), то есть отсутствует слово private, то оно далее будет импортироваться в любое пространство имён или класс, который будет импортировать этот объект.

В большинстве случаев, пространство имён должно импортировать содержимое других пространств имён «частным образом». Если пространство имён должно обращаться к некоторым импортированным связываниям в импортируемом им пространстве, то оно должно напрямую импортировать их родное пространство имён. Последнее не является абсолютно жестким требованием, и процесс построения приложения может иногда диктовать другие действия.

В первом примере выше, импортируются все связывания из пространств имён Smalltalk и Smalltalk.XML.SAX. В частности, эти строки импортируют в пространство имён XML все разделяемые переменные, определенные в Smalltalk иSAX. Поскольку SAX импортируется «публично», это делает XML также и экспортером всех импортированных им связываний, так что они далее импортируются любым классом или пространством имен, ко-

торый импортирует пространство XML. В этом случае, это именно то, что нужно, так как, если требуется XML, то требуется и SAX.

Включение ключевого слова private перед Smalltalk.* при его импортировании, запрещает, в частности, XML экспортировать свои связывания. Разумно ожидаться, что они будут импортироваться отдельно каждым заинтересованным в них пространством имён.

Иногда пространство имён или класс должны импортировать не все, а одно единственное связывание из другого пространства имён. Для этого используется импорт конкретного (specific) связывания . Например, пространство имён (ранее, пул) TextConstants должно обращаться только к классу Character из пространства имён Core, поэтому при его определении используется импорт конкретного связывания:

```
Smalltalk.Graphics defineNameSpace: #TextConstants
    private: false
    imports: 'private Core.Character'
    category: 'Graphics-Constants'
```

После этого импортированное имя может использоваться непосредственно, без квалификации пути!

Так же надо поступить, если нужно импортировать из конкретного пространства имён (пула) не все переменные, а только одну или несколько его переменных. Например, если нужно импортировать единственную текстовую константу Bold из пространства имён TextConstants, то строка импорта в определении должна иметь вид

```
imports: 'private Graphics.TextConstants.Bold'.
```

Это позволяет в коде данного пространства имён ссылаться посредством неквалифицированного имени только на разделяемую переменную Bold из пространства имён TextConstants.

Если связывание определено в одном пространстве имён, а затем импортировано в другое, полное точечное имя может определять путь к пространству имён, импортирующему ссылку, вместо пути к родному пространству имён. Так, например, если пространство Smalltalk.MyNameSpace импортирует имя Bold, точечное имя MyNameSpace.Bold является законным точечным именем, по которому достигается разделяемая переменная Bold. Поэтому нет необходимости в ссылке на разделяемую переменную Bold обязательно использовать путь к её родному пространству имён TextConstants.

Поскольку точечное имя вводит путь, начинающийся сразу после пространства имён Smalltalk, точечные имена не поддерживают правила относительного пути, подобного тому, которое используется в файловой системе. Однако, можно сослаться на связывание относительно контекста текущего пространства имён, начиная описание пути со знака подчеркива-

ния и точки (_.). Можно использовать этот подход, например, тогда, когда пространство имён MyNamespace1 импортирует другое пространство имён MyNamespace2, и в MyNamespace2 есть класс MyClass с переменной класса MyVariable. В этом случае любой объект в MyNamespace1 может ссылаться на переменную MyVariable посредством _.MyClass.MyVariable.

3.4. Особенности импорта

Когда говорят об "импортировании пространства имён", обычно подразумевают импортирование содержимого пространства имён, а не только самого имени этого пространства. Содержимое пространства имён может включать в себя:

- определения классов,
- определения других пространств имён,
- определения разделяемых переменных.

Определение пространства имён Smalltalk «публичным образом» импортирует все пространства имён системы для дальнейшего их экспорта, поэтому все эти связывания доступны из пространства Smalltalk:

```
Smalltalk.Root defineNameSpace: #Smalltalk
    private: false
    imports: '
            Core.*
            Kernel.*
            OS.*
            External.*
            Graphics.*
            UI.*
            Tools.*
            CraftedSmalltalk.*
            XProgramming.SUnit.*
            Database.*
            Lens.*
            private VWHelp.*
            '
    category: 'System-Name Spaces'
```

В то же время, каждое из определений этих пространств имён импортирует пространство Smalltalk, но уже «частным образом». Например,

```
Smalltalk defineNameSpace: #Kernel
    private: false
    imports: 'private Smalltalk.*'
```

```
category: 'System-Name Spaces'
```

Таким образом, каждое пространство имён, в свою очередь, импортирует частным образом все связывания из пространства Smalltalk, включая все связывания, которые Smalltalk импортировал из своих подпространств, но не имеет права далее импортировать эти связывания в случае, когда оно само будет импортироваться!

В результате, например, экземпляр класса External.CComposite может ссылаться на класс Core.Array с помощью неквалифицированного имени — просто Array. Таким образом, как и прежде, все основные классы, пространства имён (среди них пулы в смысле классического Смолтока), другие разделяемые переменные, доступны непосредственно из пространства Smalltalk.

Такой подход упрощает перемещение в последующие реализации VisualWorks кода предшествующих версий, гарантируя доступность всех классов системы. Когда код импортируется, он загружается непосредственно в пространство имён Smalltalk, где имеет доступ ко всем необходимым классам системы, и потому может работать с ним без изменения своего кода.

Добавленные в систему VisualWorks компоненты других производителей, которые не импортированы в Smalltalk, должны явно импортироваться создаваемым пространством имён или классом.

Изредка возникает необходимость явно импортировать переменные класса. Они неявно импортируются в класс, в котором определяются, и наследуются всеми подклассами. Так как они используются, чтобы сохранить информацию о состоянии класса, этого достаточно. Если же ссылка на переменную класса нужна вне её окружения, она должна либо импортироваться отдельно, либо ссылка на неё должна описывать соответствующий путь по иерархии пространств имён. Если нужно импортировать переменную класса, её следует импортировать, используя импорт конкретного связывания, как и в случае импортирования переменной пула.

Следует отметить тот факт, что при импортировании самого класса его переменные класса не импортируются. Класс — не пространство имён! Хотя в некоторых ситуациях класс может выступать в роли пространства имён. Одно из основных различие между классом в роли пространства имён и пространством имён состоит в том, что классы могут содержать только разделяемые переменные, являющиеся его переменными класса, которые в качестве значения не могут иметь пространство имён или другой класс.

3.5. Разделяемые переменные *VisualWorks* 7.4.1

Значением разделяемой переменной среды VisualWorks 7.4.1 может быть пространство имён, или класс, или любой другой объект. В последнем слу-

чае, разделяемые переменные играют роль, которую прежде играли глобальные переменные, переменные пула и переменные класса.

Разделяемая переменная, когда она определяется классом, заменяет реализацию переменной класса. Класс в этом случае служит пространством имён для такой разделяемой переменной. Как переменная класса такая разделяемая переменная наследуется, и доступна для самого класса его экземпляров, его подклассов и их экземпляров. Это верно, даже если классы находятся в разных пространствах имён! Поэтому в этом случае явного импортирования не требуется.

Например, класс Date имеет разделяемую переменную (переменную класса) с именем MonthNames, которая хранит массив, содержащий имена всех 12 месяцев. Однажды определенный в разделяемой переменной массив доступен всем экземплярам класса Date и его подклассов, а также экземплярам любого другого класса, который импортирует разделяемую переменную MonthNames.

Разделяемые переменные могут определяться непосредственно в любом пространстве имён (не только в классе). Например, в пространстве имён Graphics определено много классов, и еще два пространства имён SymbolicPaintConst и TextConstants. Эти пространства имён (ранее, пулы) существуют исключительно как окружения видимости для наборов специальных разделяемых переменных (ранее, переменных пула). Каждая разделяемая переменная такого пространства имён определяется и инициализируется непосредственно в своем пространстве имён. Чтобы все эти переменные стали доступны в пределах другого пространства имён, соответствующее пространство имён (пул) должно быть импортировано туда операцией общего импорта.

Глобальные переменные в смысле классического Смолтока — пример плохой практики программирования в объектно-ориентированной среде, поскольку они нарушают принцип инкапсуляции. Вместо глобальных переменных, в VisualWorks 7.4.1 определяются разделяемые переменные в том пространстве имён, которое почти всегда доступно для всех других пространств. Например, ранее глобальная переменная Transcript определена как разделяемая переменная в пространстве имён Smalltalk.Core.

Разделяемые переменные теперь наделены свойствами. Можно установить значение разделяемой переменной и далее не изменять его (сделать переменную константой). При создании или импортировании, разделяемую переменную можно объявить как общую, или как частную. Если переменная определяется как общая (public), она доступна для импортирования пространством имён или классом. Если же она определяется как частная (private), то становится недоступной для импорта пространством

имён или классом.

Пространства имён и классы обычно определяются как общие, так как они должны импортироваться пространствами имён, которые должны обращаться к ним. Переменная пула также должна определяться как общая, так как она также должна импортироваться. Переменные класса, разделяемые переменные, которые определены в пределах окружение класса, также, обычно, определяется как общие, так что они должны быть доступны подклассам класса и их экземплярам. Определение пространства имён, класса или разделяемой переменной как частной сущности — исключение, но иногда такое требуется.

Когда разделяемая переменная удаляется, но, в то же время, ссылки на неё всё ещё существуют, или ссылка на переменную загружается (пакетом или модулем), но переменная не объявляется, её имя перемещается в пространство имён Undeclared. Это пространство имён поддерживается системой и в нормальных условиях не требует внимания со стороны программиста. Но знакомство с ним может оказаться полезным для выяснения причин некоторых видов ошибок в программе. Чтобы просмотреть содержимое пространства имён Undeclared, нужно его открыть в системном браузере как любое другое пространство имён или открыть на нём окно инспектора (Undeclared inspect).

Переменные добавляются в пространство Undeclared когда:

- во время загрузки из файла командой File In... (или по другим причинам) компилируется ссылка на несуществующую переменную;
- переменная удаляется, хотя на неё всё ещё существуют ссылки;
- удаляется класс (независимо от того, существуют ли извне ссылки на него или нет); такой механизм гарантирует, что любые внешние ссылки, которые всё ещё могут существовать, будет правильно восстановлены, если класс будет создан снова.

3.6. Контрольные вопросы

1) Что такое пространство имён?
2) Что представляет собой пул Smalltalk в реализациях VisualWorks без пространств имён и с пространствами имён?
3) Какое пространство имён является корнем всей иерархии пространств имён, а какое корнем для пространств имён смолтоковского кода?
4) Что такое разделяемая переменная? Что может быть значением разделяемой переменной?
5) Что значит объявить разделяемую переменную как общую или как частную?

6) Что такое точечная нотация?
7) Что такое импорт пространства имён? Какие два вида импорта пространства имён существуют в VisualWorks и в чём их различие?
8) Что такое импорт конкретного связывания?
9) Что представляет собой пространство имён Undeclared?

Глава 4

Пакеты и парселы

Пространства имён — языковая поддержка разработки кода разными программистами, позволяющая избежать конфликта имён.

Пакеты (категории) и связки пакетов — не поддерживаемые языком Смолток, но поддерживаемые инструментами среды VisualWorks 7.4.1, структурные единицы кода, позволяющие изолировать код одного приложения от кода других приложений в рамках одного образа.

Парсел — внешнее представление пакетов и связок пакетов в виде файлов, что позволяет передавать смолтоковский код вне образа среды от одного разработчика другому и конечному пользователю.

Общую схему разработки приложения в VisualWorks можно описать следующим образом. Код приложения разрабатывается в пространствах имён и классах, сохраняется в образе системы и/или во внешних файлах с расширеием *.st командой File Out As... . По мере создания приложения, код, содержашийся в пространствах имён и классах объединяется в пакеты и связки пакетов, затем, по завершении разработки, окончательная версия сохраняется во внешних файлах, парселах.

Следует отметить, что наиболее совершенные средства управления исходным текстом приложений, отслеживающим последовательности версий, обеспечивает пакет StORE, описание которого выходит за рамки этого учебника (см. [10]).

4.1. Пакеты и связки пакетов

Пакет (или категория) — первичная организационная структура кода в среде VisualWorks, включающая определения классов, методов, пространств имён и совместно используемых переменных. Каждое определение в образе VisualWorks связывается с конкретным пакетом или со специальным пакетом (none). Пользуясь пакетами, можно хранить создаваемый код отдельно от кода основной библиотеки среды, дополнений и программных продуктов

других разработчиков. В рамках создаваемого приложения, можно использовать пакеты, чтобы разделять структурные блоки кода приложения.

Связки пакетов позволяют организовать хранение пакетов на более высоком уровне. Используя связки пакетов, можно представить создаваемый код в виде небольших пакетов, содержащих тесно связанные между собой компоненты, а затем объединить эти пакеты в большие связки, содержащие функционально родственные пакеты приложения.

Например, в системе есть очень большая связка пакетов, именуемая Base VisualWorks. Её пакеты и связки пакетов содержат родственные части основного кода системы. Например, связка Kerner содержит пакеты, которые определяют основные функциональные возможности Смолтока, такие как построение классов и их поведение. Связка пакетов Tools содержит пакеты инструментов, отделяя их от основных функциональных возможностей среды.

Вместе, пакеты и связки пакетов обеспечивают мощный и гибкий механизм разбиения кода на небольшие, легко обозримые и понятые модули. Пакет может содержать всё приложение или только единственное его определение. Однако, чаще всего полное приложение представляется в виде связки пакетов или иерархии связок и небольших пакетов.

4.2. Парселы

Парселы — одна из составляющих технологии поставки конечному пользователю программного продукта в виде файлов, которая обеспечивает механизм быстрой и корректной загрузки объектов в смолтоковский образ. Все дополнения среды VisualWorks поставляются как парселы, которые загружаются в образ.

Каджый парсел хранится в двух файлах. Файл парсела с расширением *.pcl, содержит оттранслированную программу в бинарном формате, а файл парсела с расширением *.pst содержит соответствующий исходный текст. Несмотря на поверхностное сходство между pst-файлами и файлами с расширением *.st, записываемых командой File Out As..., pst-файлы нельзя загружать командой File In.... Это только файлы исходного кода для соответствующего pcl-файла!

Из внешних файлов — парселов, или st-файлов — код загружается в VisualWorks, при загрузке распределяется по пакетам, и таким образом вводится в структуру среды. Загрузку и удаление (выгрузку) парселов из среды можно провести с помощью инструментов VisualWorks: Parcel Manager (см. раздел 5.2), страницы Parcel системного браузера (см. 6.4), и специальных инструментов быстрой загрузки и выгрузки парселов, доступных в основном окне среды VisualWorks по командам меню System → Load Parcel

Named... и System → Unload Parcel Named..., соответственно.

Но, если нужно, парселы можно загружать и выгружать программно. Например, когда пользователь запускает новый инструмент или открывает новое окно в приложении, приложение может загрузить парсел, содержащий этот инструмент или окно. Следующая строка кода программы загружает парсел из файла `UIPainter.pcl`:

```
Parcel loadParcelFrom: '..\parcels\UIPainter.pcl'.
```

Точно так же, когда парсел больше не нужен, чтобы его выгрузить из среды, достаточно выполнить следующий код:

```
Parcel unloadParcelNamed: 'UIPainter'.
```

Отметим различие между этими двумя сообщениями. Чтобы загрузить парсел, нужно указать имя его pcl-файла; чтобы выгрузить парсел, нужно указать имя парсела. Эти два имени могут отличаться.

Кроме этого, нужные изначально парселы можно загружать при запуске системы VisualWorks с помощью опций командной строки (см. [9, p. 8-29 – 8-30]).

Работа с парселами имеет некоторые особенности.

Частичная загрузка. Парселы поддерживают механизм частичной загрузки определений из парсела. Если парсел содержит класс, требующий суперкласса, которого нет в системе, или метод, требующий класса, которого нет в системе, класс или метод не устанавливаются. Вместо этого эти классы и методы добавляются в набор uninstaliedClasses или uninstaliedeMethods, соответственно. Когда парсел с неинсталированным кодом загружается вновь, парсел проверяет, был ли загружен отсутствовавший объект. Если он был загружен, парсел загружает ранее незагруженный код.

Сохранение парсела с неинсталлированным кодом приведет к потере такого кода. Диалоговое окно сообщит об этом, и, чтобы не потерять код, операцию сохранения следует отменить.

Восстановление кода при выгрузке. Парселы запоминают методы и классы, которые они заменяют при загрузке, и восстанавливают их при выгрузке данного парсела из образа.

Действия при сохранении, загрузке и выгрузке. При выполнении каждой из операций парселы могут выполнять предварительные действия до операции и заключительные по завершении операции. Эти действия первоначально определяются для пакетов и связок, а затем переназначаются на парсел, когда он создаётся.

Предпосылки и автозагрузка. Парсел может включать имена предварительно загружаемых парселов, которые автоматически загружаются, когда загружается требующий их парсел.

Устойчивость к изменению формы. Изменение формы относится к

тем переопределениям классов, при которых добавляются или удаляются переменные экземпляра, или экземпляр класса теряет или приобретает индексированные переменные, удаляются или вводятся ограничения на классы хранимых индексированными переменными объектов. Это приводит к тому, что экземпляры классов приобретают или теряют поля переменных, или, другими словами, изменяют свою форму.

Парселы поддерживают механизм замены формы экземпляров и методов, который пытается откорректировать объекты таким образом, чтобы они могли загружаться. Если, например, парсел загружает объект, число переменных экземпляра которого изменилось, он назначает те же значения переменным с теми же именами, отказывается от значений пропавших переменных, и устанавливает значения новых переменных равными nil. Система не может пока справляться с изменениями формы, отличными от добавления или удаления именованных переменных экземпляра. Если форму класса меняют два определения класса, то «победит» последнее загруженное определение. Новое определения переписывает старое, а не сливается с ним, таким образом, например, переменные экземпляра из двух переопределений не объединяются.

Парселы не требуют отдельного управления. Но следует тщательно определять их содержание, разумно организуя создаваемый код в пакеты и связки пакетов. Затем, чтобы создать парсел, следует опубликовать (издать, сохранить на внешнем носителе) пакет или связку пакетов как парсел.

Обычно парсел не приходится создавать явно, поскольку он создаётся при сохранении (публикации) пакетов и связок пакетов.

4.3. Особенности операций File Out As... и File In...

В связи с введением пространств имён и пакетов загрузка файлов исходного текста командой File In... в VisualWorks 7.4.1 приобретает некоторые особенности, а именно, поведение при загрузке меняется в зависимости от происхождения файла.

- Если файл исходного текста сохранялся командой File Out As... в версии VisualWorks ранее 5.i, в которой других пространств имён кроме Smalltalk не было, то командой File In... код будет загружаться в пространство имён Smalltalk. Будут соответствующим образом исправлены определения классов.
- Если файл сохранялся командой File Out As... в версии VisualWorks ранее 7.3, или из смолтоковской среды, отличной от VisualWorks, командой File In... код по умолчанию будет записываться в стандартный пакет (none). Затем надо будет явно определить новые пакеты, и переместить туда классы (см. раздел 6.2).

- Если файл записывался командой File Out As... в версии 7.3 или в более поздней версии, командой File In... код по умолчанию записывается в пакет с именем оригинального пакета, которое, обычно, совпадает с именем файла.

4.4. Контрольные вопросы

1) Что такое пакеты и связки пакетов?
2) Что такое специальный пакет (none) и как он используется?
3) Какие особенности возникают в работе команды File In... в связи с введением пакетов?
4) Что такое парсел? Какие два файла хранят парсел?
5) Какие инструменты позволяют работать с пакетами и парселами в среде разработки?
6) Какой инструмент позволяет производить загрузку и удаление (выгрузку) парселов из образа?
7) Как можно производить загрузку и удаление (выгрузку) парселов программно?
8) Какие особенности существуют при работе с парселами?

Глава 5

Основные инструменты

В этой главе рассматриваются основные инструменты среды VisualWorks 7.4.1, за исключением системного браузера, которому посвящена следующая глава. Рассматриваемые инструменты присутствуют практически в каждой смолтоковской среде, но в VisualWorks 7.4.1 они имеют некоторые особенности.

5.1. Рабочее окно

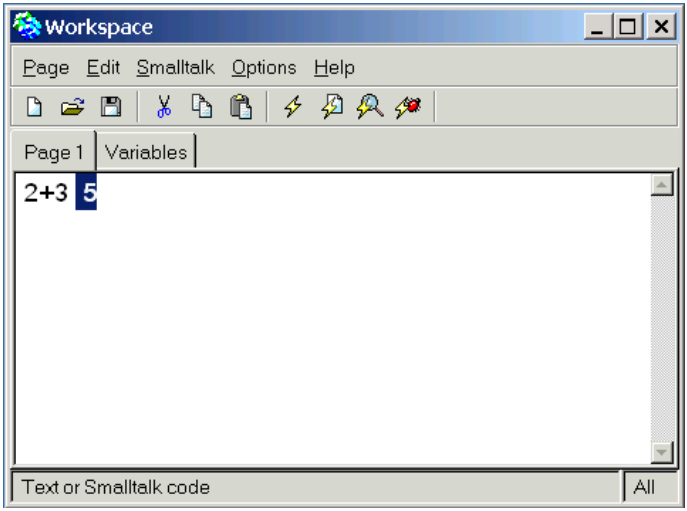


Рис. 5.1: Рабочее окно системы VisualWorks.

Рабочее окно (Workspace) — инструмент, позволяющий выполнять смолтоковский код. Рабочие окна также полезны, например, для хранения вы-

ражений, готовых к выполнению с целью тестирования различных особенностей разрабатываемого приложения. При первом запуске инсталлированной системы VisualWorks открывается окно Workspace с информацией о системе.

Чтобы открыть рабочее окно, надо выбрать команду Tools → Workspace или щелкнуть на иконке Workspace в панели инструментов основного окна.

Рабочее окно — многостраничный инструмент, каждая страница которого содержит независимое от остальных содержание. Страница с меткой Variables, отображает переменные рабочего окна и их значения.

Создать и удалить страницу позволяют команды меню Page → New и Page → Remove, соответственно. Многостраничная структура позволяет иметь одновременно открытыми несколько страниц рабочего окна, которые совместно используют одни и те же переменные.

От рабочего окна можно «оторвать» страницу (команда меню Page → Tear Off), делая ее отдельным рабочим окном с единственной страницей.

Выполнение кода в рабочем окне

Рабочее окно — основное окно для отладки смолтоковского кода. Когда в рабочее окно или в любую другую текстовую панель, вводится смолтоковское выражение, появляется возможность его выполнить. Для этого следует, либо выбрать выражение, либо, когда надо выполнить целую строку, разместить в ней где-нибудь курсор, а затем, используя, либо всплывающее меню операций текстовой панели, либо меню Smalltalk, либо соответствующую кнопку панели инструментов, выбрать одну их следующих команд:

Do it — выполняет выбранное выражение. Любой вывод — ответственность самого выполняемого выражения.

Print it — выполняет выбранное выражение и печатает возвращаемое им значение в рабочем окне.

Inspect it — выполняет выбранное выражение и открывает окно инспектора на возвращаемом им значении.

Debug it — выполняет выражение и открывает отладчик на первом посылаемом сообщении.

Последняя команда подобна размещению в коде выражения self halt и его выполнению. Далее можно использовать команды Step into и Step, что-

бы исследовать выполнение сообщений в коде. Обычно, используют команду Step into, чтобы выполнить первое посылаемое сообщение, а затем используют Step, чтобы пройти по коду сверху вниз (см. главу 7).

Например, чтобы отобразить текст 'Hello, world!' в окне Transcript, откроем рабочее окно и введем выражения:

```
Transcript show: 'Hello, world!' printString.
Transcript cr.
```

Затем выберем весь текст (используя кнопку <Select>), нажмем кнопку мыши <Operate> и, не отпуская её, когда курсор находится в текстовой панели рабочего окна, переместим курсор на строку Do it всплывающего меню и только тогда отпустим кнопку мыши: текст отобразится в окне Transcript.

Переменные рабочего окна

Переменные рабочего окна — временные переменные, используемые в рабочем окне, для которых окно служит окружением[1]. Эти переменные существует, пока существует рабочее окно, или пока они явно не будут удалены. Эти переменные создаются, когда им первоначально назначается значение. Данное назначение сохраняется и может впоследствии использоваться в выполняемом выражении этого рабочего окна. Переменная и её значение сохраняются вместе с образом, и потому доступны при последующем запуске образа.

Например, определим переменную, которая будет указывать на массив, просто выполняя операцию назначения: fred := Array with: 5. Переменная fred теперь доступна для дальнейших операций. Например,

```
fred at: 1 put: 'this is a test'.
```

Чтобы просмотреть, удалить или отредактировать переменную рабочего окна, следует перейти на страницу Variables, которое является окном инспектора на текущих переменных этого рабочего окна. Остается выбрать переменную или ее значение и воспользоваться командами из всплывающего меню Operate соответствующей панели.

Пространства имён в рабочем окне

Рабочие окна могут импортировать пространства имён, позволяя им лучше моделировать окружения имён выполняемого кода. Без импортирования, имя окружения по умолчанию — Smalltalk, так что для того, чтобы сослаться на разделяемую переменную, такую как имя класса вашего прило-

[1] В версии 3.0 такой возможности нет. Временные переменные рабочего окна надо определять явно.

жения, если его нет в пространстве Smalltalk, следует использовать длинное имя с точкой.

По умолчанию, и ради удобства, рабочее окно позволяет импортировать все пространства имён системы, что позволяет обращаться ко всем разделяемым переменным по их неквалифицированным именам. Для лучшего моделирования окружений имён, можно определить и импортировать только некоторые пространства имён.

Чтобы определить выбор импортируемых пространств имён, следует выбрать из меню рабочего окна команду Smalltalk → Namespaces.... В появившемся диалоговом окне выбрать пункт All или Some. Если выбрано Some, выбрать те пространства имён, которые следует импортировать. Импортированное пространство имён отмечается галочкой.

Сохранение содержимого рабочего окна

Каждая страница рабочего окна может сохраняться в собственном текстовом файле, а позже открыть его. Обратите внимание, что переменные рабочего окна не сохраняются в файле, так что их придется восстановить (выполнить выражения, присваивающие им значения), когда файл будет снова открыт в рабочем окне.

Чтобы сохранить содержимое рабочего окна в файле, надо выбрать команду из всплывающего меню рабочего окна Page → Save или Page → Save As.... Если рабочее окно не было ранее сохранено, возникнет запрос относительно имени файла с расширением ws.

Чтобы загрузить сохраненное ранее содержимое рабочего окна из файла, надо открыть новое рабочее окно, затем выбрать команду Page → Open, и ввести имя файла рабочего окна.

5.2. Администратор парселов

Многие особенности среды, ранее входившие в основной образ, теперь помещены в парселы (`pcl`-файлы и `pst`-файлы). Например, UIPainter — инструмент разработки графического интерфейса пользователя (GUI), ранее загружался в стандартном образе. Теперь его там нет и, если он нужен, его следует загрузить в образ. Однако, его не нужно загружать, чтобы запускать приложения с GUI. Все необходимые инструменты находятся в ядре. Такой тип разделения и хранения части среды в `pcl`-файлах, позволяет сделать образ небольшим и упрощает поставку приложения конечному пользователю, позволяя загружать только нужные компоненты.

Когда в систему предстоит загружать `pcl`-файл, он обычно отображается в образе как пакет (package) или связка пакетов (bundle of packages). Когда нужно просмотреть код, загруженный из `pcl`-файла, следует найти

связку или пакет с таким же именем и исследовать его. Администратор парселов Parcel Manager (см. рисунок 2.8) позволяет просматривать, загружать и выгружать парселы, используя пути поиска парселов, которые определяются на странице Parcel Path окна установок среды. Напомним, что это окно открывается командой меню System → Settings основного окна среды.

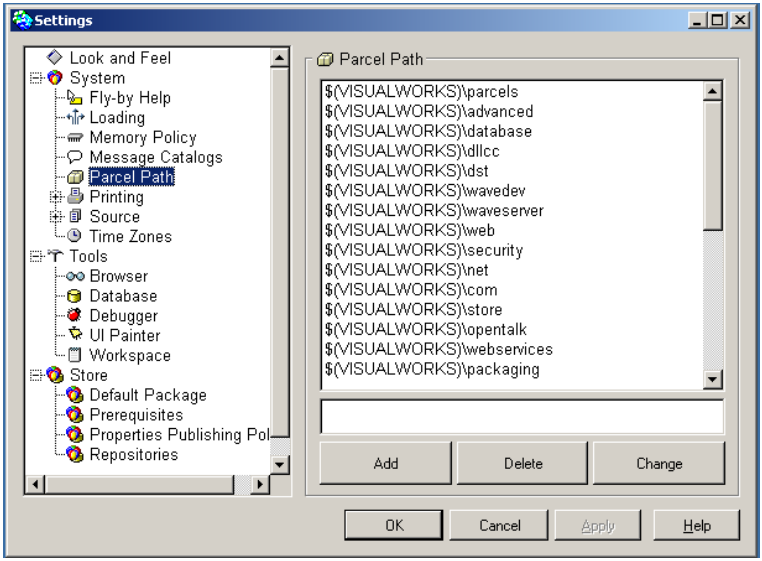


Рис. 5.2: Установка путей поиска парселов.

Пути поиска парселов можно менять. Чтобы добавить новый путь в список, надо ввести его в текстовой панели и щелкнуть на кнопке Add. Префикс $(VISUALWORKS) соответствует каталогу инсталляции VisualWorks. Можно не пользоваться префиксом, а определять полный путь. Чтобы изменить порядок поиска, следует выбрать путь и перетащить его вверх или вниз. Каталоги просматриваются сверху вниз. Чтобы удалить каталог из пути поиска, надо его выбрать и щелкнуть на кнопке Delete. Чтобы отредактировать путь, надо его выбрать, отредактировать в текстовой панели, и щелкнуть на кнопке Change. Когда изменения в списке поиска пути закончены, надо щелкнуть на кнопке Accept.

Пути поиска парселов сохраняются вместе с образом. Кроме того, их можно сохранить в отдельном файле, для чего сначала выбрать в панели установок строку Parsel Path, затем выбрать команду Save Page... из всплывающего меню панели установок, в открывшемся диалоговом окне определить имя и каталог для файла и, наконец, щелкнуть на кнопке Save.

Сохраненный файл можно загрузить в среду, используя команду меню Load Page. . . .

Чтобы открыть окно администратора парселов, следует в основном окне выбрать команду меню System → Parcel Manager или щелкнуть на иконке администратора парселов в панели инструментов.

Над левой списковой панелью расположеные три ярлыка страниц Suggestions, Directories, Loaded.

Страница Suggestions позволяет увидеть предопределённый набор парселов. Каждый элемент страницы Suggestions содержит парселы, которые были идентифицированы как ключи добавляемых в образ VisualWorks сущностей. Выбирая конкретную категорию, можно просмотреть список рекомендуемых к загрузке парселов — он отображается в правой верхней панели. Страница Alphabetical правой панели позволяют увидеть список парселов, отсортированных в алфавитном порядке, а страница Prerequisite Tree — упорядоченных в иерархическую структуру, указывающую порядок их загрузки в образ. Правая нижняя панель отображает комментарий и свойства выбранного парсела. Например, парсел UIPainter, содержащий код основного инструмента VisualWorks для разработки GUI, расположен в категории Essentials. При его загрузке предварительная загрузка других парселов не требуется.

Выбирая страницу Directories, можно просматривать все пути к парселам в дереве каталогов. Выбор конкретного каталога отобразит все парселы, включенные в него.

Выбор страницы Loaded, позволяет увидеть все загруженные в образ парселы. Чтобы удалить парсел из образа (выгрузить его) следует выбрать его из списка парселов (в правой верхней панели), а затем выбрать команду Unload из всплывающего меню операций. Можно также воспользоваться ставшей доступной кнопкой выгрузки парсела в панели инструментов или дважды щелкнуть на имени парсела.

Чтобы загрузить парсел, следует выбрать его из списка парселов (в правой верхней панели), а затем выбрать команду Load из всплывающего меню операций. Можно также воспользоваться ставшей доступной кнопкой загрузки парсела в панели инструментов или просто дважды щелкнуть на имени парсела.

Чтобы просмотреть парсел, который уже загружен в образ, следует его выбрать и воспользоваться командой Browse из меню операций или ставшей доступной кнопкой просмотра парсела в панели инструментов, открывая браузер системы на этом парселе.

В окне администратора парселов используются специальные значки, позволяющие отличить парселы от других программ:

«Полный перевязанный мешок» — родные парселы, загруженные в среду.

«Полная сумка с ручками» — парселы других производителей, загруженные в среду.

«Пустой перевязанный мешок» — родные парселы, незагруженные в среду.

«Пустая сумка с ручками» — парселы других производителей, незагруженные в среду.

В дополнение к использованию инструмента Parcel Manager загрузку и выгрузку парселов можно производить программно, что позволяет выполнять загрузку и выгрузку парселов приложением (см. раздел 4.2).

Некоторая часть возможностей по работе с парселами представлена в системном браузере (см. главу 6).

5.3. Браузер файлов

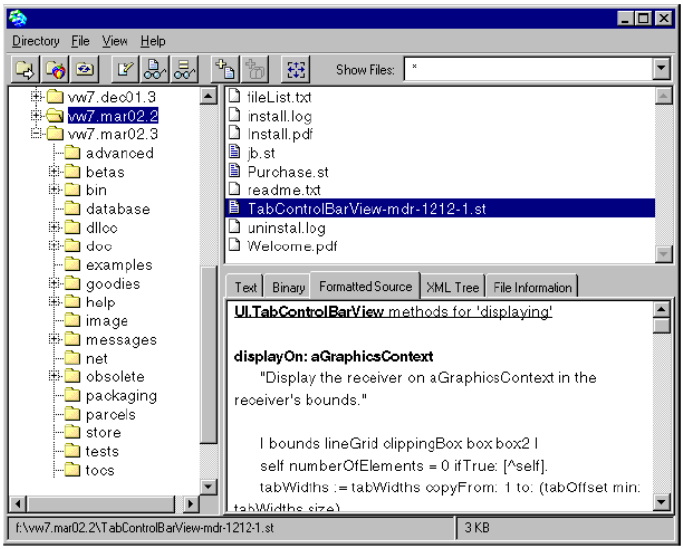


Рис. 5.3: Окно браузера файлов.

Инструмент File Browser (Браузер файлов) позволяет перемещаться по файловой системе, просматривать и выбирать каталоги и файлы. Он обычно используется для поиска файлов с расширением к `*.st`, их загрузки в

среду командой File In. . . , а так же для редактирования простых текстовых файлов.

Чтобы открыть окно браузера файлов, следует выбрать команду меню File → File Browser или щелкнуть на соответствующей иконке в панели инструментов основного окна.

Когда в левой панели браузера выбирается каталог, в верхней панели справа отображаются содержащиеся в нем файлы. Когда выбирается файл, в правой нижней панели отображается его содержание. Правая нижняя панель позволяет просматривать файлы разных форматов (надо только воспользоваться соответствующей страницей). Здесь можно просматривать файлы смолтоковского кода (.st), парселы, файлы исходного текста парселов, файлы исходного текста в формате XML.

5.4. Инспекторы

Инспектор позволяет исследовать (инспектировать) любой объект, просматривая составляющие его объекты, значения его переменных экземпляра. Инспектор включает несколько дополнительных инструментов для редактирования, а также разнообразные команды, позволяющие выполнять многочисленные операции над объектом, не открывая других инструментов системы. Традиционное окно инспектора имеет две панели. Левая панель на странице Basic отображает переменные объекта. Когда в ней выбирается переменная, её значение отображается в правой панели.

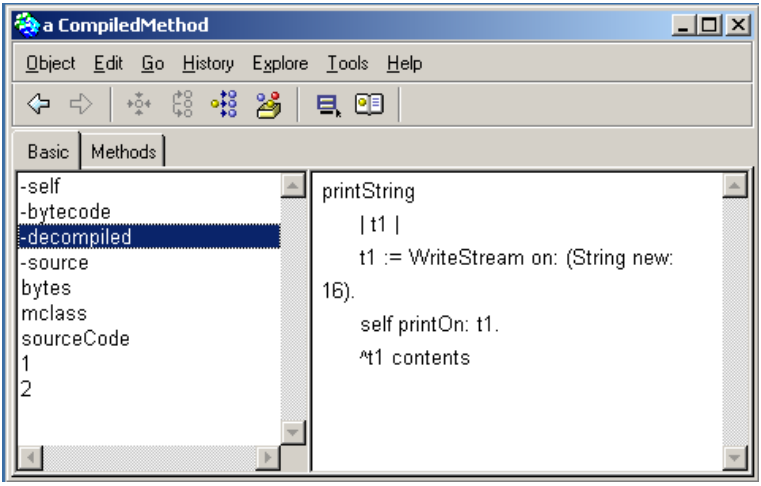


Рис. 5.4: Окно инспектора на скомпилированном методе.

Можно инспектировать любой компонент объекта, выбирая его в левой панели и выбирая в всплывающем меню <Operate> данной панели команду Dive, которая отобразит выбранный объект в текущем окне инспектора. Чтобы вернуться назад, надо в меню <Operate> выбрать команду Back или щелкнуть на кнопке со стрелкой влево в панели инструментов инспектора, которая стала активной. Чтобы открыть новое окна инспектора на компоненте объекта, следует его выбрать и выбрать команду Object → Inspect.

Для некоторых объектов, основное окно может отображать дополнительные сущности, которые не являются его переменными экземпляра. При инспектировании откомпилированного метода (см. рис. 5.4)

```
Object compiledMethodAt: #printString) inspect
```

левая панель окна включает строки -bytecode, -decompiled, -source. Они не являются частями объекта (его, переменными), но включены в окно инспектора, как "виртуальные" атрибуты, точно так же, как и элемент self, представляющий непосредственно сам объект в любом окне инспектора.

На элементах можно выполнять операцию «перетащить-и-опустить». Если выбрать одну переменную и переместить (при нажатой кнопке выбора) её поверх другой, то значение первой будет назначено целевой переменной.

Инспекторы для наборов

Специализированные инспекторы для словарей и других наборов обеспечивают расширенные возможности просмотра. Например, в дополнение к операции «перетащить—и—опустить» в нём можно переупорядочить элементы набора: выбрать элемент, перетащить его между двумя другими элементами и опустить.

Если открыть окно инспектора на наборе, определяемом выражением

```
(OrderedCollection with: 1 with: 2 with: 3 with: 4) inspect
```

то откроется окно инспектора, в котором первая списковая панель будет иметь не две, а три страницы (см. рисунки 5.4, 5.5). Страница Elements позволяет инспектировать только элементы набора. Страница Basic — окно стандартного инспектора, которое открывается при посылке набору сообщения basicInspect.

В правой панели окна инспектора изменение значения можно производить, просто вводя нового значения или выполняя смолтоковское выражение. Чтобы установить значение переменной посредством выполнения выражения, его надо ввести в правой панели, а затем выбрать в меню <Operate> панели команду Accept, которая выполнит выражение и назначит возвращаемое значение переменной в окне инспектора. В этом отно-

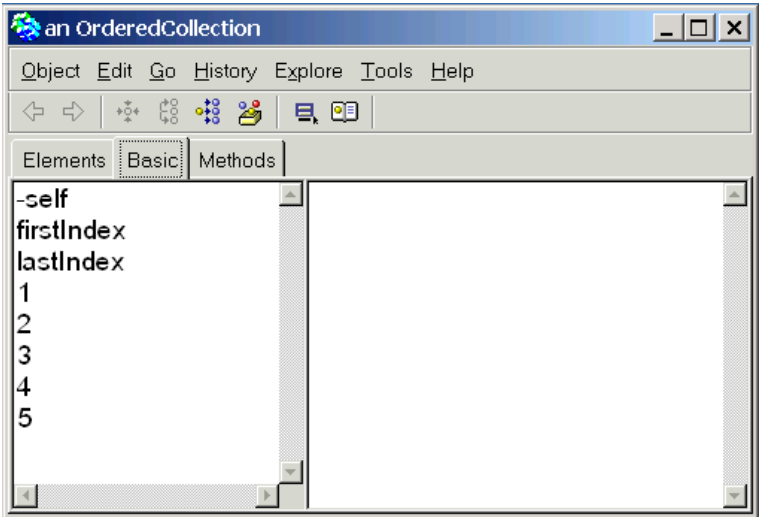


Рис. 5.5: Окно инспектора на наборе.

шении правая панель подобна рабочему окну. Переменная используется в пределах окружения окна просмотра кода.

Введенное в правой панели инспектора выражение теряется после выполнения, или после выбора в левой панели другой переменной. Но есть возможность выполнить код и сохранить выражение, которое было введено. Для этого надо открыть панель выполнения выражений, выбирая команду меню инспектора Tools → Evaluator Pane или щелкая на иконке инструмента Evaluator Pane в панели инструментов (последней иконке). Панель окна для выполнения выражений откроется внизу окна инспектора.

Данная панель работает во многом подобно рабочему окну. Однако, выполняемый контекст — объект в окне инспектора. Соответственно, можно использовать self, чтобы обращаться непосредственно к объекту, и можно выполнять операции на объекте (пользуясь командами Do It и Print It).

Наконец, можно «сохранять» содержимое панели выполнения выражений, делая данное содержимое доступным для всех инспекторов. Содержимое сохраняется в переменной класса инспектора, которая разделяется всеми экземплярами. Чтобы записать содержимое в переменную надо выбрать из меню <Operate> данной панели команду Accept.

Страница Method первой списковой панели инспектора открывает браузер просмотра класса на классе инспектируемого объекта, что позволя-

ет легко изменять поведение объекта, не открывая отдельного браузера на классе или системного браузера. Меню Inheritance, возникающее в этом случае в панели меню инспектора, перечисляет класс и его суперклассы, и, как и в браузере системы, позволяет выбирать глубину наследования методов при их отображении.

Но в окне инспектора некоторые операции выполняются не так, как в стандартных браузерах. Например, нет команды Find Method. Вместо этого, надо сделать множественный выбор категорий методов, чтобы отобразить все методы выбранных категорий. Выбор всех категорий эквивалентен тому списку, который показывает команда Find Method.

5.5. Контрольные вопросы

1) Для чего используется рабочее окно среды VisualWorks, как его открыть, какие страницы есть в нём?
2) Какие команды используются в рабочем окне для выполнения кода?
3) Что представляют переменные рабочего окна?
4) Как взаимдействует рабочее окно с пространствами имён?
5) Как сохранить и восстановить содержимое рабочего окна?
6) Какие основные операции позволяет выполнять с парселами инструмент Parcel Manager?
7) Как определить пути поиска парселов?
8) Как открыть окно администратора парселов? Какие страницы оно содержит? Какие операции с парселами можно выпольнять на каждой из страниц?
9) Как специальные значки, позволяющие отличить парселы от других программ, использует администратор парселов?
10) Для чего используется инструмент File Browser? Как его открыть?
11) Для чего используется инспекторы? Какие специализированные инспекторы существуют в VisualWorks?

Глава 6

Системный браузер

Главный инструмент программирования в VisualWorks, как и в любой смолтоковской среде, браузер системы System Browser (см. рисунок 2.4), который позволяет управлять смолтоковским кодом, располагаемым в иерархии классов. Он предоставляет возможности создания, удаления, редактирования, компилирования и печати любой выбранной части исходного текста. Разные списковые панели браузера отображают различные элементы системы и используют следующие обозначения:


Пакет.


Связка пакетов.


Пространство имен.


Подкласс класса Model.


Подкласс класса ApplicationModel.


Подкласс класса Collection.


Подкласс класса Exception.


Метод переопределяется хотя бы в одном суперклассе.


Метод переопределяется хотя бы в одном подклассе.

Каждая панель в браузере имеет собственное меню операций (<Operate>) и команды всплывающего меню (меню левой кнопки мыши), соответ-

ствующие её содержанию. Вид и содержание панелей браузера меняется в зависимости от решаемых задач.

6.1. Панели системного браузера

Чтобы открыть браузер системы надо в основном окне выбрать команду меню Browse → System или щелкнуть на иконке системного браузера в панели инструментов.

Панель пакетов, парселов и иерархии классов

Верхняя левая панель браузера системы — панель пакетов, парселов и иерархии классов. Начиная навигацию с этой панели, можно, перемещаясь от панели к панели слева направо, просматривать определения классов, пространств имен, методов и переменных.

Используя метки страниц левой верхней панели, в в ней можно просматривать пакеты — страница Package (по умолчанию) или иерархию классов — страница Hierachy. Страница Package отображает классы в виде групп, объединяющих родственные компоненты. Пакеты в этом случае играют ту же роль, что и категории. Страница Hierachy отображает все классы из иерархии классов, от класса Object до выбранного класса, если таковой есть, или все классы иерархии, если такового нет.

Еще одним специальным представлением является страница Parcel, которая становится доступной при выборе в браузере команды меню Browser → Parcel. Она нужна для выполнения такие операций на парселах, которые нельзя выполнить на пакетах или их связках.

Каждая страница панели пакетов, парселов и иерархии классов имеет своё меню операций и команды, соответствующие её содержанию.

Чтобы найти в среде нужную разделяемую переменную (в том числе класс или пространство имён) нужно щелкнуть на иконке поиска разделяемой переменной в панели инструментов, либо воспользоваться командой меню Find → Class/Variable/Name Space. . . , либо соответствующей командой всплывающего меню верхней левой панели. А можно просто ввести имя разделяемой переменной в поле ввода Find, расположенном справа в панели инструментов, затем выбрать команду Accept из меню операций этой панели или нажать клавишу Return. При поиске можно использовать сивол *, создавая шаблон поиска.

Панель классов и пространств имён

Следующая (слева направо) верхняя панель — панель классов и пространств имён. Классы и пространства имён определяются в пакетах, так что содержание этой панели зависит от выбранного пакета.

Когда в левой панели выбирается страница Hierarchy, то панель классов

и пространств имён показывает тот пакет, в котором содержится определение выбранного в иерархии класса. Если же определение класса содержится в нескольких пакетах, отображаются все пакеты, и выбирая пакет, можно просматривать ту часть определения класса, которая храниться в выбранном пакете.

Панели методов и переменных

Следующие две панели — панель категорий методов и категорий переменных, панель методов и переменных — тесно связаны между собой и представляют своеобразное окно инспектора, которое позволяет работать с методами и переменными.

Для выбранного в предыдущей панели класса в левой панели (в панели категорий методов и переменных) на странице Instance отображаются категории (в другой терминологии — протоколы) его методов экземпляра, а при выборе категории — в правой панели (панели методов и переменных) отображается список всех её методов. Аналогично, на странице Class в левой панели отображаются категории методов класса, а при выборе категории — список всех её методов в правой панели.

Чтобы найти нужный метод, можно в поле Find ввести его имя, начинающееся с символа # (при этом можно пользоваться символом *, создавая шаблон для поиска), либо щелкнуть на иконке поиска метода в панели инструментов системного браузера.

На странице SharedVariable в первой панели отображаются категории разделяемых переменных (переменных класса), а при выборе категории в правой панели появляется список всех её переменных класса.

Если класс определяет переменные экземпляра, на странице InstanceVariable в левой панели отображаются все переменные экземпляра, а при выборе переменной, в правой панели отображаются все методы экземпляра, использующие эту переменную в теле метода. Если класс не определяет переменных экземпляра, то левая панель страницы InstanceVariable остается пустой, а в правой панели отображаются все методы экземпляра. Обычно, выбор любой из страниц остается возможным, даже в том случае, когда на ней нет отображаемых элементов.

Если в панели классов и пространств имён выбирается пространство имён, то доступна только страница SharedVariable. А её левой панели отображаются категории разделяемых переменных пространства имён, а в правой панели при выборе категории — список разделяемых переменных. Чтобы найти нужную переменную, в поле Find нужно ввести её имя (можно использовать символ *, создавая шаблон для поиска).

Панель редактирование исходного текста

Если в любой из верхних панелей производится выбор, на странице Source нижней панели системного браузера (текстовой панели) отображается смолтоковский код, с которым можно работать. В этой панели через её меню <Operate>, меню Edit браузера доступны обычные операции редактирования, типа «вырезать», «копировать» и «вставить», «найти» и «заменить».

Первые три операции доступны и через иконки панели инструментов системного браузера.

Когда выбран пакет, но не выбран класс, в текстовой панели отображается шаблон определения класса. Точно так же, когда выбран протокол, но не выбран метод, отображается шаблон определения метода. Чтобы создать новый класс или метод, надо просто заменить шаблон соответствующим определением. Когда определение отредактировано, его следует сохранить (принять), выбирая из меню <Operate> текстовой панели браузера команду меню Accept.

Как было сказано, образ смолтоковской системы связан с файлом исходного кода. Если исходные файлы правильно не идентифицированы в окне Settings Tool, или неправильно установлен основной каталог системы VisualWorks, или исходный текст просто не доступен, то вместо кода в браузере отображается комментарий, объясняющий, что код невозможно декомпилировать. Если такое произошло, следует правильно установить основной каталог и/или отредактировать страницу Source Files окна Settings Tool, гарантируя, что имя файла с расширением `*.sou` согласуется с именем файла образа.

Чтобы изменить местоположение либо метода экземпляра или класса, либо категории методов, можно воспользоваться командами подменю Move из всплывающего меню операций соответствующей панели.

По умолчанию, панель методов в браузере отображает методы, принадлежащие выбранному классу и протоколу. Есть несколько опций, которые позволяют контролировать и расширять возможности видимости методов. Когда класс выбран, а протокол нет, браузер может отображать все методы класса. Чтобы такое стало возможно, следует выбрать флажок «show all methods when no protocols selected» на странице Browser окна Settings Tool.

Часто полезно видеть еще и методы, наследуемые из других классов. Для этого надо выбрать в системном браузере команду меню Method → Visibility... и в открывшемся вторичном меню выбрать имя нужного суперкласса. Эта установка будет действовать до выбора другого класса.

Чтобы установить возможность видимости наследуемых методов и тогда, когда выбирается другой класс, следует выбрать в том же вторичном

меню одну из команд Show All Inherited или Show All Inherited Except for Object. Чтобы вернуться к отображению только собственных методов, следует выбрать команду Show No Inherited.

Системный браузер позволяет открыть на методах несколько активных панелей, чтобы, редактируя один метод, можно было перейти на другую панель, просмотреть в ней другой метод, и вернуться вновь к редактируемому методу, не открывая нового браузера.

Чтобы создать новую панель, нужно выбрать команду меню View → New View или щелкнуть на иконке новой панели просмотра в панели инструментов браузера. Можно перемещаться по панелям, выбирая нужную из меню View браузера или пользуясь клавишами Alt + n, где n — номер панели. Команда View → Remove current view позволяет удалить активную панель.

6.2. Управление пакетами

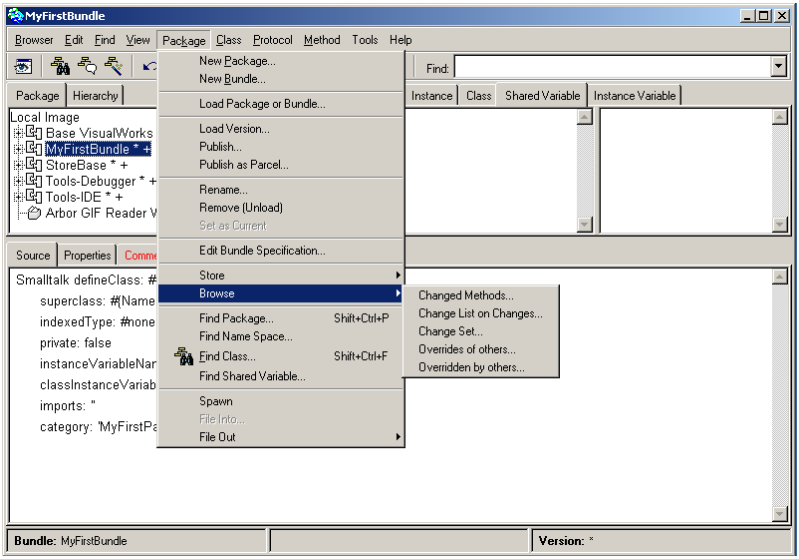


Рис. 6.1: Окно браузера для работы с пакетами и связками.

Страница Package панели пакетов, парселов и иерархии классов отображает среду в виде иерархии пакетов и связок пакетов, загруженных в образ среды. Связку пакетов можно раскрыть, отображая содержащие в ней пакеты и связки пакетов.

Если выбрать пакет, в панели классов и пространств имён отобразится список классов и пространств имён, определенных в этом пакете. Отметим, что сам класс может быть определен не в данном пакете, а в другом, но в данном пакете есть определение метода из данного класса. Имена пространств имён и классов, определенных в пакете, отображаются в панели жирным шрифтом. Имена классов, которые в пакете определяют всего несколько методов, отображаются обычным шрифтом.

Когда выбирается связка пакетов, в панели классов и пространств имён перечисляются все объекты, определенные во всех пакетах, содержавшихся в связке. По мере того, как раскрывается дерево связок и пакетов, выбор вложенной связки или пакета сужает перечень определенных объектов.

Рядом с именем пакета или связки пакетов может отображаться один из следующих индикаторов состояния:

- `*` — пакет был изменен (только при загрузке пакета Store);
- `+` — пакет расширяет классы из других пакетов, возможно переопределяя некоторые определения;
- `–` — пакет имеет определения, которые переопределяются другими пакетами.

Создание пакета

Чтобы создавать новый пакет, следует выбирать команду меню Package → New Package... в системном браузере или команду меню New Package... всплывающего меню страницы, и определить имя нового пакета. Новый пакет будет добавлен в список на странице Packages, добавлен в образ среды, и далее будет сохраняться в образе.

Добавление определений в пакет

Вообще говоря, все новые определения должны назначаться в пакете. Однако, временно, определения можно помещать в специальный пакет (none). Когда создаётся новый класс, пространство имён или разделяемая переменная, пакет или выбирается предварительно, или устанавливается в диалоговом окне создания объекта, или вносится в определение при редактировании шаблона определения объекта.

Если первоначально код размещался в специальном пакете (none), позднее его можно переместить в новый пакет, выбирая перемещаемый код, а затем выбирая команду меню Move из всплывающего меню операций панели классов и пространств имён. Есть несколько возможных выборов из подменю меню Move. Для классов и пространств имён, можно произвести следующий выбор:

Definition to Package... — в появившемся окне указать целевой пакет, и,

нажимая кнопку OK, переместить в него только определение выбранного объекта.

Selection to Package... — в появившемся окне указать целевой пакет, и, нажимая кнопку OK, переместить в него выбранный объект вместе со всеми структурными составляющими, определёнными в данном пакете.

All to Package... — в появившемся окне указать целевой пакет, и, нажимая кнопку OK, полностью переместить в него выбранный объект, независимо от того, определён от только в данном пакете или ещё и в других.

Чтобы отдельно переместить категории (протоколы) переменных, методов или разделяемых переменных вместе с определенными в них объектами, нужно их выбрать, выбрать команду Move → to Package... из всплывающего меню операций панели категорий, в появившемся окне указать целевой пакет, и, нажимая кнопку OK, переместить в него выбранные категории.

Удаление пакета

Чтобы удалить пакет, другими словами, выгрузить его из среды, нужно выбрать пакет и выбрать команду меню Package → Remove (Unload) или команду Remove (Unload) в всплывающем меню страницы пакетов.

6.3. Управление связками пакетов

Связки используются и для того, чтобы собрать вместе и организовать родственные пакеты и связки пакетов, разработанные, возможно, разными членами группы, и для того, чтобы в последующем сделать более удобной загрузку приложений, организованных в виде пакетов.

Создание и размещение связки пакетов

Чтобы создать связку пакетов, следует:

1) В списке пакетов системного браузера выбрать корень связок верхнего уровня Local Image. Для новой связки, выбрать родительскую связку.
2) Выбрать команду меню Package → New Bundle..., чтобы открыть редактор Bundle Specification Editor.
3) В редакторе, в верхнем поле ввода ввести имя новой связки.
4) Выбрать пакеты и/или связки и включить их в новую связку, щелкая на кнопке Add.
5) Ввести порядок загрузки пакетов, выбирая пакет и перемещая его вверх или вниз кнопками Move Up и Move Down. Редактор Bundle

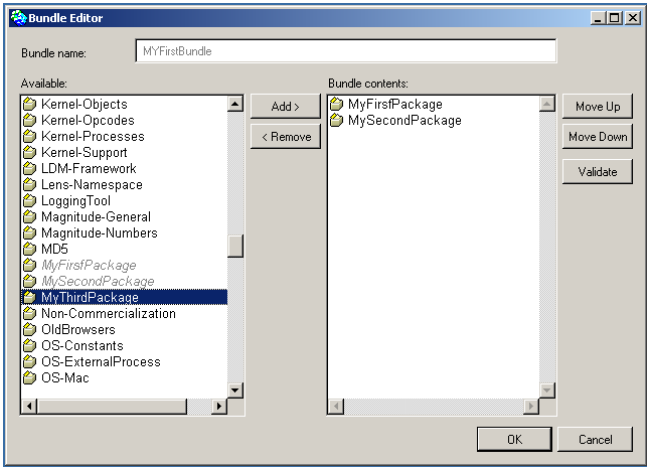


Рис. 6.2: Редактор определения связки пакетов.

Specification Editor всегда перечисляет пакеты и связки пакетов в порядке их загрузки. Если определение пакета A зависит от определения пакета B, то пакет B должен указываться выше по списку.

6) Щелкнуть на кнопке Validate, проверяя, что указанный порядок загрузится. При проверке создаётся список загрузки пакетов данной связки и проверяется, что в получающемся порядке загрузки, каждое пространство имён и каждый класс, требуемый каждым из пакетов либо загружается этим пакетом или пакетом, загружаемым ранее, либо не загружается вовсе ни одним из пакетов. Если это так, порядок пакетов считается правильным. В этом процессе нет попытки отследить определения, которые не загружаются ни одним из пакетов. Сделать корректировку, если она необходима.

7) Когда связка создана, щелкнуть на кнопке OK, создавая связку в образе.

Чтобы отредактировать ранее созданную и сохранённую связку, следует её выбрать и воспользоваться командой меню Package → Edit Bundle Specifications. . . , открывая редактор связок.

Удаление связки пакетов

Чтобы удалить связку из образа, следует выгрузить ее из системы. Для этого надо выбрать связку на странице Package и выбрать команду меню Package → Remove(Unload).

6.4. Управление парселями

Помимо инструмента Parcel Manager, работу с парселями можно вести на странице Parsel системного браузера, которая открывается при выборе команды меню Browser → Parcel. При этом меню браузера Package заменяется на меню браузера Parcel, предоставляя команды по обслуживания парселов.

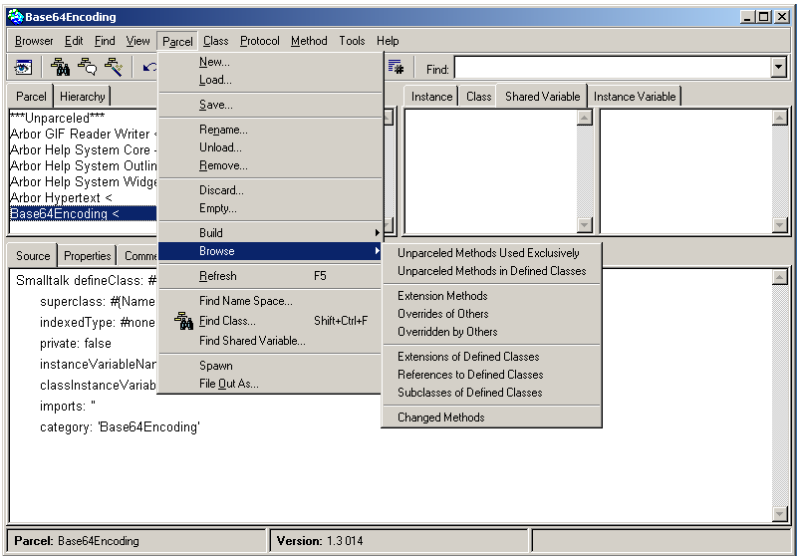


Рис. 6.3: Окно браузера для работы с парселями.

При выборе на странице парсела, в панели классов и пространств имён отображается содержимое выбранного парсела. Имя компонента отображается полужирным шрифтом, если он содержит нечто, определенное в выбранном парселе. Красный текст имени указывает на элементы, определенные более чем в одном парселе, как предупреждение о потенциальных конфликтах.

В списке парселов, после их имени могут стоят символы, указывающие на состояние парсела:

- `*` — “грязный” парсел; он был некоторым образом изменен, но изменения не сохранены в этом парселе;
- `<` — загруженный парсел;
- `!` — парсел имеет незагруженный код;

+ — парсел при загрузке переопределил ранее загруженные определения;
– — парсел содержит определения, которые были переопределены другим загруженным кодом.

Создание нового парсела

Обычно, парсел не приходится создавать явно, поскольку он создаётся при сохранении (публикации) пакетов и связок пакетов, командой меню Publish as Parcel.... То, как формируется парсел, и то, как организуется код, загружаемый из парсела, определяется тем, как был создан парсел.

- Если в парсел был записан отдельный пакет, он загружается в тот же самый пакет.
- Если в парсел была записана связка пакетов и при этом был выбран флажок Include bundle structure, то при загрузке воспроизводится структура связки пакетов.
- Если в парсел была записана связка пакетов, но флажок Include bundle structure не был выбран, код загружается в отдельный пакет, а прежняя структура связки пакетов теряется.

Но могут возникать ситуации, когда нужно явно создать парсел. Чтобы создать новый парсел, следует воспользоваться командой меню Parsel → New... браузера или командой New... всплывающего меню операций страницы парселов. В диалоговом окне ввести имя нового парсела и щелкнуть на кнопке OK. Именем парсела может быть любая строка, которая не содержит начального и конечного пробелов, а внутри строки нет нескольких пробелов подряд. Имя парсела должно быть уникально в пределах образа.

Добавление и удаление определений

Когда создаются новые классы или методы, они добавляются в выбранный парсел, если он есть. Определения, не попавшие ни в один парсел, попадают в системный парсел ***Unparcelled***.

Чтобы добавить существующее определение в парсел, надо его выбрать и переместить (Move) его в парсел, выбирая подходящую команду меню (аналогично перемещению в пакет). Есть несколько команд перемещения, меняющихся вместе с типом перемещаемого определения.

Например, при перемещении класса, доступны такие команды меню:

Move → Defintion to Parcel... — перемещается определение класса, но не его методы;

Move → Selection to Parcel... — перемещается определение класса и его методы, определенные в текущем пакете;

Move → All to Parcel... — перемещается класс, все его методы, определения разделяемых переменных, независимо от того, определенны они в выбранном пакете или в некотором другом.

Этими же командами нужно перемещать пространства имён. Но пространства имён должны перемещаться отдельно. Командой Move → to Parcel... следует перемещать методы и категории методов (вместе с методами) класса, категории разделяемых переменных (вместе с переменными) класса и пространства имён.

Сохранение парсела

Сохранение парсела со страницы Parsel системного браузера аналогично сохранению пакета или связки пакетов со страницы Package.

Чтобы, в последующем, работа с созданными парселами проходила без осложнений, следует соблюдать следующие рекомендации:

- располагать парселы в виде дерева и не допускать перекрестных ссылок;
- тщательно упорядочивать предусловия и постусловия загрузки и выгрузки парселов;
- каждый класс располагать только в одном парселе;
- тщательно упорядочивать классы в парселах;
- после операций загрузки и выгрузки проверять пространство имён Undeclared.

6.5. Определение пространства имён

Левая верхняя панель системного браузера на странице Packages отображает связки и пакеты системы. Вторая слева панель перечисляет имена классов и пространств имен в выбранном пакете. Когда во второй панели выбирается класс или пространство имен, его определение появляется в нижней текстовой панели. Если в этой панели нет выбранного элемента, в текстовой панели отображается шаблон определения класса.

Чтобы создать новое пространство имён, в первой верхней панели следует выбрать пакет, в котором должно быть создано новое пространство имён. Если нужен новый пакет, его надо определить, пользуясь в браузере системы командой меню Package→ New Package и вводя в возникающем диалоговом окне имя нового пакета. Чтобы определить пространство имён вне пакетов системы, следует выбрать в раскрывающемся списке пакетов строку (none) (для начала так и сделаем). Затем выбрать команду меню Class → New → Name Space..., открывая диалоговое окно New Name Space для определения нового пространства имён. В этом окне следует заполнить следующие поля ввода информации:

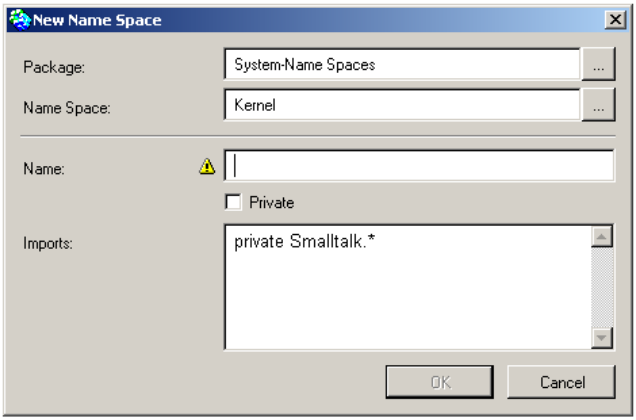


Рис. 6.4: Диалоговое окно определения пространства имён.

Package: — имя пакета, в который будет включено определение нового пространства имен (отобразится имя выбранного пакета). Имя пакета, содержащего определяемое пространство имён, можно изменить, выбирая его из списка, открывающегося при нажатии кнопки с многоточием, справа от поля ввода.

Name Space: — имя родительского пространства имен для нового пространства имён. Как и имя пакета, имя родительского пространства имён можно изменить. В большинстве случаев таким пространством следует выбирать пространство имён Smalltalk, так как создаваемое пространство имён вряд ли должно быть видимо в стандартной библиотеке VisualWorks.

Name: — имя нового пространства имён. Нет никаких ограничений на имя пространства имён, но в родительском окружении имя должно быть уникальным. Чтобы потом ещё и не забыть, зачем создавалось пространство имен, можно использовать в его имени название компании или нечто подобное. Пока имя верхнего уровня уникально, последующие имена, выбранные для пространств имён, классов и разделяемых переменных, защищены от конфликтов.

Private — переключатель, выбор которого делает новое пространство имён частным, то есть недоступным для последующего импорта.

Imports: Список импортируемых пространств имён (операцией конкретного или общего импорта), разделяемых пробелами, и, если необходимо, включающих префикс private. В списке уже есть строка private Smalltalk.*, обеспечивающая доступ создаваемого пространства имён

к множеству стандартных библиотек VisualWorks.

После заполнения всех полей, надо щелкнуть на кнопке OK, определяя новое пространство имён.

Остаётся пояснить значение выбора переключателя Private. Смолток долго не имел механизмов для разделения общих и частных классов и методов. Переменная являлась любо частной (доступной единственному объекту), либо общей (доступной многим объектам). Пространства имён и разделяемые переменные позволяют частично устранить этот недостаток, причем, на двух уровнях: на уровне определения и на уровне импорта. При создании или при импортировании, разделяемая переменная может объявляться как общая (public) или как частная ((private,). Частная разделяемая переменная не доступна для импорта пространством имён или классом!

На уровне определения, каждое определяемое пространство имён (или другая разделяемая переменная) объявляется как общая или частная, через логическое значение поля private:. Когда значение поля — false (флажок не выбран), переменная определяется как общая, и может в последующем импортироваться. Когда значение поля true (флажок выбран), переменная определяется как частная и не может в последующем импортироваться. Частная разделяемая переменная доступна только в том окружении в котором она определена.

Пространства имён обычно определяются как общие, так как они должны импортироваться другими пространствами имён, которые должны обращаться к ним.

Как альтернативный вариант определения пространства имён, можно выбрать пакет и любое пространство имён в нём, чтобы отобразить в текстовой панели его определение. Отредактировать определение, задавая, по крайней мере, новое имя пространства имён, а затем выполнить команду Accept из всплывающего меню текстовой панели, чтобы сохранить новое определение.

При создании собственного приложения следует иметь хотя бы одно пространство имён верхнего уровня. Ответ на вопрос о том, нужны ли дополнительные подпространства имён, зависит от требований доступа к именам в создаваемом программном продукте, от множества классов приложения, от того, как тесно создаваемые классы взаимодействуют между собой. Следует помнить, что все пространства имён и классы, созданные в пределах единственного пространства имён, имеют доступ ко всем разделяемым переменным, определенным в нём. Следует учитывать и следующие моменты:

- если каждый создаваемый класс должен видеть все другие создаваемые классы, то разумно определять их в одном пространстве имён;

- когда создаются классы, которые не нуждаются в доступе к некоторым другим созданным классам, следует рассмотреть возможность их определения в разных пространств имён;
- если в одном пространстве имён создаются классы, которые должны обращаться к объектам из другого пространства имён, можно импортировать в н их такое пространство имён.

Почти наверняка, в ходе разработки приложения, придется перемещать классы и пространства имён из одного пространства в другое. Это весьма просто сделать. Например, чтобы переместить пространство имён, нужно выбрать его в списке пространств имён, а затем выбрать команду Move → to Name space ... из меню Class или из всплывающего меню панели. В открывшемся диалоговом окне выбрать целевое пространство имён и щелкнуть на кнопке OK.

6.6. Определение класса

В отличие от классического определения, когда класс определяется только в иерархии классов, в VisualWorks 7.4.1 класс определяется в пространстве имён, как значение разделяемой переменной в этом пространстве. Причем, эта переменная определяется как константа, так что имя класса изменить нелегко.

Определение класса через диалоговое окно

Чтобы определить класс через диалоговое окно New Class следует в браузере системы выбрать команду меню Class → New Class или щелкнуть на иконке определения класса в панели инструментов браузера.

Свойства класса при его определении определяются двумя страницами: Basic и Advanced. На странице Basic следует определить следующую информацию:

Package: Имя пакета, в котором создается класс. Определяется точно так же, как и в случае определения пространства имён (для начала выберем в раскрывающемся списке пакетов строку (none), то есть определим класс вне пакетов системы).

Name Space: Имя пространства имён, в котором создаётся класс. Пространство имён определяет окружение видимости имени класса. Выберем пространство имён MySmalltalkNameSpace.

Name: Имя класса, которое в указанном пространстве имён должно быть новым, уникальным и начинаться с прописной буквы. Введем имя MyFirstClass.

Superclass: Имя суперкласса определяемого класса, записанное в точечной нотации с именем его пространства имён (см. главу 3). Пространства

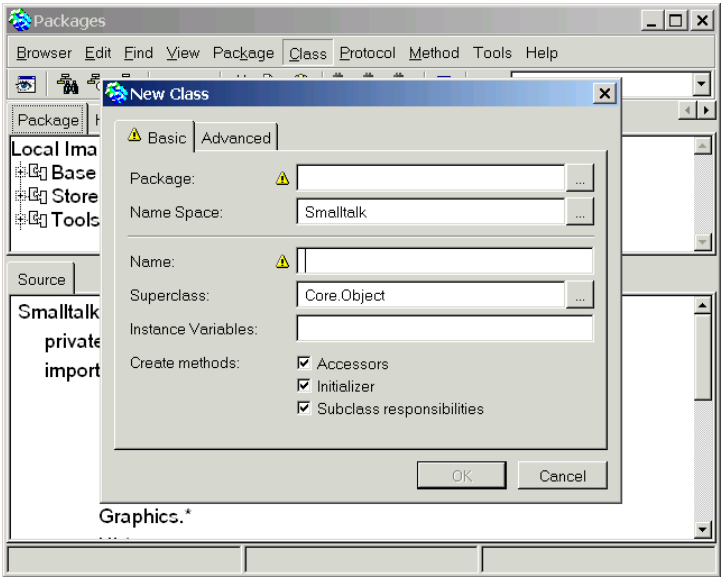


Рис. 6.5: Диалоговое окно определения класса. Страница Basic.

имён класса и его суперкласса могут не совпадать. Определим суперкласс, заданный по умолчанию — Core.Object

Instance Variables: Разделяемый пробелом список имён переменных экземпляра. Введем переменные first second third.

Набор флажков Create Methods: позволяет указать, надо ли при определении класса автоматически создавать методы доступа (Accessors), метод инициализации каждой переменной экземпляра (Initializer), а также создавать шаблоны для тех методов, которые в суперклассах реализованы через #subclassResponsibility. Выберем все эти флажки.

На странице Advanced определяется следующая информация:

Private — если флажок выбран, создается частный класс, который невидим при импорте любому другому классу или пространству имён. Класс обычно определяется как общий (флажок остаётся невыбранным), так как класс должен импортироваться вместе со своим пространством имён.

Indexed Type: — поле определяет тип класса и, в частности, тип значения, которое может храниться в его индексированных переменных (оставим значение по умолчанию #none).

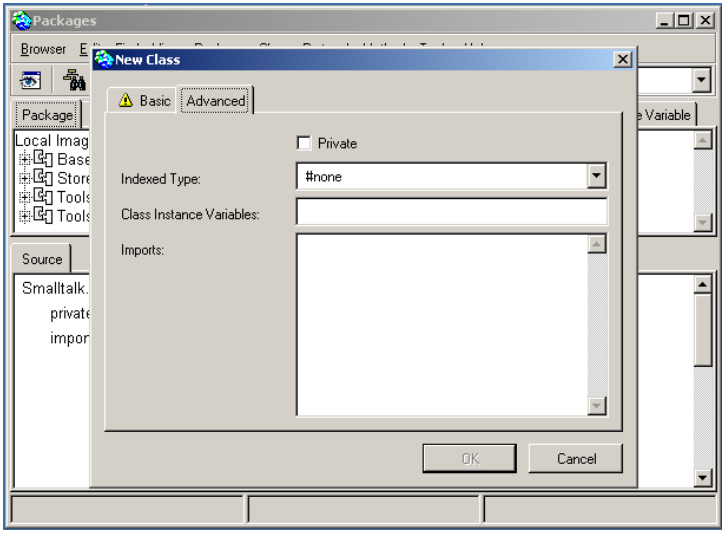


Рис. 6.6: Диалоговое окно определения класса.Страница Advanced.

Class Instance Variables: — разделяемый пробелом список имён экземплярных переменных класса (оставим строку пустой).

Imports: — список связываний, импортируемых классом (ничего импортировать не будем).

Когда обе страницы заполнены, следует нажать кнопку OK, чтобы определить сам класс и все указанные методы.

Отметим, что в VisualWorks переменные класса не указываются при определении класса, а создаются как разделяемые переменные в пространстве имён класса (см. раздел 6.7).

Определение класса через шаблон

Можно определить новый класс и через шаблон определения класса, который является сообщением, посылаемым пространству имён. Чтобы отобразить шаблон в текстовой панели браузера системы, нужно выделить пакет, в котором создается класс, и ничего не выбирать в панели классов и пространств имён. Шаблон имеет следующий вид

```
Smalltalk defineClass: #NameOfClass
    superclass: #NameOfSuperclass
    indexedType: #none
    private: false
    instanceVariableNames: ' instVarName1 instVarName2 '
```

```
classInstanceVariableNames: ''
imports: ''
package: ' NameOfCategory '
```

В шаблоне нужно сделать следующие изменения (сверху вниз):

- Сделать получателем сообщения то пространство имён, в котором будет создаваться новый класс. По умолчанию это пространство имён Smalltalk.
- Определить имя класса как символ, вводя его после ключевого слова defineClass:. Имя должно начинаться с прописной буквы.
- В поле superclass: определить суперкласс для определяемого класса, используя точечную нотацию с его пространством имён.
- В поле indexedType: определить, будет ли класс определять индексированные переменные.
- Определить будет ли класс частным в своем пространстве имён. В поле private: изменить значение на true, чтобы сделать класс недоступным для импортирования другим классам и пространствам имён. Иначе оставить false.
- После ключевого слова instanceVariableNames: ввести строку имён переменных экземпляра, разделённых пробелами.
- После ключевого слова classInstanceVariableNames: ввести строку имён экземплярных переменных класса, разделённых пробелами.
- После ключевого слова imports: ввести строку, состоящую из импортируемых классом связываний, разделяемых пробелами, или оставить строку пустой.

Полностью изменив шаблон, можно определить класс, выполняя либо команду Accept в всплывающем меню текстовой панели браузера, либо команду меню Edit → Accept самого браузера.

Определение класса с помощью шаблона не дает возможности сгенерировать методы. Но системный браузер обеспечивает возможность генерации методов доступа через команду меню Class → [Class] Instance Variables → Create Accessors. . . .

Чтобы изменить определение конкретного класса, достаточно сделать нужные изменения в его определении, а затем сохранить их командой Accept. Но таким образом нельзя изменить имя класса, переместить его в другое пространство имён или пакет. Вместо этого следует воспользоваться командами меню системного браузера Class → Rename или Class → Move, соответственно.

Типы классов

Класс может определять как именованные, так и индексированные переменные экземпляра. Тип класса определяется тем, какие значения могут хранить его индексированные переменные. Например, класс Set определяет свои экземпляры с именованной и индексированными переменными, которые содержат в качестве значения произвольные объекты:

```
Smalltalk.Core defineClass: #Set
    superclass: #Core.Collection
    indexedType: #objects
    private: false
    instanceVariableNames: 'tally'
    classInstanceVariableNames: ''
    imports: ''
    category: 'Collections-Unordered'
```

Переменная tally хранит число элементов в множестве, а индексированные переменные хранят элементы множества, которые могут быть любыми объектами.

Как видно из примера, тип класса определяется символом, стоящим в определении класса в качестве аргумента ключевого слова indexedType:. Предопределены следующие типы классов (символы):

#none — класс только с именованными переменными (возможно, наследуемыми). Его подклассами могут быть классы любого типа.

#objects Класс с индексированными переменными, которые могут хранить любые объекты, и с любым числом именованных переменных. Его подклассами могут быть классы типа #objects или #weak.

#bytes Класс без именованных переменных, но с индексированными переменными, которые могут хранить только байты (например, класс ByteEncodedString и все его подклассы). Методы доступа к индексированным переменным — примитивные методы at: и at:put:, с аргументом ключевого слова put: в виде однобайтного или двухбайтного символа. Класс типа #bytes не может наследовать именованные или индексированные переменные, поскольку его экземпляр содержит только бинарные данные. Поэтому его цепочка суперклассов может состоять только из классов типа #none без именованных переменных или #bytes. Подклассом класса типа #bytes может быть только класс типа #bytes.

#immediate Экземпляры класса этого типа являются указателями на объект (например, SmallInteger, Character). Такой класс не может наследовать именованные или индексированные переменные экземпляра,

поэтому его цепочка суперклассов может состоять только из классов типа #none без именованных переменных. Класс типа #immediate не может иметь подклассов.

#ephemeron Класс только с именованными переменными (возможно, наследуемыми). Первая переменная экземпляра такого класса трактуется сборщиком мусора специальным образом. Поэтому его суперклассами могут быть только классы типа #none и #ephemeron, а подклассами — только классы типа #ephemeron.

#weak Класс с индексированными переменными, которые могут хранить любые объекты, с именованными переменными (возможно, наследуемыми), и со слабо индексированными переменными, содержащими объекты (например, класс WeakArray). Только слабая ссылка на индексированную переменную не защищает последнюю от сборки как мусор. Его суперклассами могут быть только классы типа #none или #objects , а подклассами — классы типа #weak.

Обращение к классу

Наличие пространств имён позволяет иметь множество классов с одним и тем же именем, поэтому теперь для идентификации класса одного имени класса, возвращаемого ранее сообщением name, недостаточно. Нельзя теперь и просто спросить о классе, используя выражение Smalltalk at: aSymbol, как это было в более ранних версиях. Вместо этого используются другие сообщения.

Класс Object — Протокол экземпляра

---

fullName — возвращает строку, содержащую полное имя приемника в точечной нотации.

printString — возвращает строку, представляющую имя класса.

fullyQualifiedReference — когда сообщение посылается классу или пространству имён, возвращает полностью квалифицированное имя.

asQualifiedReference — когда посылается строке или символу, возвращает связанную ссылку.

Вместо выражения Smalltalk at: stringOrSymbol следует использовать выражение stringOrSymbol asQualifiedReference value.

Выражение	Возвращаемый объект	Класс объекта
Object fullName	'Core.Object'	ByteString
Core fullName	'Core'	ByteString
Object printString	'Object'	ByteString
Core printString	'Core'	ByteString
'Object' asQualifiedReference	#{Object}	BindingReference
#Object asQualifiedReference	#{Object}	BindingReference
#Object asQualifiedReference value	Object	Object class
#Core asQualifiedReference	#{Core}	BindingReference
#Core asQualifiedReference value	Core	NameSpace
Object fullyQualifiedReference	#{Core.Object}	BindingReference
Core fullyQualifiedReference	#{Core}	BindingReference

6.7. Определение переменной класса

Чтобы определить переменную класса нужно:

1) В браузере системы, выбрать класс, который будет служить как пространство имён для переменной, и выбрать страницу Shared Variable.

2) Выбрать, если существуют, или добавить и выбрать, категорию для новой разделяемой переменной, в панели категорий методов и переменных (команда New из всплывающего меню этой панели). После добавления первой категории переменных класса, вид текстовой панели кода меняется: в правой части этой панели появляется окно инспектора для значения переменной класса. В уменьшившейся текстовой панели кода отобразится шаблон для определения разделяемой переменной:

```
MySmalltalkNameSpace.MyFirstClass defineShared: #NameOfBinding
    private: false
    constant: false
    category: 'category description'
    initializer: 'Array new: 5'
```

3) В шаблоне следует:

- Заменить #NameOfBinding системным именем, определяющим имя разделяемой переменной (например, #MySharedObject).
- Установить значение поля private: равным true, чтобы сделать переменную частной, или оставить значение поля равным false.
- Установить значение поля constant: равным true:, если значение переменной не должно изменяться; иначе, оставить значение поля равным false.
- Ввести выражение для инициализации значения разделяемой переменной в виде строки в поле initializer:, или ввести nil.

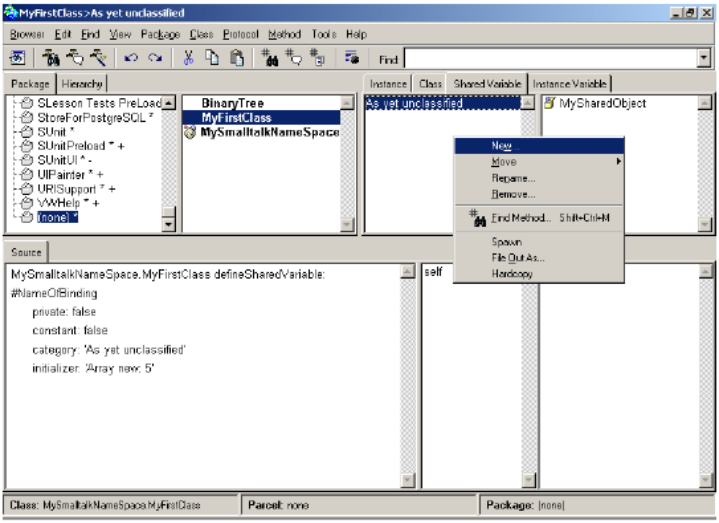


Рис. 6.7: Определение категории переменных класса.

4) Выбрать команду меню Edit → Accept системного браузера или команду Accept из всплывающего меню текстовой панели, чтобы сохранить определение и создать переменную класса. Новая разделяемая переменная будет добавлена в список переменных класса, а её имя появится в панели методов и переменных. После этого данная переменная станет доступной в браузере на странице Shared variables.

6.8. Определение переменных в пространстве имён

Как было сказано ранее, в уже существующем пространстве имён можно определить разделяемую переменную, значением которой будет новое пространство имён (см. раздел 6.5) или класс (см. раздел 6.6).

Но можно определить в пространстве имён разделяемую переменную, значением которой будет некоторый объект, отличный от класса и пространства имён. Именно такие переменные в браузерах обозначаются указанной специальной иконкой.

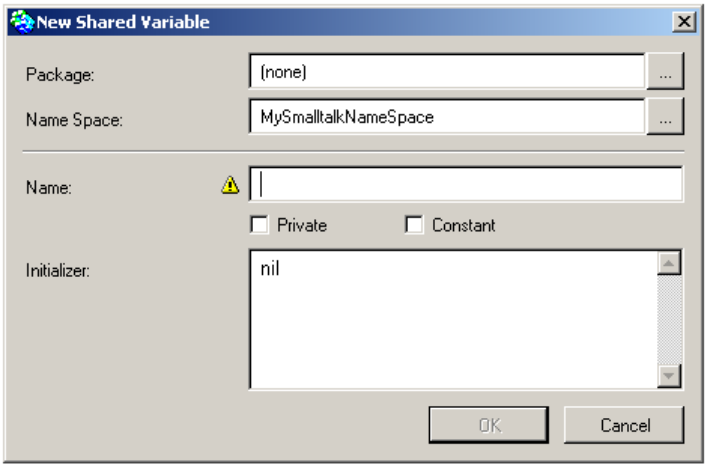


Рис. 6.8: Окно определения переменной в пространстве имён.

Чтобы определить такую переменную, следует выбрать пространство имён, затем страницу Shared Variable... и далее действовать точно также, как и при определении переменной класса (см. раздел 6.7). Того же результата можно добиться, если сначала выбрать команду меню Class → New → Shared Variable... системного браузера или команду New → Shared Variable... всплывающего меню панели классов и пространств имён, а затем в возникающем диалоговом окне ввести всю информацию, необходимую для определения разделяемой переменной (см. раздел 6.7).

6.9. Работа с экземплярами

Смолтоковские объекты или, по-другому экземпляры, обычно создаются при посылке классу сообщений new или new:, возможно в совокупности с дополнительными сообщениями. Сообщения new и new: определены в классе Behavior и наследуются всеми классами.

Уничтожать экземпляры явно не требуется, поскольку VisualWorks, как и любая смолтоковская среда, использует автоматическую сборку мусора. Когда на объект больше не указывают другие объекты (на объект нет ссылок, слабые ссылки не в счёт), система отмечает его как не нужный, и автоматически уничтожает, освобождая память и ресурсы.

В некоторых случаях, например тогда, когда объект использует внешние ресурсы, простой сборки мусора оказывается не достаточно. В этих случаях, используется механизм VisualWorks, называемый завершением (finalization) (см. [9, Chapter 18, Weak Reference and Finalization]).

И в Смолтоке может возникать «утечка памяти», вызванная теми экземплярами, которые не освободили ресурсы и не были собранны как мусор. Чтобы найти их, следует отыскать необычное использование памяти. Для этого можно загрузить в систему пакет AT System Analysis и открыть окно инструмента Class Reporter, выбирая в основном окне системы команду Tools → Advanced → Class reports. На странице Space выбрать в левой нижней панели подозрительный класс, выбрать радио-кнопку Instance size и нажать кнопку Run. Если выполнить эту операцию с используемым образом и с чистым образом, можно выделить классы с возможным мусором. Затем нужно послать классу сообщение allInstances и на результате открыть окно инспектора. Используя команду Inspect Reference Path из меню Object инспектора (она станет доступной, если загрузить парсел All Advanced Tools), отследить снизу вверх содержащие объект корневые структуры. Потенциально опасны корни

- Object classPool at: #DependentsFields
- Object classPool at: #EventHandlers
- ObjectMemory dependents
- sysOopRegistry

Неизменяемые объекты

Некоторые объекты являются неизменяемыми, то есть их внутреннее состояние нельзя изменить. В VisualWorks 7.4.1 все литеральные объекты, экземпляры классов SmallInteger, Character, Symbol и общие экземпляры класса Number являются неизменяемыми. Кроме того, появилась возможность сделать конкретный объект неизменяемым. И, наоборот, исключая экземпляры классов SmallInteger, Character, Symbol, неизменяемые объекты можно сделать изменяемыми.

Попытка внести изменения в неизменяемый объект (например, сообщением become:) или изменить класс такого объекта поднимет исключение NoModificationError. Ответная реакция на такие изменения может быть весьма разнообразной.

Следует иметь ввиду, что адаптированный код старых версий VisualWorks, действуя в новых условиях, не может учитывать свойство неизменяемости. Во многих случаях, таких как создание копии литеральных строк или массивов, сообщение copy будет возвращать изменяемую копию объекта. Например, ошибочно сообщение

```
'' writeStream nextPutAll: 'abc'
```

поскольку пустая строка создана литерально ('') и не изменяема, но следующие два сообщения будут успешно выполнены:

```
'' copy writeStream nextPutAll: 'abc'
String new writeStream nextPutAll: 'abc'
```

Для тестирования и управления неизменяемыми объектами можно использовать сообщения из протокола testing класса Object:

Класс Object	Протокол экземпляра

asImmutableLiteral — возвращает приемник как неизменяемый литерал, если он может быть так представлен.

beImmutable — делает приемник неизменяемым.

beMutable — делает приемник изменяемым, если он не является экземпляром классов Character, SmallInteger, Symbol.

isImmutable — возвращает true, если приемник неизменяем; иначе возвращает false.

isImmutable: aBoolean — делает приемник неизменяемым, если аргумент aBoolean — true, или изменяемым, если в качестве аргумента стоит false) (данное сообщенеие не может посылаться неизменяемым объектам).

isImmutableLiteral Возвращает true, если приемник неизменяемый литерал; иначе возвращает false.

Однако, нововведение имеет и недостатки. Чтобы их заметить, достаточно посмотреть на результаты выполнение следующих выражений:

```
'' isImmutable          → true
'' copy isImmutable     → false
String new isImmutable  → false
'' class                → ByteString
'' copy class           → ByteString
String new class        → ByteString
'' copy = ''            → true
'' copy = String new    → true
'' copy == ''           → false
'' = ''                 → true
'' = String new         → true
'' == ''                → false
```

Два экземпляра одного и того же класса, неотличимы друг от друга, но имеют разное поведение! С некоторой натяжкой с нововведением можно было бы согласиться, если бы последнее выражение возвращало true, как

для неизменяемых экземпляров классов SmallInteger, Character, Symbol. Похожие проблемы возникают с массивами и литеральными массивами, например, с изменяемым массивом Array new: 1 и неотличимым от него неизменяемым литеральным массивом #(nil).

6.10. Определение метода

Методы определяют поведение классов и их экземпляров. Именно здесь происходит реальное программирование на языке Смолток. Методы создаются при помощи браузера системы через изменение шаблона определения метода. Есть два вида методов: (1) методы класса и (2) методы экземпляра. Методы класса определяют поведение класса, а методы экземпляра — поведение экземпляров класса. Методы класса наиболее часто используются для создания экземпляра класса и для инициализации и доступа к переменным класса.

Чтобы увеличить возможность многократного использования методов, методы должны быть предельно короткими и простыми. Имена методов должны быть функционально содержательны, они могут содержать буквы английского алфавита, числа и символ подчеркивания, но не могут начинаться с цифры. Первая буква имени должна быть строчной.

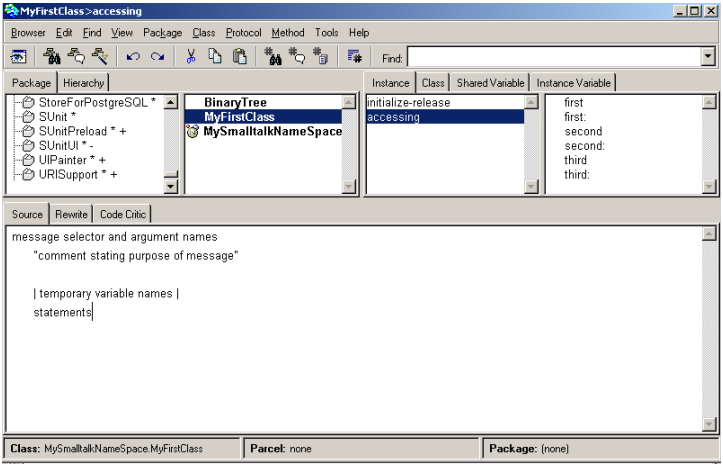


Рис. 6.9: Определение метода в браузере системы.

Чтобы создать метод, следует

1) В браузере системы выбрать класс, в котором создаётся метод.
2) Над панелью категорий методов и переменных выбрать страницу Instance или Class.

3) Выбрать существующую категорию (протокол) или добавить новую, выбирая команду New из всплывающего меню этой панели или команду меню Protocol → New системного браузера, и определяя имя новой категории. В текстовой панели отобразиться шаблон определения метода.
4) Изменить шаблон определения метода, определяя имя метода в первой строке определения. В двойных кавычках "..." следует написать комментарий, кратко описывающий, что метод делает и что возвращает. Если необходимо, указать в вертикальных скобках |...| имена временных переменных метода. Наконец, ввести последовательность смолтоковских выражений (тело метода), определяя выполняемые методом операции.
5) В меню операций текстовой панели выбрать команду Accept, чтобы откомпилировать и сохранить метод. Если текст метода не содержит синтаксических ошибок, он будет откомпилирован.

Если в процессе компиляции были найдены синтаксические ошибки, система сообщит о них, помещая краткое сообщение или открывая диалоговое окно для коррекции ошибки. Ошибку следует исправить и вновь попытаться сохранить метод. Перечислим некоторые наиболее часто встречающиеся синтаксические ошибки.

Undeclared temporary variables (Необъявленные временные переменные). Система выдаст запрос с меню типов переменных, благодаря которому можно быстро и легко объявить каждую из временных переменных.

Undeclared class and instance variables (Необъявленные переменные класса и экземпляра). Когда появиться запрос на объявление переменной класса или экземпляра, следует выбрать в меню команду Abort и объявить переменную прежде, чем продолжить работу над методом. Чтобы сохранить набранный, но ещё не откомпилированный метод и получить, используя браузер системы, возможность переопределить класс, следует выбрать в меню браузера команду Class → Spawn или выбрать команду Spawn из всплывающего меню панели классов браузера. Это приведет к открытию нового браузера на классе, где можно объявить переменные. Вместо команды Spawn можно воспользоваться уже рассмотренной ранее командой View → New View.

Missing period (Отсутствие точки). Когда пропущена точка, система воспринимает две инструкции, как единое выражение сообщения. Как результат, появляется описание ошибки "Nothing more expected" — "Нечто странное".

Missing delimiters (Отсутствие разделителей). Когда пропускается скобка, появляется описание ошибки "Right parenthesis expected", указыва-

ющие на потерю правой скобки, или "Period or right bracket expected", указывающие на потерю точки или правой скобки.

Каждый метод возвращает один объект, который может быть и набором. Чтобы возвратить объект, отличный от приемника, такому объекту или выражению, его вычисляющему, должен предшествовать символ оператора возврата объекта (^). Клиенты, которые заинтересованы только в побочных результатах метода, могут игнорировать возвращенный методом объект. Возвращенный объект может сохраняться в переменной, если позже этот объект должен рассматриваться снова.

Напомним еще раз, когда символ оператора возврата объекта находится в блоке, это приводит к прекращению дальнейшего выполнения метода и возвращению методом результата, полученного в блоке.

Когда метод выполняет некоторый тест (условное выражение) и возвращает одно значение, если результат выполнения теста истина, и другое значение, если ложь, то символ оператора возврата объекта может использоваться только однажды — перед условным выражением. Такой подход рассматривается как хороший стиль программирования, поскольку имеет то преимущество, что объединяет две точки выхода в одну.

6.11. Контрольные вопросы

1) Какие панели имеет системный браузер?
2) Что и как отображает панель пакетов, парселов и иерархии классов?
3) С какими символами и в каких случаях отображеются имена парселов в панели парселов?
4) Какие команды доступны в браузере для работы с парселами?
5) Какие предосторожности следует соблюдать, чтобы работа с созданными парселами проходила без осложнений?
6) Что и как отображает панель классов и пространств имён?
7) Как найти в среде нужную разделяемую переменную, в том числе класс или пространство имён?
8) Что и как отображает панели методов и переменных? Какие страницы имеют эти панели?
9) Как установить возможность видимости наследуемых методов и отказаться от такой возможности?
10) Как найти нужный метод?
11) Как открыть несколько активных панелей браузера?
12) Как создать новый пакет и удалить существующий?
13) Как создать и разместить новую связку пакетов, как удалить связку пакетов?

14) Как определить новое пространство имён?
15) Как определить в существующем пространстве имён новую разделяемую переменную?
16) Как определить новый класс через диалоговое окно?
17) Как определить новый класс через шаблон определения?
18) Какие типы классов существуют в VisualWorks и как, используя наследование, они могут располагаться в иерархии классов?
19) Из каких сообщений состоит протокол обращения к классу в связи с существованием в VisualWorks пространств имён?
20) Как определить новую переменную класса?
21) Какие объекты являются неизменяемыми? Каков протокол тестирования и управления неизменяемыми объектами?
22) Как определить новый протокол методов класса или экземпляра?
23) Как определить новый метод класса или экземпляра в уже существующем протоколе?
24) Какие синтаксические ошибки чаще всего созникают в процессе компиляции метода?

Глава 7

Отладка кода

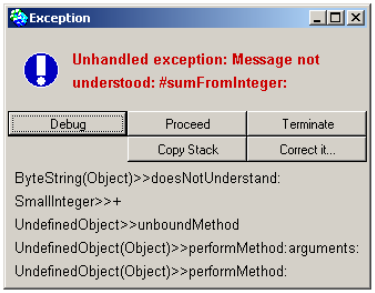
Отладка — трудная задача, состоящая в поиске причин сбоя программы. Синтаксические ошибки здесь не виноваты — они отлавливаются при компиляции. Причина таких ошибок семантическая, и их поиск может потребовать много времени и сил. Когда в программе обнаруживается некоторая ошибка, открывается окно уведомлений Walkback, показывая несколько последних посланных сообщений. Окно для отладки Debugger, открывающееся из окна уведомлений, позволяет исследовать хронологию посылки сообщений и изменений значений переменных, позволяет, изменяя код, управлять выполнением программы. Общие принципы отладки в смолтоковской среде описываются во многих книгах, например, [4, глава 23], [8, глава 21], [15, стр. 73–80, 191–197]. Поэтому здесь мы их кратко повторим, отмечая особенности версии VisualWorks.

VisualWorks 7.4.1 содержит дополнительные инструменты, позволяющие проследить за потоком выполнения выражений и назначением переменных, и тем самым существенно упростить процесс отладки. Все эти инструменты построены над программными зондами. Зонды позволяют вставлять в код триггеры (совокупность условий, инициирующих выполнение некоторых действий, запускаемых автоматически при возникновении определенных условий), не изменяя хода выполнения исходного текста, но позволяя установить причины отклонений в выполнении программы. Одни зонды прерывают выполнение кода (breakpoint — контрольная точка), а другие только выводят информацию о состоянии кода (watchpoint — точка отслеживания).

7.1. Окно уведомлений

Когда в программе происходит ошибка, появляется окно уведомлений об исключительной ситуации, которое отображает последние пять сообщений, посланных в контекстном стеке. Панель стека вызовов перечисляет те

сообщения, которые были посланы, но так и остались ждать возвращаемого объекта, когда произошло прерывание.



Часто этой информации вполне достаточно, чтобы понять причину ошибки и исправить её. Если дело обстоит именно так, нужно нажать на кнопку Terminate (Завершить), закрывая окно уведомлений и прерывая выполнение кода.

Когда ошибка не серьезна и позволяет продолить выполнения кода (окно отображает только предупреждение), можно нажать кнопку Proceed (Продолжить), закрывая окно уведомлений и продолжая выполнение программы.

7.2. Окно отладки кода

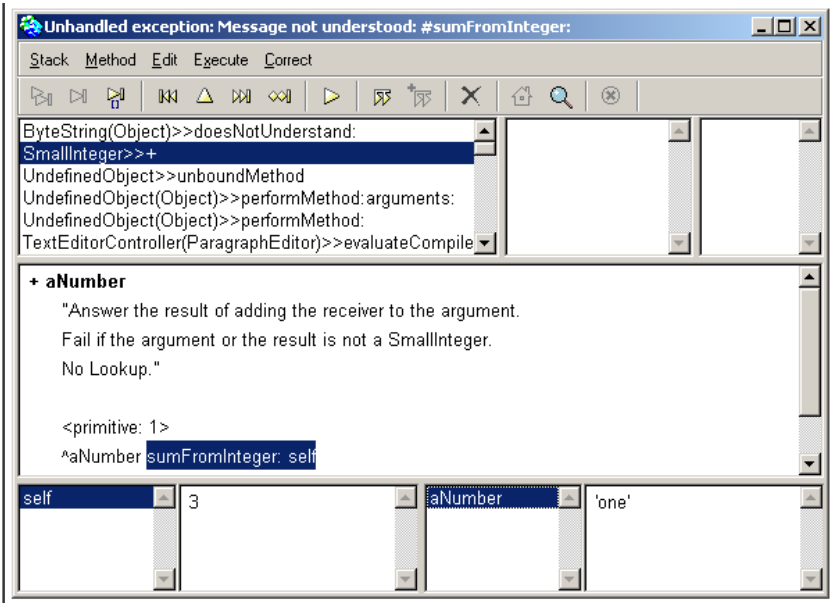


Рис. 7.1: Окно отладчика VisualWorks.

Когда необходимо исследовать ситуацию более подробно, следует нажать кнопку Debug (Отладить), закрывая окно уведомлений и открывая окно отладчика (см. рис. 7.1). Окно отладчика отображает существенно боль-

ше информации о происшедшем, и дает возможность в пошаговом режиме проследить за ходом выполнения кода до возникновения ошибки, исследовать каждый метод на каждой стадии его выполнения, проследить за значениями всех переменных в каждом контексте, динамически изменить значения переменных или код методов, перекомпилировать любой метод, вставить контрольные точки и точки отслеживания, повторно выполнить код с выбранного места с новыми значениями переменных и логикой.

Наверху окна отладчика располагаются три стековых панели. Слева — панель стека вызовов (стека выполнения), перечисляющая посланные сообщения, выполнение которых не было завершено, поскольку из-за ошибки они не получили ожидаемого ими объекта. Эта панель подобна окну уведомлений об ошибке. Две панели справа — инспектор стека выполнения, который позволяет просматривать значения промежуточного стека выбранного выражения (см. раздел на стр. 109).

Панель под ними — текстовая панель, аналогичная текстовой панели кода системного браузера. Когда выбрано посланное сообщение в панели стека, соответствующий метод отображается в панели кода. В теле этого метода автоматически выделяется фрагмент кода, выполнение которого не было завершено.

Вдоль нижнего края окна расположены окна двух инспекторов, слева — переменных экземпляра, и справа — временных переменных, которые позволяют просматривать и изменять значения соответствующих переменных (см. раздел **??**). Переменные и их значения изменяются всякий раз, когда выбирается другая строка в панели стека.

Панель инструментов по умолчанию расположена в верхней части окно, но её можно установить между тремя верхними панелями к панелью кода, выбирая флажок Show toolbar below context list на странице Debugger инструмента настройки параметров среды Settings.

Чтение стека вызовов

Окно уведомления возникает как реакция на исключительную ситуацию. Например, при попытке выполнить в рабочем окне выражение 3 + 'one'. Сообщение в верхней части окна (метка окна уведомления) информирует о том, что сообщение #sumFromInteger: было послано объекту, но не было им понято. Верхняя строка стека вызовов ByteString(Object)>> doesNotUnderstand сообщает о том, что это был экземпляр класса ByteString, который не понял сообщение, и потому вызвал метод doesNotUnderstand, реализованный суперклассом Object). Такая информация может показаться несколько странной, поскольку мы посылали сообщение + экземпляру класса SmallInteger, что и написано в второй строке стека вызовов. Послед-

ние три строки ничего не объясняют — они просто отражают некоторые из механизмов выполнения кода, которые в этом случае не имеет смысла исследовать.

Отметим синтаксические особенности строк в стеке вызовов. Символ соединения (>>), используемый и в окне уведомления, и в панели стека вызовов отладчика, не является селектором. Он только указывает на связь метода с реализующим его классом. Например, строка вида SmallInteger>> + относится к методу экземпляра + из класса SmallInteger, указывая на класс приемника собщения и на селектор сообщения, посланного приемнику, в виде имяКласса>>сообщение. Иногда, как в первой строке примера, после имени класса, в скобках указывается имя того класса, в котором метод был найден (если он не совпадает с классом приемника).

Вернемся к примеру и постараемся понять, что произошло в при выполнении метода SmallInteger>> +. Для этого откроем окно отладчика, нажимая кнопку cdDebug в окне уведомлений. Метка окна отладчика идентична метке окна уведомлений, из которого оно было создано. Выберем в панели стека строку SmallInteger>> + и в панели кода отобразиться текст метода, внутри тела которого автоматически выделяется сообщение, которое было послано и обрабатывалось в тот момент, когда произошла ошибка (см. рисунок 7.1).

Мы видим, что примитивным методом, который обычно складывает два целых числа, исходное выражение не было обработано. После чего стал выполняться альтернативный смолтоковский код, в котором сообщение sumFromInte было послано экземпляру класса ByteString. В новом сообщении число 3, ранее приемник сообщения +, стало параметром, а строка 'one', ранее параметр сообщения +, стала приемником. Таким образом, сообщение sumFromInteger: было послано строке, а не целому числу, то есть была предпринята попытка выполнить выражение 'one' sumFromInteger: 3. То, что переменные имеют такие значения, видно в нижних окнах-инспекторах, которые позволяют просматривать значения переменных, существовавшие момент выполнения кода.

Инспекторы отладчика позволяют изменить значение переменной и заново запустить процесс вычисления. Например, в примере, сначала изменим значение переменной, заменяя 'one' на допустимое значение, скажем, 1. Из всплывающего меню операций этой панели выберем команду Accept. И, наконец, выберем, по порядку, команды меню отладчика Execute → Restart и Execute → Run (или нажмем на соответствующие иконки в панели инструментов отладчика), вызывая процесс выполнения выражения. Поскольку исходное выражение задавалось в рабочем окне, в нём и появится результат.

На практике, значение параметров сообщений обычно обеспечиваются соответствующими методами, а не литеральными выражениями, как в нашем примере. С другой стороны, ошибка могла бы состоять не в значении переменной, а в неправильной логике одного из методов. В любой ситуации, решая задачу отладки, работая в панели кода отладчика как в панели кода системного браузера, можно отредактировать код метода, воспользоваться командой Accept для сохранения нового определения и продолжить обработку, используя новое определение.

Если в определении метода изменить имя метода и воспользоваться командой Accept, метод будет сохранён в системе, и на нём откроется браузер метода, после чего, в текстовой панели отладчика восстановиться первоначальный текст метода из выбранного контекста. Как результат, вне каталогов методов будет создан новый метод.

Особенности инспектора стека

Инспектор стека занимает верхний правый угол окна отладчика. Он позволяет просмотреть значения промежуточного стека выбранного выражения. Панели инспектора определяются динамически и появляются в двух формах.

Первая форма появляется тогда, когда выбранный контекст — верхний контекст и он собирается посылать сообщение. В этом случае панели определяют аргументы сообщения и получателя сообщения. Если, например, в стеке вызовов посылается сообщение (Array with: 2 with: 4) инспектор стека отобразит следующую информацию:

```
arg2:     4       "второй аргумент"
arg1:     2       "первый аргумент"
r:        Array   "приемник сообщения"
```

Если выбранный контекст не является верхним контекстом и на стеке есть объекты, они будут идентифицированы как "top "$-1\ldots$, "$-n$". Идентификатор "top" относится к вершине стека, а поля от "-1" до "$-n$" ссылаются на элементы стека, отстоящие от вершины на $1, \ldots, n$. Дополнительно, если выбран верхний контекст, и он собирается сохранить объект в переменной или возвратить объект, панели инспектора стека отобразят только "top".

Когда в выполняемом коде выполняется следующий шаг, инспектор автоматически выбирает самый верхний элемент стека, если он есть. Это позволяет сразу же увидеть, что сообщение возвращает. Однако, нужно знать, что этот элемент не всегда является результатом последнего посланного сообщения.

Трассировка потока сообщений

Панель стека выполнения в окне отладчика отображает сверху вниз последние посланные сообщения перед тем, как произошла ошибка. Чтобы увидеть связанный с посланным сообщением метод, следует выбрать это сообщение. В теле метода, посланное, но не выполненное сообщение, автоматически подсвечивается. Пользуясь многочисленными командами, присутствующими в виде команд меню и кнопок в панели инструментов, можно перемещаться по потоку сообщений и исследовать его.

Меню Stack**:**

Copy Stack Report
: Копирует список из панели стека в буфер обмена, затем его можно вставить в документ или рабочее окно.

Show More Stack
: Команда добавляет (если возможно) новые элементы в список. В нормальных условиях отладчик открывается с размером стека в 500 элементов.

Filter Stack
: Допускает фильтрацию стека, в соответствии с определениями, сделанными в окне инструмента Settings на странице Debugger в редакторе, который открывается при нажатии кнопки Edit. . . .

Use Short Class Names
: Когда команда выбрана, отображаются только имена классов, без использования точечной нотации для тех классов, которые не видны вне пространства имён Smalltalk.

Select Home Context
: Просматривает стек и ищет начальный (базовый) контекст текущего выбранного контекста. Если начального контекста нет на стеке, то диалоговое окно сообщит пользователю о возникшей ситуации.

Inspect Context
: Открывает окно инспектора на контексте метода.

Bookmark Context
: Подсвечивает элемент стека (контекст) и добавляет его как элемент (пункт) меню Stack → Bookmark, тем самым обеспечивая быстрый доступ к этому контексту.

Clear Bookmark
: Удаляет закладку для этого контекста.

Меню Method**:**

Большинство пунктов этого меню такие же, как и в системном браузере. Есть только одно исключение.

Recompile with Full Blocks

Повторно компилирует метод так, чтобы все блоки являлись полными блоками. Это приводит к тому, что метод повторно вводится в среду (как и командой Accept). Но этот метод — временный метод и он исчезает после выполнения.

Меню Execute**:**

Step Into

Команда наиболее детализированного продвижения по коду. Когда выбирается посылаемое сообщение, команда посылает это сообщение, отображает получающийся контекст и позволяет его исследовать. Другими словами, происходит перемещение по методу, входя в контекст каждого встречающегося метода и блока.

Step Команда перемещения по методу с заходом в контекст блоков, встречающихся по пути.

Step Over

Команда перемещения по методу с пропуском блоков, встречающихся по пути.

Restart

Инициализирует выбранный контекст и перезапускает его на выполнение с начала метода, как будто отладчик только что ступил в него. Метод может быть или экземпляром класса CompiledMethod или экземпляром класса CompiledBlock.

Return

Позволяет прекратить дальнейшее выполнение выбранного метода или блока и немедленно передать управление его отправителю.

Run to caret

Продвинуться до символа (^).

Jump to caret

Пропустить, не выполнения, код до следующего символа (^), что переместит точку выполнения в начало утверждения, содержащего символ (^).

Run Продолжить выполнение от текущей точки.

Run with Break on Return

Эта и следующая команды полезны при отладке циклов. Данная команда подобна команде Run, за исключением того, что устанавливает неявную контрольную точку, которая вызывается по возвращении из текущего контекста. Кроме того, отладчик остается открытым. Выполнение останавливается или по возвращении из контекста, или если встречается другая контрольная точка. Выполнение гарантирован-

но остановится, и, таким образом, можно прервать выполнение цикла, вышедшего из под контроля.

Run with Break Again
: Команда подобна предыдущей, за исключением того, что она не устанавливает контрольную точку, которая будет вызвана по возвращении из контекста. Эта команда доступна только после того, как контрольная точка была установлена командой Run with Break on Return.

Terminate
: Завершить процесс отладки и закрыть отладчик. Происходит то же, что и при закрытии окна, используя команду close.

Abort
: Команда активизируется, когда выполняется код, во время выполнения одной из команд Step или команды Run with Break on Return. При этом, выполнение кода должно занять достаточное времени, прежде чем можно будет заметить, что команда стала доступной.

Меню Correct**:**

Define method
: Команда активизируется тогда, когда наверху контекстного стека оказывается Message doesNotUnderstand:. Она вставляет новое определение метода для непонятого селектора, в котором просто вызывается сообщение halt.

Correct selector
: Команда активизируется тогда, когда наверху контекстного стека оказывается Message doesNotUnderstand:. Она предлагает список предположений о правильном правописания непонятого селектора сообщения. Если правильный селектор есть в списке, его можно выбрать, и команда исправит исходный текст, повторно откомпилирует метод, и пошлёт сообщение.

7.3. Программные зонды

Программные зонды содержат механизмы аналогичные техническим средствам, используемым при поиске неисправностей в электронных блоках для проверки состояния системы в конкретных точках. Такой зонд не изменяет структуры радиосхемы, но, когда используется, может немного изменить её характеристики. Точно так же используются программные зонды, которые не изменяют структуры исходного смолтоковского текста (введение и удаление зонда не влияет на выполнение кода), но влияют на время выполнения кода.

Зонд может быть вставлен до или после посылки любого сообщения, оператора назначения значения, или после ссылки на переменную. Вставка

зонда фактически производит вставку сообщения, которое посылают зонду.

Существуют два основных механизма: установка контрольных точек и установка точек отслеживания. Каждый зонд определяет условное выражение и выполняемую операцию. Если условное выражение возвращает истину, то операция выполняется. В случае контрольной точки, условное выражение всегда возвращает истину. Выполняемая операция определяется типом зонда.

Контрольная точка

Контрольная точка (breakpoint) является самым простым видом зонда. Когда она вызывается, то сразу открывает окно системного отладчика, пропуская стадию оповещения через окно уведомлений. Верхний метод в стеке — метод, содержащий контрольную точку. Посылаемое текущее сообщение зависит от размещения контрольной точки. Когда нужно сразу вызвать отладчик, контрольная точка — хорошая альтернатива вставке в код выражения self halt, не требующая изменений в исходном тексте.

С контрольной точкой можно использовать условное выражение, позволяя протестировать конкретное условие и по его результату вызвать контрольную точку. Условное выражение может включать любые операции. Однако, после завершения, оно должно возвращать логическую переменную. Окно отладчика откроется, если возвращается логическое значение true, и не откроется, если false.

В дополнение к вставке в программу контрольной точки или сообщения останова self halt, можно вручную остановить смолтоковскую программу, нажимая специальную последовательность клавиш.

Последовательность клавиш Control + y вызывает функцию обработки прерываний. Эту последовательность следует использовать тогда, когда нужно остановить (заморозить) выполнение программы, которая выполняет бесконечный цикл, или зафиксировать её состояние в определенной стадии выполнения.

Последовательность клавиш Control + \ замораживает все пользовательские процессы и открывает окно инструмента Process Monitor, позволяя исследовать каждый из них отдельно.

Точка отслеживания

Точка отслеживания (watchpoint) отображает информационную строку в специальном окне Watchpoint, не прерывая выполнение кода. Есть следующие четыре типа точек отслеживания, из которых надо будет произвести выбор при их определении.

Top of Stack (Вершина стека). Отображает значение переменной, распо-

ложенной в данное время наверху стека аргументов, которое может быть аргументом или результатом посылки последнего сообщения.

Instance Variable (Переменная экземпляра). Отображает значение заданной переменной экземпляра.

Temp Variable (Временная переменная). Отображает значение выделенной временной переменной.

Expression Watch (Надзорное выражение). Отображает строку, которая является результатом выполнения смолтоковского кода. Данный тип зонда дает возможность пользователю самостоятельно сформировать отображаемую строку и, таким образом, отобразить информацию любой сложности.

Строка, отображаемая каждой из первых трёх точек отслеживания, получается при посылке объекту сообщения debugString. Этот метод определен в классе Object как ^ self printString.

7.4. Работа с зондами через браузер

Вставка любого зонда в исходный текст производится при выборе соответствующей команды меню браузера и, в большинстве случаев, сопровождается вводом дополнительной информации. Когда зонд вставляется в метод в браузере, метод изменяется с объекта CompiledMethod на объект ProbedCompiledMethod, а все его блоки изменяются с объекта CompiledBlock на объект ProbedCompiledBlock. Кроме того, когда вставляется второй зонд, создаётся копия первого откомпилированного с зондом метода, и новый зонд вставляется в копию. Это сделано для того, чтобы по неосторожности не был изменён метод активного процесса, вызывая отказ виртуальной машины.

Установка простой контрольной точки

Простая контрольная точка в определении метода устанавливается при размещении курсора в том месте метода, в котором следует прервать выполнение, с последующим выбором из всплывающего меню операций панели команды Insert Breakpoint. . . . Символ в месте расположения курсора подсвечивается, указывая контрольную точку (о характере подсветки зонда см. раздел на стр. 120).

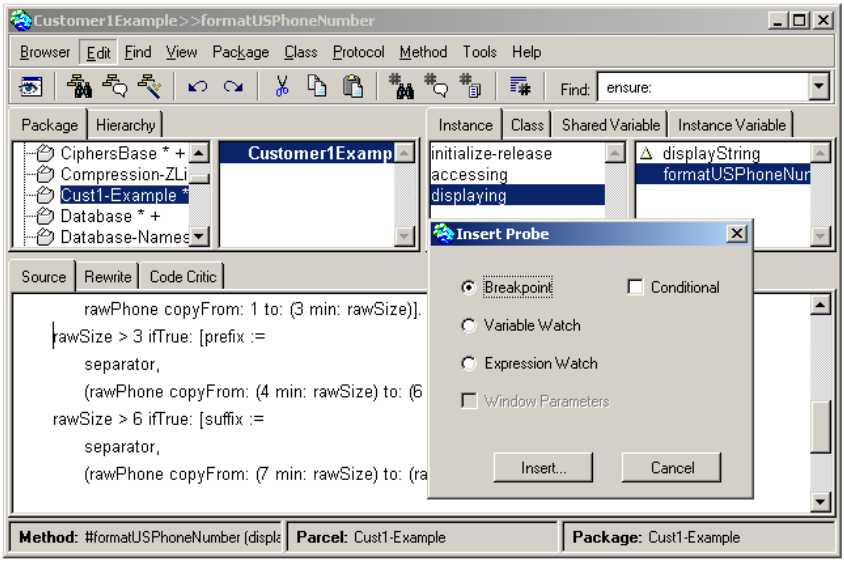


Рис. 7.2: Вставка контрольной точки.

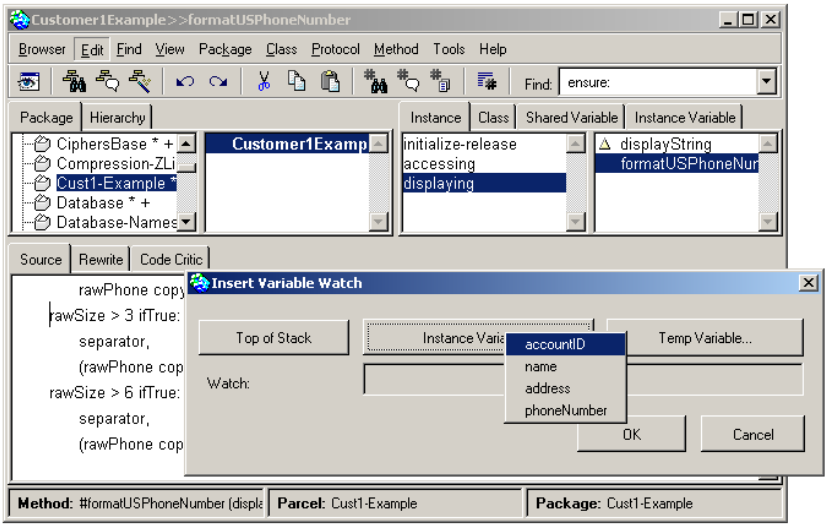


Рис. 7.3: Вставка точки отслеживания за переменной.

Установка точки отслеживания за переменной

Точка отслеживания, не прерывая выполнение кода, отображает сообщение в специальном окне просмотра. Чтобы установить точку отслеживания за переменной, следует разместить курсор в определении метода, и выбрать в всплывающем меню опреаций команду Insert Probe. В редакторе Probe Type выбрать радио кнопку Variable Watch. Затем щелкнуть на кнопке Insert..., открывая диалоговое окно Select Watch Variable (см. рис. 7.3). Три кнопки этого окна позволяют определить одну переменную, за которой нужно наблюдать. При выборе кнопки Top of Stack в окне просмотра будет отображаться значение, расположенное на вершине аргументного стека. При выборе кнопки Instance Variable или Temp Variable появится список доступных переменных (кнопка Temp Variable активна, если метод имеет временные переменные). Когда переменная выбрана, следует щелкнуть на кнопке OK. Зонд установится, и окно просмотра в первый раз откроется, когда будет вызван этот зонд, отображая значение выбранной переменной.

Установка надзорного выражения точки отслеживания

Надзорное выражение обеспечивает контроль за выполнением кода через отображение некоторой информации. Выражение должно сформировать свою содержательную информационную строку.

Чтобы установить точку отслеживания, следует разместить курсор в определении метода, и выбрать в всплывающем меню операций команду Insert Probe. В редакторе Probe Type выбрать радио кнопку Expression Watch. Затем щелкнуть на кнопке Insert..., открывая диалоговое окно Expression Watch Probe.

Верхнее текстовое поле — редактор Conditional Test Expression (редактор выражения условного тестирования) (см. раздел на стр. 117). Текстовое поле Window ID и связанные с ним кнопки позволяют выбирать окно просмотра для отображения результата выполнения надзорного выражения.

Нижнее текстовое поле Watch Expression позволяет определить, что и как отображать в окне просмотра. Здесь можно определить любое смолтоковское выражение, которое возвращает строку. Например, значение переменной rawSize можно отобразить так:

```
^ 'The current value is: ', rawSize value printString, '.'
```

или так

```
^ 'The current value is: ',
        rawSize value printString, '.'
```

(перевод каретки включается в такую строку). Когда соответствующая информация введена, следует щелкнуть на кнопке OK. Зонд установится и, когда он будет вызван в первый раз, откроется окно просмотра.

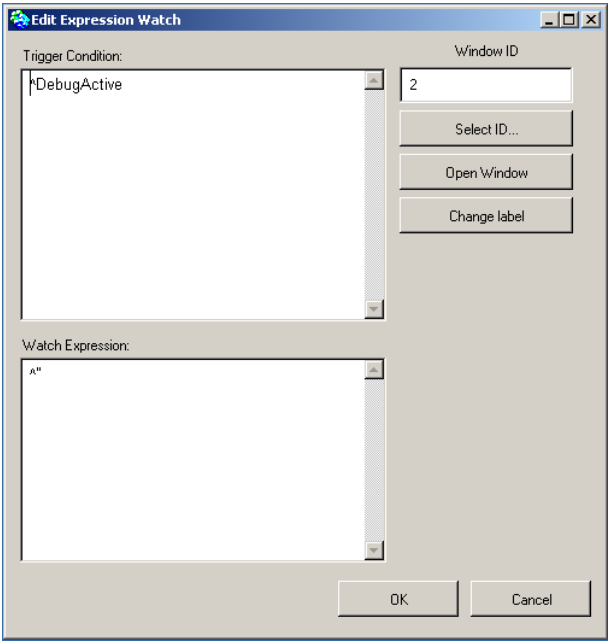


Рис. 7.4: Диалоговое окно Expression Watch Probe.

Удаление зонда

Из метода можно удалять каждый из установленных зондов по одиночке или сразу все. Чтобы удалить единственный зонд, следует выбрать его подсвеченный символ, а затем выбрать команду Remove Selected Probe из всплывающего меню операций. Чтобы удалить из метода все зонды, следует выбрать команду Remove All Probes из всплывающего меню операций.

Создание контрольной точки с условием

Контрольная точка с условием прерывает работу кода в заданной точке, только если выполняется указанное условие. Чтобы разместить условную контрольную точку, следует поместить курсор в нужную точку кода, а затем выбрать из всплывающего меню операций команду Insert Probe, открывая окно редактора Select Probe Type. Выбрать радио кнопку Breakpoint и флажок Conditional. Щелкнуть на кнопке Insert..., открывая окно редактора Conditional Text Expression.

Первоначальное выражение — false, которое не позволит вызвать контрольную точку. Следует заменить его таким выражением, которое возвра-

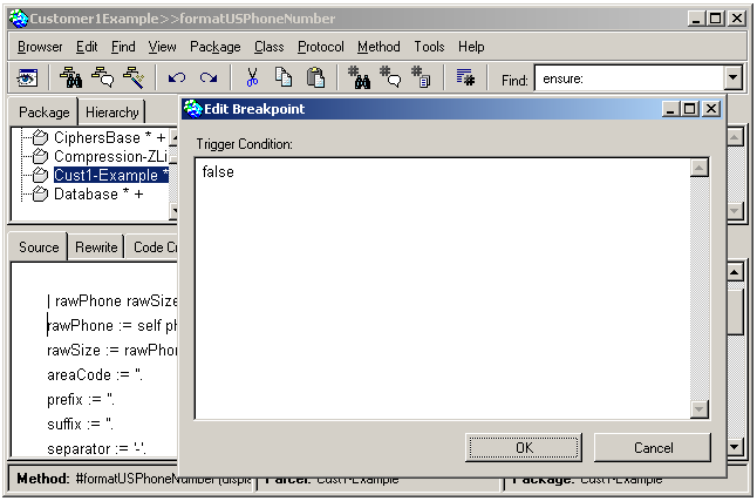


Рис. 7.5: Вставка контрольной точки с условием.

щает истину только в тех случаях, когда должна вызываться контрольная точка, позволяя исследовать состояние программы. Например, если в приложении надо вставить контрольную точку, отслеживающую значение переменной, скажем rawSize, то можно было бы установить условное выражение rawSize value < 0 которое прервет выполнение кода приложения только тогда, когда значение переменной rawSize окажется меньше 0.

Когда выражение введено, его следует сохранить (командой Accept из всплывающего меню операций), и щелкнуть на кнопке OK.

Выражение зонда — нормальное смолтоковского выражение, за исключением того, что оно имеет дополнительное окружение для переменных. Это окружение разрешает выражению ссылаться на переменные в контексте исследуемого метода и на переменные экземпляра его получателя. Дополнительно, каждый зонд может обращаться к собственным локальным переменным отладки и к глобальным переменным отладки. Команды меню, которые доступны в панели редактора условия, позволяют определять новые переменные.

Есть две предопределенные переменные, на которые можно ссылаться, чтобы получить информацию о том контексте, в котором произошла активация зонда. Это переменные DOITCONTEXT и TopOFStack. Переменная DOITCONTEXT содержит сам контекст, а TopOFStack — объект, находящийся на вершине контекстного стека. Дополнительная предопределенная переменная ThisProbe содержит информацию о состоянии зонда (напри-

мер, такой как characterIndex). Эта информация может оказаться полезной, при конструировании надзорного выражения.

Ещё одна глобальная переменная отладки DebugActive позволяет сделать все зонды доступными или недоступными, используя соответствующие команды меню Debug → EnableProbes, Debug → DisableProbes в стартовом окне среды.

Текстовая панель редактора условного выражения содержит следующие команды, помогающие в формировании выражений:

Insert var (Вставить переменную) — открывает ряд меню и подменю, содержащих все допустимые локальные переменные, обеспечивая удобный способ расположения имени переменной и вставки его в текст. Выбранная переменная вставляется в текст.

Define debug var (Определить переменную отладки) — позволяет пользователю определить переменные отладки (локальные и глобальные).

Inspect debug vars (Инспектировать переменные отладки) — открывает окно инспектора на словаре локальных или глобальных переменных отладки. Фактический инспектируемый словарь определяется выбираемым пунктом подменю, local или global.

Reset method (Сбросить метод) — сбрасывает выражение к стандартному выражению и методу.

Insert expression (Вставить выражение) — отображает меню выражений в текущей библиотеке выражений. Выбранное выражение вставляется в точке вставки текста.

Save expression (Сохранить выражение) — спрашивает пользователя об имени, идентифицирующем выражение, а затем сохраняет текст выражения в библиотеке выражений. Существует одна библиотека для выражений тестирования и другая библиотеки для выражений надзора.

Выбор окна просмотра

Для зондов можно определить окно, какое будет отображать выражение. Это позволяет многократно использовать существующие окна, и отображать множество надзорных строк в одном окне.

Когда зонд определяется впервые, в диалоговом окне Select Probe Type нужно выбрать флажок Window parameters. Для надзорного зонда за переменной диалоговое окно Window ID открывается после выбора переменной. Для надзорного выражения диалоговое окно Window ID встроено в диалоговое окно редактора выражения (см. рисунок 7.4).

Чтобы выбрать окно, нужно либо ввести числовой идентификатор окна в панели ввода, либо щелкнуть на кнопке Select ID и выбрать окно из списка. Кнопка Open Window немедленно открывает указанное окно, не ожидая

того момента, когда зонд будет вызван. Кнопка Change Label позволяет ввести более описательную строку для метки окна, по которой позже можно идентифицировать это окно.

Когда параметры окна установлены, следует щелкнуть на кнопке OK.

Изменение зонда

Надзорные зонды и зонды условных контрольных точек можно изменять. Переменную для зондов, отслеживающих переменные, изменить нельзя, но можно изменить условие, надзорное выражение и параметры окна.

Чтобы изменить зонд, следует выбрать представляющий его подсвеченный символ, после чего выбрать из всплывающего меню операций команду Modify probe, открывая окно редактора зонда. Хотя окно редактора меняется в зависимости от вида зонда, команды редактора остаются теми же, что и для установки зонда. В редакторе следует сделать нужные изменения и щелкнуть на кнопке OK.

Расположение зонда

Когда зонд присутствует в методе, его позиция в исходном тексте обозначается подсвеченным символом в той позиции, в которой находится зонд. Постоянные зонды обозначаются подчеркиванием символа и его красным цветом. Временные зонды, которые доступны только в среде отладчика, обозначаются подчеркиванием символа и его желтым цветом [1]. Значение подсветки в разных ситуациях означает следующее:

В селекторе сообщения:

- Первый символ или только символ — активация зонда происходит перед посылкой сообщения.
- Последний символ или последующий пробел — активация зонда происходит после посылки сообщения.
- Последний символ первого ключевого слова — активация зонда происходит после посылки сообщения.

В имени переменной:

- Первый символ — активация зонда происходит перед доступом к переменной (обычно, перед назначением значения).
- Последний символ — активация зонда происходит после доступа к переменной (обычно, после чтения).

[1] Так как подсвечивание — один из способов выделения текста, операции, изменяющие текст, могут удалять подсветку зонда, не удаляя сам зонд.

Перекомпиляция метода с зондами

Всякий раз, когда метод перекомпилируется, вследствие сохранения метода в среде или переопределения класса, зонды удаляются из метода. Однако, браузер предоставляет пользователю возможность, вновь установить зонд, или отказаться от его установки. Если пользователь вновь устанавливает зонд, зонд проверяется для того, чтобы определить, являются ли он всё ещё совместимым с повторно откомпилированным методом. Если выражение зонда более не совместимо с методом, зонд всё же устанавливается, но блокируется, а пользователь имеет возможность скорректировать ситуацию. Если переменная, ради которой определялся надзорный зонд удаляется, зонд не устанавливается.

Ограничения

Подсветка зонда. Выполнение операции форматирования текста в браузере приводит к потере подсветки зонда. Если после изменения формата, текст сохраняется в среде, зонд теряется. Поскольку зонды определяются их позицией в исходном тексте, переформатирование приводит к потере этой позиции и зонд уже нельзя восстановить.

Вставка зонда в выражения возврата объекта. Компилятор VisualWorks код вида

```
^ condition ifTrue: [expression1]
            ifFalse: [expression2]
```

компилирует так, как-будто было написано

```
condition ifTrue: [^ expression1]
          ifFalse: [^ expression2]
```

Зонды добавляются согласно дереву синтаксического разбора. Если была сделана попытка зондировать возвращаемое значение, и зонд вставлялся в символ (^) первого примера, результат будет такой, как если бы зонд был вставлен в символ (^) только для одного из выражений возврата объекта второго примера. Поэтому следует вставлять два зонда, один в конец expression1, другой — в конец expression2.

Та же самая ситуация возникает для следующего блока кода:

```
[statements. . .
condition ifTrue: [expression1]
          ifFalse: [expression2]] value
```

Если зонд помещается в условие, полагая отобразить значение, возвращаемое одним из выражений expression1 или expression2, реально он будет перехватывать только одно из выражений. Снова, следует вставить зонд и в конец expression1, и в конец expression2.

7.5. Зонды на уровне класса

Три команды Add Class Probe, Remove Class Probe, Browse Probed Methods из меню Class системного браузера или всплывающего меню его панели классов и пространств имён поддерживают управление зондами на уровне класса в целом.

Добавление зондов на уровне класса

Команда меню Add Class Probe позволяет вставить зонды в несколько методов одной операцией. Такие зонды совместно используют любое условное выражение и любое надзорное выражение, позволяя единственному надзорному выражению или условному выражению контрольной точки использоваться несколькими методами. Однако, однажды вставленные, выражения становятся независимыми, и, если позже выражение изменить, изменение будет распространяться только на один зонд. Эта команда имеет два подменю: On Instance Variable Access... и On Message Receipt....

Меню On Instance Variable Access...

Команда вставляет зонд в каждой точке ссылки на выбранную переменную в каждом методе в пределах группы методов. Если ссылка — операция чтения, зонд вставляется сразу после байт-кода операции, которая размещает объект на стеке. Если ссылка — операция записи, зонд вставляется сразу перед байт-кодом операции, который сохраняет объект в переменной. Когда команда выбирается, открывается окно установки, списковая панель которого отображает отфильтрованный список методов, ссылающихся на выбранную переменную экземпляра. Нужно выбрать переменную из выпадающего списка Selected Instance Variable, и критерии фильтрации, используя флажки On Read, On Write, Include Subclasses. Тип ссылки отобразится непосредственно перед строкой каждого метода. В списке методов нужно выбрать те методы, в которые следует встроить зонд.

Тип действия (Type of Action), которое будет выполняться на выбранных методах, определяется выбором соответствующей радио кнопки и/или флажка, и может быть одним из перечисленных ниже.

Breakpoint (Контрольная точка)
: Вставляет контрольную точку в точку ссылки на переменную в каждом выбранном методе.

Smart Watch (Интеллектуальное надзорное выражение)
: Вставляет зонд надзорного выражения в точку ссылки на переменную в каждом методе. Выражение возвращает строку, содержащую имя класса, селектор метода и расположение символа зонда в методе. Когда зонд вызывается, эта строка записывается в окно просмотра.

Затем сообщение debugString посылается объекту на вершине стека и результирующая строка записывается в окно просмотра после строки идентификации метода.

Simple Watch (Простое надзорное выражение)
: Вставляет зонд надзорного выражения в точку ссылки на переменную в каждом методе. Когда зонд вызывается, объекту, находящемуся на вершине стека, посылается сообщение debugString, и возвращённая строка записывается в окно просмотра.

N Simple Watches (N простых надзорных выражений)
: Аналогично предыдущему. Отличие состоит в том, что каждый зонд имеет собственное окно просмотра.

Expression Watch (Надзорное выражение)
: Вставляет зонд надзорного выражения с определённым пользователем выражением в точку ссылки на переменную в каждый метод. Когда зонд вызывается, выражение выполняется, и возвращённая строка записывается в окно просмотра. После закрытия окна установок, открывается редактор выражения, позволяя определить надзорное выражение (см. раздел на стр. 116).

Conditional (Условное выражение)
: После закрытия окна установок, открывает редактор условного выражения, позволяя его определить (см. раздел на стр. 117).

Window parameters (Параметры окна)
: После закрытия окна установок, открывает панель Window parameter, позволяя задать окно просмотра (см. раздел на стр. 119).

Generate report (Сгенерировать отчет)
: Заставляет сгенерировать отчёт, когда закроется окно установок. В отчёте перечисляются все методы, выбранные для вставки зонда.

Меню On Message Receipt...

Команда позволяет вставить зонд в начало каждого метода в выбранной группе методов. Когда команда выбирается, открывается окно установок. Панель отображает список методов, определенных в классе и, возможно, в его подклассах. Тип действия (Type of Action), которое будет выполняться на выбранных методах, определяется выбором соответствующей радио кнопки и/или флажка, и может быть одним из перечисленных ниже.

Breakpoint, Conditional, Window parameters, Generate report
: Совпадают с аналогичными действиями для команды On Instance Variable Access....

Simple Msg Trace (Трассировка простого сообщения)
: Вставляет зонд надзорного выражения перед первым выражением в

каждом выбранном методе. Когда вызывается, зонд записывает в окно просмотра класс приемника метода и селектор метода.

Ivar Watch
: Вставляет надзорный зонд за переменной экземпляра в каждый выбранный метод. Когда вызывается, зонд записывает в окно просмотра значение переменной. Когда выбрана эта радио кнопка, становиться доступной кнопка выпадающего меню Select Variable, позволяя выбрать переменную экземпляра.

Expression Watch (Надзорное выражение)
: Вставляет зонд надзорного выражения в каждый метод. Когда зонд вызывается, он заставляет выражение выполниться, и результирующая строка записывается в окно просмотра. После закрытия окна установок, открывается редактор выражения, позволяя определить надзорное выражение (см. раздел на стр. 116).

Команда меню Remove class probes

Меню позволяет удалить все зонды из методов выбранного класса. Команда From This Class Only удаляет зонды только в выбранном классе. Команда From This Class and Subclasses удаляет все зонды из класса всех его подклассов.

Команда Browse probed methods

Команда открывает браузер на всех исследуемых методах в выбранном классе.

7.6. Установка временных зондов в отладчике

Всплывающее меню операций панели кода отладчика содержит те же команды, что и панель кода браузера, но добавляет команды поддержки временных зондов. Временные зонды сохраняются только на время отладки, и удаляются, когда метод возвращает объект, поскольку они обращаются только к контексту метода и его блокам. Временный зонд выделяется в тексте желтым цветом, вместо красного, которым выделяются постоянный зонд. Создание отладчиком временного зонда достигается выбором дополнительного флажка Temporary в окне инструмента Probe Selection Panel (смю рис. 7.6).

Отладка итераций

Уже отмечалось, что команды Run with Break on Return и Run with Break Again используются при отладке итераций, в том числе и циклов. Чтобы ими воспользоваться следует войти в отладчик (вставляя перед циклом контрольную точку или выражение self halt), вставить временную контрольную

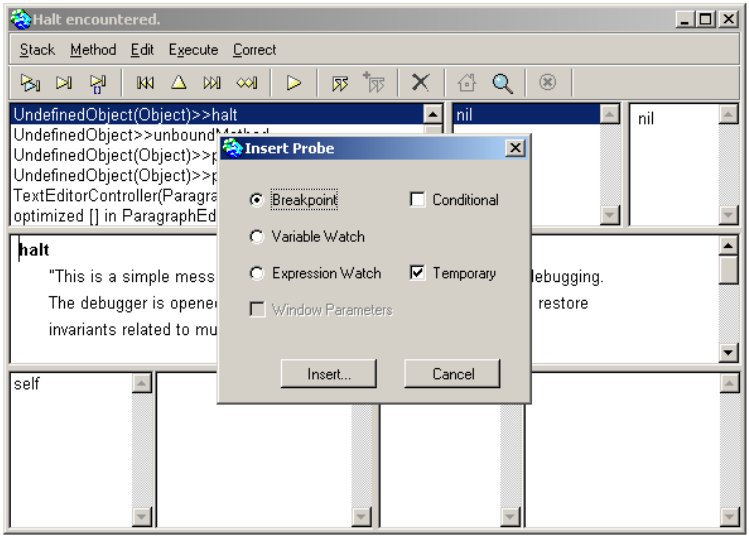


Рис. 7.6: Вставка временной контрольной точки в отладчике.

точку нужного типа либо в код цикла, либо в некоторое сообщение, которое посылается в цикле, и пошагово выполнить код пользуясь командами Run with Break on Return и Run with Break Again.

7.7. Контрольные вопросы

1) Как среда разработки реагирует на ошибки в коде?
2) Какие инструменты используются в среде VisualWorks для отладки кода?
3) Что перечисляется в панели стека вызовов?
4) Что позволяет делать окно отладчика со смолтоковским кодом?
5) Что позволяют делать инспекторы отладчика, расположенные в нижней части окна отладчика?
6) Что позволяет делать инспектор стека, расположенный в справа вверху окна отладчика?
7) Какие команды отладчика позволяют перемещаться по коду?
8) Что такое программный зонд?
9) Что такое зонд контрольной точки?
10) Что такое зонд точки отслеживания?
11) Какие четыре типа точек отслеживания существуют в среде VisualWorks?

12) Что такое надзорное выражение?
13) Как установить простую контрольную точку в системном браузере?
14) Как установить контрольную точку с условием в системном браузере?
15) Какое смолтоковское выражение в коде эквивалентно установке простой контрольной точки?
16) Как установить точку отслеживания за переменной в системном браузере?
17) Как установить надзорное выражение для точки отслеживания?
18) Как удалить вставленные в метод зонды в системном браузере?
19) Какие команды поддерживают управление зондами на уровне класса?
20) Поддержку каких зондов добавляет отладчик?
21) Какими двумя цветами выделяются зонды в отладчике и системном браузере, какой характеристикой зонда определяется цвет выделения зонда?

Глава 8

Исключения и их обработка

Исключение (особая ситуация) — необычное или нежелательное событие, которое может происходить во время выполнения приложения. Хотя не все исключения — ошибки, ошибки являются одними из тех наиболее важных исключений, которые должно обрабатывать приложение. Когда возникает исключение, приложение может его перехватывать и вызывать специальную обработку исключения.

В версии VisualWorks 7.4.1 используется механизм обработки особых ситуаций, соответствующий ANSI-стандарту и реализованный через иерархию класса GenericException (ОбщееИсключение). Этот механизм должны использовать все создаваемые приложения.

8.1. Классы исключений

Исключения представляются экземплярами класса Exception и его многочисленных подклассов, наиболее важными из которых являются Error (Ошибка) и Notification (Уведомление). В свою очередь, подклассы этих классов, определяют более специальные виды исключений. При создании приложений и, в случае необходимости, определения новых классов для специальных исключений, новый класс исключения следует создавать как подкласс Error (в ситуации, когда нельзя будет продолжать выполнение приложения), или Notifier (в ситуации, когда выполнение приложения можно будет продолжить).

Исключение работает, сотрудничая с экземпляром класса Signal, который создаёт исключение, когда ему посылается некоторый вариант сообщения #raise. Исключение хранит информацию о природе ошибки — о сигнале (signal), который создал его, об объекте (originator), который породил его, о параметре исключения (parameter). Приложение может использовать эту информацию, чтобы определить реакцию на исключение, которая может состоять в том, чтобы прервать выполнение операции, продолжить вы-

полнение или перезапустить операцию. Все экземпляры класса Exception и его подклассов отвечают на сообщение description, возвращая строку, которая описывает данное исключение.

Каждый подкласс в Exception или определяет, или наследует сообщение defaultAction, которое вызывается тогда, когда данное исключение происходит, если явно не вызывается другое сообщение. Создавая приложение, можно создавать новые подклассы исключений и, если потребуется, специальные методы их обработки (обработчики) (см. в последующих разделах).

Перечислим общие классы исключений, указывая особое событие, представляемое этим классом, и выполняемое по умолчанию действие.

Таблица 8.1: Классы исключений и их действия по умолчанию

Класс исключения	Событие исключения	Действие по умолчанию
ArithmeticError	Любая ошибка, возникающая при вычислении арифметической операции.	Наследуется из класса Error.
Error	Любая ошибка программы.	Открыть окно уведомлений.
MessageNotUnderstood	Сообщение было послано объекту, который не определяет соответствующего метода.	Наследуется из класса Error.
Notification	Любое необычное событие, которое не вредит выполнению программы.	Ничего не делать, продолжить выполнение.
Warning	Необычное событие, о котором пользователь должен знать.	Задать вопрос, предполагающий ответ Yes/No (Да/Нет) и возвратить логическое значение.
ZeroDivide	Попытка деления на нуль.	Наследуется из класса ArithmeticError.

8.2. Обработка исключений

Обработка по умолчанию большинства исключений приводит к отображению окна уведомления. В многих ситуаций этого достаточно. Когда же для обработки исключения требуется действие, отличное от действия по умолчанию, нужно определить свой *обработчик исключения*.

Обработчик исключения определяется через сообщение on:do:. Первый

аргумент — класс исключения, за которым наблюдает обработчик. Второй аргумент — блок кода (блок обработки), который выполняется тогда, когда происходит данное исключение. Блок обработки — блок с одним аргументом. Экземпляр класса произошедшего исключения передается блоку обработки как аргумент.

Например, следующий код определяет обработчик для исключения, возникающего при попытке деления на нуль, который печатает информационное сообщение в виде строки, отображаемой в окне Transcript:

```
-1.0 to: 1.0 do:
    [ :i |
        [ 10.0/i. Transcript cr; show: i printString]
            on: ZeroDivide
            do: [:ex | Transcript cr; show: 'divideByZero'.]
    ].
```

В окне Transcript будет отображена следующая информация:

```
-1.0
divideByZero
 1.0
```

Обработчик исключения обычно завершает свою работу, возвращая значение блока обработки, вместе значения блока приемника (защищенного блока). Если в дополнение к возвращению значения из блока обработки, надо ещё и выйти из метода, содержащего этот блок, то, как и в случае любого другого блока, следует использовать оператор возврата значения.

```
-1.0 to: 1.0 do:
    [ :i |
        [ 10.0/i. Transcript cr; show: i printString]
            on: ZeroDivide
            do: [:ex | Transcript cr; show: 'divideByZero'.
                    ^ 'divideByZero abort'.]
    ].
```

В окне Transcript будет отображена следующая информация:

```
-1.0
divideByZero
```

а метод возвратит строку 'divideByZero abort' и прекратит работу.

Вместо точного указания характера исключения, можно указывать более общее исключение. Например,

```
[x / y] on: Error do: [:ex | ^ 'divideByZero abort'].
```

В этом примере класс Error определяет то исключение, которое будет обрабатываться вместо исключения ZeroDivide. Тогда, в силу наследова-

ния, обработчик будет обрабатывать и исключение Error, и исключение, которое являются экземпляром его подкласса. В данном случае используется уже существующая в VisualWorks иерархия классов

```
GenericException
    Exception
        Error
            ArithmeticError
                DomainError
                    ZeroDivide
```

В примере выше возвращаемая информация минимальна, но вполне достаточна. Однако, вряд ли разумно обрабатывать каждое происходящее исключение, используя выражение, подобное такому:

```
[... некоторый код ...]
        on: Exception
        do: [:ex | Transcript show: 'An exception occurred'; cr.]
```

Класс Exception является слишком общим, и при таком коде приложение будет так отвечать на все исключения, включая обращения к объектам Notification, которые никак не влияют на выполнение приложения.

Обработчик исключения может автоматически получать информацию о том исключении, с которым он имеет дело:

```
-1.0 to: 1.0 do:
    [ :i |
        [ 10.0/i. Transcript cr; show: i printString]
            on: Error
            do: [:ex | Transcript cr; show: ex description.]          ].
```

В этом случае в окне Transcript будет отображена информация, учитывающая строку, возвращаемую возникшим исключением по сообщению description:

```
-1.0
Can't divide a number by zero
 1.0
```

Иногда нужно установить обработчик исключений так, чтобы он обрабатывал несколько исключений, которые не обязательно входят в одну ветвь иерархии. Это можно сделать, используя класс ExceptionSet. Если происходит исключение из указанных классов или их подклассов, активизируется блок обработки. Можно неявно создавать множество исключений, определяя список исключения непосредственно в обработчике. Например:

```
[... некоторый код ...]
```

```
    on: ZeroDivide, Warning
    do: [код обработки исключения]
```

Посылка сообщения (,) классу исключения с другим классом исключения в качестве аргумента, создает экземпляр класса ExceptionSet.

Если нужно многократно использовать одно и то же множество исключений, можно создать множество исключений явно и связать с ним переменную:

```
specialExceptions := ExceptionSet with: ZeroDivide with: Warning
```

после чего, данное множество исключений можно использоваться как аргумент ключевого слова on: в обработчике исключений.

8.3. Оповещение о возникновении исключения

Исключение создается (поднимается, вызывается) при посылке сообщения raiseSignal классу, определяющему исключение. Например, выражение Error raiseSignal создает исключение, оповещающее об ошибке. Если был определен специальный обработчик исключения Error, именно он и будет выполняться. Иначе, будет выполняться обработчик исключения по умолчанию.

Часто полезно, поднимая исключение, обеспечить текстовое описание ситуации. Это можно сделать, используя сообщение raiseSignal:

```
Warning raiseSignal: 'the disk is almost full'
```

Строка-аргумент ключевого слова raiseSignal: включается в возвращаемое значение сообщения description, когда оно посылается возникшему объекту-исключению.

Иногда полезно поднять исключение с определенным параметром, а не со значением по умолчанию. В этом случае можно послать сообщение signalWith:, с объектом, возвращаемым как аргумент. Например, иногда полезнее поднимать исключение, передающее в качестве параметра сам объект, а не исключение:

```
Exception signalWith: self.
```

Поднять исключение можно посылая любому классу сообщение из категории signal constants, определенное в самом классе или наследуемое из его суперклассов, в том числе и из класса Object, которое просто возвращает класс поднимаемого исключения. Например, следующий код прервёт вывод информации в окно Transcript на одну секунду, если во время его выполнения пользователь нажмёт клавиши Ctrl+Y.

```
[1 to: 500 do: [:i | Transcript show: ' ', i printString]]
    on: Object userInterruptSignal
    do: [:ex | (Delay forSeconds: 3) wait.
            ex resume]
```

При создании приложения и, в случае необходимости, определения нового класса для специального исключения, новый класс исключения следует создавать как подкласс Error (в ситуации, когда нельзя будет продолжать выполнение приложения), или как подкласс Notifier (в ситуации, когда выполнение приложения можно будет продолжать).

8.4. Среда исключений процесса

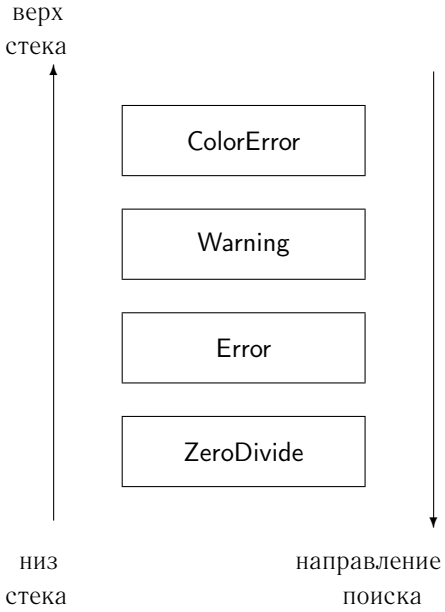
Каждый процесс VisualWorks имеет свою среду исключений, которая представляется упорядоченным списком активных обработчиков. Когда начинается новый процесс, список пуст. Когда выполняется блок, являющийся приемником сообщения on:do:, обработчик исключения добавляется в начало списка, и его элементом становиться выражение on:do:. Если в пределах блока-приемника определен другой обработчик исключения, он снова добавляется в начало списка среды исключений данного процесса.

Если в пределах среды исключений сообщается об исключении, система обработки исключений посылает сообщение первому элементу списка, то есть последнему добавленному в список элементу, чтобы определить, обрабатывает ли он сгенерированное исключение. Первый обработчик исключений, к которому обратились и который может обработать поднятое исключение, делает это. Рассмотрим пример.

```
[ block 1 code
    [ block 2 code
        [ block 3 code
            [ block 4 code]
            on: ColorError
            do: [ handler code for 4]]
        on: Warning
        do: [ handler code for 3]]
    on: Error
    do: [handler code for 2]]
on: ZeroDivide
do: [ handler code for 1]
```

Первому блоку-приемнику посылается сообщение on: ZeroDivide do: [handler code for 1], которое первым попадает в среду исключений. Внутри первого блока второму блоку-приемнику посылается сообщение on: Error do: [handler code for 2], и этот обработчик вносится в начало списка. Затем, внутри второго блока третьему блоку-приемнику посылается сообщение on: Warning do: [handler code for 3], и этот обработчик вносится в начало списка. Наконец, внутри третьего блока четвертому блоку-приемнику посылается сообщение on: ColorError do: [handler code for 4], и этот обра-

ботчик вносится в начало списка. Таким образом в списке находятся четыре обработчика исключений.



Предположим, что в указанной среде исключений выполняется код, который сообщает о возникновении ошибки ZeroDivide. Первый активный обработчик исключений (последний добавленный в список) обрабатывает исключение ColorError, и потому не выполняется. Следующий — обработчик исключения Warning так же не выполняется. Третий элемент обрабатывает экземпляр класса Error, который является суперклассом для ZeroDivide. Он может обрабатывать исключение ZeroDivide, и делает это. Выполняя свой do-блок, обработчик исключения создаёт новую собственную среду исключений, "урезая" существующую среду исключений и включая в новую только те обработчики, которые были созданы перед ним. В этом примере обработчик исключения Error создаёт новую среду исключений, содержащую только обработчик исключения ZeroDivide, поскольку он был единственным обработчиком, созданным перед обработчиком исключения Error.

Если обработчик исключения попытается продолжить выполнение задачи с той точки, где оно было остановлено (послано сообщение resume), то восстанавливается первоначальная среда исключений, иначе первоначальная среда исключений отбрасывается.

Если для возникшего исключения при просмотре среды исключений обработчик не найден, выполняется метод по умолчанию defaultAction, при этом среда исключения остаётся той же самой, какой она была, когда возникло исключение.

8.5. Возобновляемые и невозобновляемые исключения

Блок обработки обычно завершается выполнением последнего выражения, значение которого используется как значение, возвращаемое обработчиком исключения. Однако, куда должно возвращаться управление с этим значением, зависит от того, является ли исключение возобновляемым или

невозобновляемым.

Возобновляемость — свойство исключения, а не его обработчика. Большинство экземпляров подклассов Error не возобновляемы поэтому не возвращают управления в метод, который поднял исключение, а производят возвращение значения непосредственно из блока обработки. С другой стороны, исключения типа Notification и Warning не являются ошибками, они возобновляемы, и обычно возвращают значение активного обработчика исключения из сообщения, вызвавшего исключение.

```
Warning raiseSignal: 'Low memory, save files!'
```

Обработку исключений можно контролировать посредством явной проверки, является ли исключение возобновляемым, используя сообщение isResumable. В следующем примере обработчик исключения возвращает 5 или 10 из сообщения on:do: в зависимости от характера поднятого исключения:

```
[someExceptionClass raiseSignal]
    on: Exception
    do: [:exception |
            exception isResumable
                ifTrue: [5]
                ifFalse: [10]]
```

Например, если защищенный блок — это [Error raiseSignal], возвращается 10, если защищенный блок — это [Warning raiseSignal], возвращается 5.

Большинство классов исключения наследуют характер возобновляемости из суперкласса. Чтобы определить новый класс возобновляемого исключения, надо инициализировать его переменную экземпляра isResumable значением true.

8.6. Явный выход из обработчика

Если возникает потребность контролировать передачу управления между множествами обработчиков исключений, следует использовать сообщения, посылаемые аргументу блока обработки, чтобы завершить процесс выполнения блока обработки раньше, чем он достигнет последнего выражения, или прерывать процесс выполнения и возвратиться к нему позже.

exit, exit: — в случае возобновляемого исключения послать исключению собщение resume, восстанавливая среду, в который возникло исключение, и продолжая выполнение; в случае невозобновляемого исключения послать сообщение return, урезая среду исключений до среды исключений активного обработчика (это расширение VisualWorks ANSI-спецификации).

resume, resume: — попытаться продолжить обработку защищенного блока, начиная с сообщения, которое расположено после сообщения, вызвавшего исключение (это возможно только для возобновляемых исключений).

return, return: — завершить процесс выполнения защищенного блока, который вызвал исключение.

retry — заново выполнить защищенный блок.

retryUsing: — выполнить новый блок-аргумент вместо защищенного блока.

resignalAs: — см. раздел 8.7.

pass — выйти из текущего обработчика и перейти к следующему внешнему обработчику; управление не возвращается к передавшему его обработчику.

outer — подобно сообщению pass, за исключением того, что восстанавливает управление, если внешний обработчик продолжает выполнение.

Сообщения exit:, resume:, return: в качестве возвращаемого значения возвращают свои аргументы, вместо значения последнего выражения в блоке обработки.

Рассмотрим примеры посылки перечисленных сообщений.

```
[Error raiseSignal]
        on: Exception
        do: [:exception |
            exception isResumable
                ifTrue: [exception exit: 5].
            Dialog warn: 'Nonresumable exception']
```

Поскольку экземпляр класса Error — невозобновляемое исключение, отображается предупреждающее об этом диалоговое окно. Заменяя защищенный блок блоком [Notification raiseSignal] и выполняя выражение снова, получим в качестве возвращаемого значения 5.

Если аргумент блока обработки — возобновляемое исключение, вместо сообщение exit можно использовать сообщение resume, которое для возобновляемых исключений ведет себя точно так же, как и exit. Попытка послать сообщение resume невозобновляемому исключению вызывает ошибку “attempt to proceed” (попытка продолжения).

Следующее выражение возвратит строку 'Value from protected block', значение последнего выражения в защищенном блоке, если поднятое исключение является возобновляемым и поднимет исключение, сообщающее об ошибке, если поднятое исключение не возобновляемо:

```
[someExceptionClass raiseSignal.'Value from protected block']
```

```
        on: Exception
        do: [:ex | ex resume: 'Value from handler'].
```

Чтобы завершить выполнение и выйти из блока, вызвавшего исключение, следует в блоке обработки послать исключению сообщение return. Когда оно посылается возобновляемому исключению, то вынуждает передачу управления, приводящую к выходу из защищенного блока. Таким образом, сообщение return может имитировать поведение невозобновляемого исключения, когда само исключение реально возобновляемо. Сообщение return урезает среду исключения до среды исключения активного обработчика.

Например, следующее выражение возвращает строку 'Value from handler' как значение сообщения on:do:, независимо от характера поднятого исключения:

```
[someExceptionClass raiseSignal. 'Value from protected block']
        on: Exception
        do: [:ex | ex return: 'Value from handler']
```

Другой способ выйти из блока обработки — послать сообщением retry, которое заканчивает работу блока обработки и пробует снова выполнить блок, являющийся приемником сообщения on:do:. Например, следующий метод попробует выполнить защищенный блок снова после возникновения ошибки деления на нуль (division-by-zero):

```
[^ x/y]
        on: ZeroDivide
        do:[:exception |
            "Сделать делитель маленьким, но большим 0."
            y := 0.00000001.
            exception retry]
```

Сообщение retry, когда оно повторяет процесс выполнения, урезает среду исключения до среды исключения активного обработчика.

Сообщение retryUsing: производит повторение выполнения, но вместо защищенного блока выполняет блок, переданный в качестве аргумента. Например:

```
[^ x/y]
        on: ZeroDivivde
        do: [:exception|
             exception retryUsing: [^ x=0
                                ifTrue: [#indeterminancy]
                                ifFalse: [#infinity]]]
```

Сообщение retryUsing:, когда повторяет процесс выполнения, также урезает среду исключения до среды исключения активного обработчика.

Сообщения pass может использоваться внутри блока обработки, чтобы завершить выполнение блока обработки и выполнить любые объемлющие блоки обработки для текущего исключения. Сообщение pass устанавливает среду исключения равной среде обработчика, которому передается управление. Расмотрим пример:

```
[n/m]
        on: ZeroDivide
        do: [:exception|
            "0/0 = 1; иначе raiseSignal для исключения ZeroDivide."
            exception dividend ~= 0
            ifTrue: [exception pass]
            ifFalse: [exception return: 1]
```

В этом примере случай 0/0 обрабатывается особо. Если делимое — нечто, отличное от нуля, то управление переходит к исключению ZeroDivide. Управление никогда не возвращается отправителю сообщения pass.

8.7. Преобразование исключений

Иногда, обработчик исключения должен иметь возможность преобразовывать одно исключение в другое. Например, в некоторых случаях низкоуровневое исключение, связанное с ошибкой операционной системы, должно быть преобразовано в высокоуровневое исключение пользователя.

Здесь нужна осторожность, чтобы случайно не выполнить не тот обработчик. Дело в том, что среда исключений в пределах обработчика, поднятого исключением низкого уровня, не обязательно такая же, как среда исключений, поднятая исключением высокого уровня. Эта проблема решается использованием в блоке обработки сообщения resignalAs: вместо сообщения raiseSignal. Например:

```
[low-level I/O]
        on: OperatingSystemException
        do: [:ex |
                ex errorCode = -213
                    ifTrue: [ex resignalAs: EndOfFile new]
                    ifFalse: [ex resignalAs:
                        [Error new messageText: 'OS Error']]
```

Сообщение resignalAs: прерывает работу текущего обработчика исключения, восстанавливая исключение и среды выполнения в то первоначальное состояние, в котором они были, когда исключение, которое является приемником сообщения resignalAs: было первоначально поднято. После восстановления, поднимается исключение, которое является аргументом resignalAs:. Это приводит к тому, что исключение, поднятое блоком- аргументом, будет

функционировать так же, как если бы оно было первоначально поднято на месте приемника.

8.8. Развертывание защиты

Возникновение исключения обычно приводит к прекращению работы приложения. Иногда в приложении, независимо от того, возникает исключение или нет, должны выполниться некоторые действия. В этом случае, используется специальный механизм обработки исключения, определенный в классе BlockClosure.

Когда выполнение некоторого блока выражений может завершиться преждевременно, следует защитить этот блок и использовать сообщение ifCurtailed: с блоком-аргументом, содержащим выражения, предотвращающие разрушительные для программы последствия такого завершения. Этот блок часто называют блоком зачистки.

Чтобы выполнить блок зачистки, независимо от характера (нормального или аварийного) выхода из защищённого блока, используется сообщение ensure:.

ifCurtailed: terminationBlock

Выполнить блок-получатель(защищаемый блок) и возвратить его результат. Если происходит аварийное завершение защищаемого блока, выполнить блок зачистки terminationBlock, но от возвращаемого им значения отказаться.

ensure: terminationBlock

Выполнить защищаемый блок и возвратить его результат. Сразу после успешного выполнения защищаемого блока, но перед возвращением его результата, выполнить terminationBlock. Если произошло аварийное завершение защищаемого блока, выполнить terminationBlock. В любом случае отказаться от значения, возвращенного блоком зачистки.

8.9. Контрольные вопросы

1) Что такое исключение или особая ситуация?
2) Экземплярами каких классов представляются исключения?
3) Что такое метод обработки (обработчик) исключения?
4) К чему приводит обработка по умолчанию большинства исключений?
5) Через какое сообщение определяется обработчик исключения?
6) Что такое блок обработки и защищенный блок?
7) Как при обработке исключений используется иерархия классов исключений?

8) Как установить обработчик исключений так, чтобы он обрабатывал несколько исключений, не обязательно входящих в одну ветвь иерархии?
9) Как создается (поднимается, вызывается) исключение в коде?
10) Что такое среда исключений процесса? Как она формируется и изменяется?
11) Что такое возобновляемые и невозобновляемые исключения?
12) Как проверить, является исключение возобновляемым или нет?
13) Какие команды испоьзуются для явного выхода из обработчика, и каковы их особенности?
14) Как в приложении выполнить некоторые обязятельные действия, независимо от того, возникает исключение или нет? Какой класс обеспечивает такую возможность? Что такое блок зачистки?

Глава 9

Поставка приложения

Когда разработка приложения завершена, его следует извлечь из среды разработки VisualWorks и подготовить к выполнению в виде автономного приложения. Этот процесс называют поставкой (развёртыванием) приложения. В рамках поставки приложения нужно выполнить следующие основные операции:

- подготовить приложение к автономному выполнению, удаляя из него зависимость от среды разработки;
- поместить код в парселы поставки;
- сформировать образ поставки.

Чтобы упростить процесс подготовки образа и его последующую инсталляцию в операционную систему конечного пользователя, VisualWorks включает:

- инструмент Runtime Packager для создания образа поставки из образа разработки;
- среду инсталляции приложения, которая используется в программе инсталляции VisualWorks (чтобы познакомиться с этой средой, следует загрузить парсел VWInstaller).

9.1. Выбор стратегии поставки

Существуют три способа организации приложения для его поставки:

1) в виде отдельного образа поставки, содержащего весь код приложения;
2) в виде одного или нескольких отдельно загружаемых парселов, содержащих код приложения, и поставляемых вместе с минимальным образом поставки;
3) в виде комбинации образа, содержащего начальную часть кода приложения, и парселов, содержащих остальную часть кода.

Каждый подход имеет свои преимущества.

Отдельный файл образа обычно используют для небольших приложений. В этом случае для поставки требуется только двигатель объектов и файл образа. Однако, отдельный загрузочный модуль часто оказывается слишком большим даже для простого приложения и неудобным для организации соответствующей поддержки подсистем или подприложений.

Парселы — файлы, содержащие объекты приложения, которые, не используя компилятор, можно быстро загрузить в образ. Парселы обычно используют для поставки больших и сложных приложений. Парселы позволяют:

- поставить минимальный образ, достаточный для его запуска;
- обновлять приложение, не поставляя новый образ;
- настраивать приложение во время выполнения;
- планировать объём памяти, занимаемой выполняющимся приложением.

Комбинированная поставка. Хотя загрузка парсела проходит очень быстро, загрузка образа проходит ещё быстрее. Загрузка всего приложения из парселов в минимальный образ может оказаться по ряду причин неоптимальной. Комбинированное использование образа и парселов позволяет сохранить в образе основной код приложения, нужный, по крайней мере, для запуска первого окна. После чего, нужный код можно подгружать из парселов по мере необходимости.

Для поставки приложения нужны следующие файлы:

- исполняемый файл виртуальной машины (файл `visual.exe` или `vwnt.exe`, возможно, переименованный) и все необходимые файлы поддержки (обычно, соответствующий dll-файл, на платформе Windows 2000 это `vwntoe.dll`);
- образ поставки созданного приложения;
- парселы, поставляемые с VisualWorks, и/или парселы сторонних производителей, которые будут устанавливаться во время выполнения созданного приложения (например, парселы поддержки базы данных).

Развертывание в виде отдельного файла

В Windows есть возможность объединить образ и виртуальную машину в автономно выполняемый файл. Для этого существует программное обеспечение сторонних производителей, позволяющее добавить нужные файлы к выполняемому файлу виртуальной машины как ресурсы. Например, программа `ResHacker.exe`, которая вместе с инструкцией `WindowsPackaging.txt` находится в каталоге `packaging\win\`.

Для построения программы инсталляции поставляемого приложения следует использовать среду инсталляции VisualWorks, которая становиться доступной после загрузки парсела VWInstalierFramework.

9.2. Подготовка к поставке приложения

Загрузка кода приложения в образ разработки

Код приложения может содержаться в образе и/или в парселах. Если приложение разрабатывается непосредственно в образе разработки и сохраняется в нём, то его код готов к обработке инструментом Runtime Packager для поставки в виде единственного образа. Если часть кода приложения должна загружаться в виде парселов, следует создать их, а затем работать с таким кодом как с кодом, содержащимся в парселах.

Если код приложения сохраняется в виде текстовых файлов командой File Out, его следует загрузить в образ командой File In и продолжить работу с приложением непосредственно в образе.

Вообще говоря, код, заключенный в парселы, тоже нужно загрузить в образ перед запуском инструмента Runtime Packager. Это позволит последнему включить этот код для просмотра зависимостей при определении того, какой код следует сохранить или удалить из образа. Runtime Packager также позволяет сохранять парселы как парселы времени выполнения, которые оптимизированы и сохраняются без исходного текста.

Для каждого парсела, загруженного в образ, следует определить, сохранять ли его код с образом поставки (и потому не использовать файл парсела) или во время выполнения приложения он будет загружаться в образ. Для парселов, загружаемых во время выполнения, следует определить и другую информацию. Она устанавливается в инструменте Runtime Packager на этапе Set Common Options на странице Parcels.

Важно спланировать расположение парселов. Может существовать специальный путь к парселам приложения или его может не быть, когда все парселы содержатся в том же каталоге, в котором расположен образ. По умолчанию, инструмент Runtime Packager не устанавливает путь к парселам (выбран флажок Clears parcel search path на странице Details этапа Set common options). После определения стратегии размещения парселов, следует указать необходимую информацию на странице Parcels) этапа Set common options.

Если пути к парселам определяются относительно каталога инсталляции VisualWorks ($VISUALWORKS), нужна стратегия установки этого каталога.

Удаление файлов исходного текста

По умолчанию, файлы исходного текста включаются в образ поставки. Иногда это не желательно, как в целях экономии дискового пространства, так и из соображений безопасности. Чтобы отделить файлы исходного текста, следует в образе разработки выполнить выражение

```
SourceFileManager default removeAllSources.
```

Чтобы выборочно удалить некоторые файлы исходного текста, следует послать сообщение removeFileAt:. При этом нужно знать индекс удаляемого файла, который можно узнать, просматривая в окне инспектора результат выполнения кода

```
files := OrderedCollection new.
SourceFileManager default fileIndicesDo:
    [:index| files add: (SourceFileManager default fileAt: index)].
^ files.
```

Чтобы удалить файл по его индексу, следует выполнить выражение вида

```
SourceFileManager removeFileAt: 2.
```

Окно Transcript

Объект Transcript сохраняется в образе поставки, но не отображается, как в образе разработки. Сообщения, посланные Transcript продолжают обрабатываться без ошибок, но визуально не отображаются, пока не будет определено окно, позволяющее показывать состояние объекта Transcript.

Обработка ошибок

Приложение должно перехватывать большинство ошибок и обработать их. Для необрабатываемых ошибок, Runtime Packager заменяет вызов окна NotifierView на вызов окна DeploymentNotifier. Это упрощенное окно уведомлений, исключающее поддержку инструментов, типа отладчика, а только сообщающее пользователю, что возникло необработанное исключение с его кратким описанием. Все сообщения об ошибках записываются в специальный файл ошибок, который по умолчанию имеет имя `error.log`.

Как класс обработки исключений, так и имя специального файла ошибок определяются на странице Exceptions этапа Options. Можно создавать собственные процедуры обработки для необработанных исключений, и определять их на этой странице.

Обработка системных событий

Часто приложение должно иметь особое поведение при запуске системы или выходе из неё. Есть два механизма, обычно используемых для этой цели. Один базируется на псевдокомментариях (pragma), а другой основан

на сообщениях, но оба позволяют зарегистрировать сообщения в качестве иждивенца событий системы. Механизм, основанный на псевдокомментариях предпочтительнее, поскольку автоматически устанавливает зависимость при загрузке парсела, и уничтожает её при его выгрузке.

Для обоих механизмов, событие системы — событие, посланное из ObjectMemory которое представляется одним из следующих символов: #aboutToQuit, #aboutToSnapshot, #earlySystemInstallation, #returnFromSnapshot, #scavengeOccurred. Обе механизма поддерживают регистрацию только унарных селекторов.

Наибольший интерес представляют два события — #returnFromSnapshot и #aboutToQuit. Событие #returnFromSnapshot возникает при запуске системы, после того как инициализированы все подсистемы VisualWorks. Это событие можно использовать для того, чтобы выполнить некоторые действия, например, запустить приложение. Событие #aboutToQuit возникает перед закрытием системы. Его можно использовать для того, чтобы выполнить операции, подобные закрытию сетевых связей, открытых приложением.

Механизм, основанный на псевдокомментариях, используется при определении метода (в категории методов класса dependencies-pragma) в классе SystemEventInterest. Примером такого метода является метод

```
startMyApplication
    <triggerAtSystemEvent: #returnFromSnapshot> "псевдокомментарий"
    MyApplication open
```

Когда возникает событие системы #returnFromSnapshot, класс SystemEventInterest посылает себе сообщение #startMyApplication, которое посылает сообщение open классу MyApplication.

Этот тип зависимости регистрируется всякий раз, когда метод с таким псевдокомментарием компилируется или загружается в класс SystemEventInterest, и регистрация аннулируется, когда метод либо заново компилируется, но без псевдокомментария, либо удаляется, или выгружается из системы.

Механизм, основанный на сообщениях, кроме определения события системы, должен определить приемник и посылаемый ему селектор. Например, выражение

```
SystemEventInterest
    atSystemEvent: #returnFromSnapshot
    send: #start
    to: anObject
```

зарегистрирует заинтересованность в событии #returnFromSnapshot и, когда объект ObjectMemory вызовет событие #returnFromSnapshot, сообщение #start будет послано объекту anObject.

Чтобы уничтожить такую зависимость, используется выражение

```
SystemEventInterest
    removeDependencyOnSystemEvent: #returnFromSnapshot
    selector: #start
    receiver: anObject
```

Чтобы для объекта anObject уничтожить все зависимости, основанные на сообщениях, надо выполнить выражение

```
SystemEventInterest removeAllDependenciesFor: anObject.
```

Механизмы, основанные псевдокомментариях и на сообщениях, используются и в других ситуациях (см., например, [9, p.20-11 – 20-14]).

Закрытие приложения при закрытии последнего окна

В образе разработки следует явно указывать, что следует выйти из среды VisualWorks и закрыть её. Однако, поставленное конечному пользователю приложение, как правило, должно закрываться, когда закрывается его последнее окно. Такая возможность явно настраивается в инструменте Runtime Packager. Но существуют специальные методы управления процессом закрытия образа, позволяющие приложению на стадии завершения работы выполнить специальные операции (например, закрыть связь с базой данных). Для этих целей приложение может зарегистрировать специальный блок, который будет выполняться при закрытии приложения. Блок регистрируется при выполнении выражения вида

```
RuntimeManager quitBlock: applicationShutdownBlock,
```

где applicationShutdownBlock — блок без аргументов или блок с одним аргументом. Если блок имеет аргумент, блок будет обеспечиваться одним из следующих значений, в зависимости от причины закрытия образа:

normal — закрытие вызвано последним закрываемым окном или нет окна приложения, которое было бы открытым после завершения запуска образа.

exception — закрытие вызвано необработанным исключением или другой ошибкой.

Выгрузка инструментов, загруженных из парселов

Даже при том, что Runtime Packager в процессе работы удалит классы инструментов разработки, перед запуском инструмента Runtime Packager желательно предварительно удалить вручную те инструменты разработки, которые были загружены из парселов (например, инструмент UIPainter). Выгрузка таких парселов позволит быстрее очистить образ и упростит процедуру подготовки образа поставки приложения.

Удаление необъявленных переменных

Среда VisualWorks для хранения необъявленных переменных поддерживает пространство имён Undeclared. Runtime Packager как часть операции подготовки поставляемого образа, просматривает и удаляет такие ссылки, но их можно удалить и перед запуском Runtime Packager. Пространство имён Undeclared должно быть пустым в поставляемом образе.

Напомним, что можно исследовать содержимое пространства имён Undeclared, открывая на нём окно инспектора (например, вводя Undeclared в рабочее окно и выбирая команду inspect из меню операций). Инспектор содержит команды для просмотра переменных и поиска методов, которые обращаются к выбранной переменной. Когда есть полная уверенность, что никаких ссылок на переменную более не существует, её можно удалить, используя команду remove.

Неактивные экземпляры как мусор

Вполне возможно, что во время длительной разработки приложения, в образе разработки сохраняться экземпляры, которые перед поставкой приложения надо бы собрать как мусор, чтобы добиться минимального размера образа. Методы обнаружения и уничтожения таких экземпляров рассматривались в разделе 6.9.

Замена заставки и звукового сопровождения

Можно изменить заставку и звуковое сопровождение, которые возникают при запуске приложения (но не следует изменять их в образе разработки!). Для их замены на платформе Windows, надо просто заменить один или оба файла `herald.bmp` и `herald.wav` в подкаталоге `\bin\win`.

Часто надо не заменить заставку и звуковое сопровождение, а подавить их появление. Самый простой способ добиться этого в RuntimePackager состоит в том, чтобы на этапе Set Common Options, на странице Details, оставить выбранным флажок Suppress splash screen and herald sound (Подавить появление заставки и звукового сопровождения).

9.3. Создание поставляемого образа

Проделав всю необходимую подготовку приложения и образа разработки, можно запускать инструмент Runtime Packager (см. раздел 2.5 и рисунок 2.9), который в результате создаст поставляемый файл образа, содержащий объекты, требуемые только для работы приложения.

Интерфейс пользователя инструмента Runtime Packager позволяет поэтапно создавать образ поставки, предоставляя общее описание каждого этапа и детальную справочную информацию о нём. Основная процедура

состоит из следующих этапов:

1) Clean Up Image (Очистить образ). Проверить образ на наличие посторонних глобальных объектов.
2) Set Common Options (Установить общие опции). Определить параметры, используемые на последующих этапах.
3) Specify Items to Keep and Delete (Определить элементы для сохранения и удаления). Настроить элементы, которые должны быть сохранены в поставляемом образе.
4) Scan for Unreferenced Items (Обзор элементов без ссылок). Просмотреть образ с целью обнаружения классов, методов и глобальных переменных, на которые нет ссылок.
5) Review Kept Items (Обзор сохраняемых элементов). Рассмотеть еще раз результат предыдущего просмотра.
6) Save Loadable Parcels (Сохранить загружаемые парселы). Сохранить все парселы, необходимые для образа времени выполнения.
7) Test the Application (Протестировать приложение). В интерактивном режиме обнаружить пропущенные ссылки на классы и методы приложения.
8) Set Runtime Memory Parameters (Установить параметры памяти времени выполнения). Установить размер различных частей пространства памяти для образа поставки (в том числе и для запуска).
9) Strip and Save Image (Произвести разбор и сохранить образ). Создать образ для автономного выполнения приложения.

После создания, образ поставки запускается так же, как и образ разработки (см. раздел 2.1). Но, создавая образ приложения, нужно организовать запуск самого приложения после загрузки образа. Для этого инструмент Runtime Packager позволяет использовать одну из следующих возможностей.

- В окне Runtime Packager, на странице Basics этапа Set common options, определить класс (Startup Class) и метод класса (Startup Method). Метод посылается классу после запуска образа и загрузки всех предписанных парселов.
- Если приложение во время запуска загружает хотя бы один парсел, приложение можно открыть, определяя для последнего загружаемого парсела блок с выражением запуска приложения для свойства Postload Action.

Обе эти возможности демонстрируются ниже.

Короткая процедура создания образа поставки

Основная процедура построения образа для автономного выполнения приложения может быть довольно долгой, и не всегда нужно выполнять каждый этап. Команда меню File → Package Runtime Image создаёт образ времени выполнения одной операцией, автоматически выполняя этапы по просмотру ненужных элементов, сохранению загружаемых парселов, разбору и сохранению образа. Но и здесь следует установить опции, особенно для определения того, как управлять парселами.

Короткая процедура создания образа для поставки приложения может оказаться хорошим подспорьем для понимания процесса создания образа и файла его параметров, определяющих разные особенности инструмента. В любое время, но до этапа разбора и сохранения образа, можно сохранить установленные ранее параметры. Для этого надо выбрать в окне инструмента Runtime Packager команду меню File → Save parameters. . . , создавая файл с расширением *.rtp, содержащий смолтоковский код определениий параметров. Чтобы загрузить файл параметров, следует воспользоваться командой меню File → Load parameters. . . .

Используя приложение RuntimeExample, загружаемое как часть парсела Runtime Packager, рассмотрим короткие примеры построения образа поставки.

Построение автономного образа

Построим образ приложения в виде единого образа, содержащего весь код. Это — самая простая процедура. Здесь должны быть установлены только опции Startup Class и Startup Method.

1) Загрузить и запустить обычным способом инструмент Runtime Packager в чистом образе.
2) Выполнить этап Clean up image.
3) Выполнить этап Set common options. На странице Basics установить:
   Startup Class: RuntimePackager.RuntimeExample
   Startup Method: open
   Runtime Image Page Name: runtime1
   Закрыть окно Common Option.
4) Выполнить этап Scan for unreferenced items, чтобы отыскать классы и методы, которые следует сохранить и удалить.
5) Выполнить этап Review kept classes and methods.
   Можно просмотреть результат. В панели Packages/Bundles выбрать RuntimePackager, а в панели Kept Classes/Globals — RuntimeExample. Все методы, определенные в классе RuntimeExample, кроме postLoadActionFor:, находятся в панели Kept Methods. Исключенный метод,

предназначенный для использования, когда приложение загружается как парсел, здесь не используется, поскольку явно определяется класс и метод запуска приложения. Закрыть окно.

6) Выполнить шаг Strip and Save Image.

Образ будет сохранён в виде файла `runtime1.im`. Теперь, чтобы запустить образ на выполнение, следует переместить файлы `visual.exe`, `runtime1.im` в один каталог и выполнить команду

visual.exe runtime1.im.

Построение образа, использующего парселы

Построение приложения в виде базового образа с загружаемыми парселами во время запуска проще всего создать, загружая сначала все парселы, используемые в приложении, в образ разработки. Поставляемый образ и его парселы будут затем создаваться из образа разработки.

Парселы, загружаемые в образ среды, определяются опциями командной строки, перечисляемыми индивидуально или в файле конфигурации. Когда запускается образ приложения, он ищет в стартовом каталоге файл конфигурации парселов с именем `imagename.cnf`, где `imagename` то же имя, что и у файла образа. Если такой файл существует, образ загружает перечисленные в нём парселы. Имена файлов парселов должны перечисляться по одному на каждой строке, с указанием пути относительно рабочего каталога.

Можно, используя аргументы командной строки, определить дополнительно загружаемые парселы (-pcl filename) и файлы конфигурации парселов (-cnf filename).

Парселы, которые будут загружаться во время выполнения приложения, указываются на этапе Set Common Options, на странице Parcels. После просмотра не упоминаемых элементов, если таковые есть, уже можно создавать версии парселов, загружаемых образом приложения. Инструмент RuntimePackager позволяет проанализировать парселы на предмет обнаружения неиспользуемых классов и методов.

Чтобы проиллюстрировать процесс построения образа, в котором приложения загружаются через парселы, рассмотрим процедуру создания образа поставки для приложения RuntimeExample. В класс приложения добавлен метод postLoadActionFor:, позволяющий проиллюстрировать открытие приложения с помощью свойства Post-load Action. Начать работу следует с построения парсела, и только потом строить поставляемое приложение.

1) Загрузить и запустить Runtime Packager.
2) Выполнить этап Set common options и определить:

Runtime Image Path Name на странице Basics: runtime2

3) На странице Parcels окна Common Option, нажать кнопку New Parcel и ввести RuntimeExample создаваемого нового парсела.
4) Открыть браузер системы и установить его на отображение парселов (выбрать команду Browser → Parcel). В браузере выбрать парсел RuntimePackager и класс RuntimeExample, а затем выбрать команду Class → Move → All to Parcel..., чтобы переместить этот класс в парсел RuntimeExample.
5) В панели свойств парсела RuntimeExample выбрать страницу Properties и определить блок, выполняемый после загрузки парсела (блок Post-load Action):

```
[:pkg | #RuntimePackager.RuntimeExample
        value postLoadActionFor: pkg ]
```

6) В окне Common Option инструмента Runtime Packager, на странице Parcels, выбрать парсел RuntimeExample и сделать следующее:

   Parcel is loaded into image at runtime: yes (выбрать)
   Strip unreferenced items and save: yes (выбрать)
   Path name: RuntimeExample.pcl

   Закрыть окно Common Option.
7) Выполнить этап Scan for unreferenced items.
8) Выполнить этап Save rintime loadable parcels, записывая файл `RuntimeExample.pcl` в текущий каталог.
9) Выполнить этап Strip and save image.

Образ времени выполнения сохраниться как файл `runtime2.im`. Чтобы запустить его, загрузить созданный парсел и запустить приложение, следует выполнить команду:

```
visual.exe runtime2.im -pcl RuntimeExample.pcl
```

Подробное описание меню и команд инструмента Runtime Packager можно найти в [9, с.20-21 – 20-40].

9.4. Контрольные вопросы

1) Что означает поставка приложения, созданного в среде VisualWorks, конечному пользователю?
2) Какие основные операции следует выполнить в рамках поставки приложения?
3) Какие существуют стратегии поставки приложения?
4) Какой инструмент можно использовать для создания образа поставки приложения из образа его разработки?

5) Какие подготовительные дейстаия следует предпринять в образа разработки приложения перед созданием его образа поставки?
6) Из каких этапов состоит процесс создания образа поставки инструментом Runtime Packager?
7) Как запускается образ поставки приложения?
8) Как организуется запуск приложения после загрузки его образа поставки?

Предметный указатель

Литература

1. Draft American National Standard for Information Systems — Programming Languages — Smalltalk. — Revision 1.9. — NCITS, 1997.
2. *Goldberg A., Robson D.* Smalltalk-80: The Language and its Implementation. — Reading, MA: Addison-Wesley, 1983.
3. *Goldberg A., Robson D.* Smalltalk-80: The Language. — Reading, MA: Addison-Wesley, 1989.
4. *Hopkins T., Horan B.* Smalltalk: An introduction to application development using VisualWorks. — Prentice-Hall, 1995.
5. *Lalonde W., Pugh J.* Smalltalk/V: Practice and Experience. — Prentice Hall, 1994.
6. *Lewis S.* The Art and Science of Smalltalk. — Prentice Hall, 1995.
7. *Linderman M.* Developing Visual Programming Applicftions Using Smalltalk. — New York: SIGS Books, 1996.
8. *Sharp A.* Smalltalk by Example. The developer's guide. — McGraw-Hill, 1997.
9. VisualWorks: Application Developer's GuideVisual. — Cincom, 2005.
10. VisualWorks: Source Code Management Guide. — Cincom, 2003.
11. VisualWorks: Taming Name Spaces. — Cincom, 2000 – 2001.
12. VisualWorks: Walk Through. — Cincom, 2000 – 2005.
13. *Бадд Т.* Объектно-ориентированное программирование в действии. — СПб.: Питер, 1997.
14. *Буч Г.* Объектно-ориентированное проектирование с примерами применения. — М.: Конкорд, 1992.
15. *Кирютенко Ю. А., Савельев В. А.* Объектно-ориентированное программирование. Язык Smalltalk. — М.: «Вузовская книга», 2007.
16. Смолток. Объектно-ориентированная система программирования. Руководство пользователя, часть 1, 2, 3. — М.: Ин-т проблем информатики РАН, 1995.

Список иллюстраций

Оглавление