МАТЕМАТИЧЕСКИЕ МОДЕЛИ
В ИНФОРМАТИКЕ
(Вычислительные системы)
2002 год Выпуск 169
УДК 519.682.2:519.682.6:519.685.1

"Zebra" — СИСТЕМА АВТОМАТИЗАЦИИ ОСНОВАННЫХ НА RUP ПРОЦЕССОВ СОЗДАНИЯ БОЛЬШИХ ПРОГРАММНЫХ СИСТЕМ

О.Ю.Чунихин

Введение

Данная статья посвящена более подробному описанию технологии "Zebra", упомянутой в предыдущей статье. Технология предназначена для сокращения затрат в процессах разработки больших программных продуктов, основанных на RUP, UML и Java.

Один из путей достижения этого — автоматизация, частичная или полная, каких-либо этапов создания приложения или его частей. На данном этапе своего развития технология "Zebra" предлагает инструмент полуавтоматического создания (генерации) java-кода по UML спецификации системы.

"Zebra" разработана в компании Novosoft Inc в рамках семейства проектов, направленных на повышение эффективности процессов разработки программного обеспечения в компании.

Для чтения статьи необходимо быть знакомым с основными принципами объектно-ориентированного подхода [1,4,7,8], обладать начальными знаниями UML [3,9,11] и Java. Так же, весьма полезным является знакомство с организацией процессов разработки программного обеспечения по стандартам RUP [2,6].

§ 1. Система "Zebra"

Технология "Zebra" разрабатывалась в основном с целью уменьшения общей стоимости разработки и поддержки программного обеспечения. Другая цель — это увеличение конкурентоспособность продуктов, разработанных с помощью данной технологии, за счет большей гибкости и возможности их подгонки под конкретные нужды клиента. Кроме того, использование "Zebra" подразумевает использование RUP или процесса разработки, основанного на RUP.

Эти цели достигаются за счет наличия у продукта следующих черт:

• Уменьшение количества ручного кодирования в проекте. В "Zebra" исходный код заменяется исходной моделью. Для описания внутренней структуры и логики используются UML и Java (Java в основном для описания элементарных операций в рамках сценариев и объектов описанных на UML). Преимущество этого подхода в том, что большое количество ручного кодирования заменяется просто уточнением дизайнерской модели (которая должна иметься в проекте в любом случае). Кроме того, это существенно ускоряет построение прототипов приложений.

• Четкое разделение программных уровней приложения [5]. Архитектура "Zebra" подразумевает выделение в приложении как минимум трех слоев:

1) уровень представления пользовательского интерфейса (User Interface presentation layer), где разработчик концентрируется на визуальном представлении компонент пользовательского интерфейса;

2) уровень логики пользовательского интерфейса (UI logic layer), где основные усилия разработчиков направлены на реализацию сценариев пользовательского интерфейса (последовательности экранов, которые видит пользователь), взаимодействия UI компонент, потоков данных и событий между ними, валидацию данных, их трансляцию в данные и события следующего слоя;

3) уровень логики приложения или операционной логики (Business Logic layer), на котором реализуются сценарии и операции предметной области, для которой создается система.

• Большие требования к циклу разработки приложения. Среди основных этапов цикла разработки, описанных выше, некоторые имеют при использовании "Zebra" особенное значение:

1) этап *анализа* случаев использования системы, на котором выделяются функциональные требования к системе и роли ее пользователей. Для их описания используются Use–case диаграммы языка UML, состоящие из актеров (actors), случаев использования (use–cases) и сценариев использования (scenarios) системы;

2) *прототипирование*. На этом этапе, обычно являющемся существенным для средних и больших проектов, строятся функциональные прототипы и прототипы пользовательского интерфейса. "Zebra" уменьшает стоимость как данного, так и разработки в целом за счет более быстрого построения прототипов и возможности переиспользования полученных на этапе прототипирования UML–моделей в дальнейшей работе над приложением;

3) *реализация*. Эта стадия очень сильно связана с этапом дизайна, если в проекте используется "Zebra". Фаза дизайна включает разработку бизнес–сценариев и операций (Business Logic modeling) и разработку логики пользовательского интерфейса (UI logic modeling). "Zebra" подразумевает визуальную разработку (с помощью какого–либо UML–редактора), что упрощает понимание логики приложения всеми членами команды, работающей над проектом, поддержку приложения, а также снижает стоимость работ по созданию документации (многие части модели, из которой в дальнейшем будет генерироваться код, будут понятны и без детальных описаний, которые обычно требуются, что бы сделать код, написанный вручную более прозрачным). Кроме того, генерация кода гарантирует, что все дизайнерские решения будут реализованы именно так, как это задумывалось.

• Упрощение поддержки приложения. Уменьшается стоимость поддержки приложения за счет того, что

1) облегчается обучение новых членов команды, поскольку структура приложения всегда (даже при отсутствии документации) явно специфицирована на UML;

2) упрощается отладка (малые модификации приложения) за счет требований к модульности разбиению на слои, которые процесс с использованием "Zebra" ставит перед разработчиками;

3) упрощается развитие приложения в условиях меняющихся требований к нему.

§ 2. Архитектура системы "Zebra"

Технически "Zebra" состоит из компонент времени разработки и компонент времени исполнения. Основной компонентой времени исполнения является генератор java-кода. В качестве входной информации он берет UML-модель в формате rational rose (rose petal files) [10] либо в универсальном XML-формате передачи метаинформации (XMI) и генерирует набор java-классов, реализующих описанное в модели поведение.

В принципе сгенерированные классы уже могут использоваться сами по себе, являясь частью некоторого приложения или полностью составляя его, но для получения всех преимуществ, которые дает "Zebra", их имеет смысл использовать с входящими в состав "Zebra" компонентами времени исполнения. Это в основном Zebra-сервер, Zebra-презентации, коннекторы презентаций и другие, более мелкозернистые компоненты, облегчающие создание приложений.

Архитектура "Zebra" изображена на рис. 1.

Как видно отсюда, классы приложений "Zebra" генерируются из UML-модели в java-классы, компилируются любым компилятором java и исполняются в некотором окружении, обеспеченном взаимодействием Zebra-сервера и одной или нескольких презентаций.

• Zebra-сервер обеспечивает стандартную реализацию таких общих для многих приложений сущностей, как сессии, сценарии работы с пользователями, функций управления презентациями, подсистема безопасности и др. На Zebra-сервере выполняются классы, реализующие операционную логику и логику пользовательского интерфейса.

• Презентация представляет собой активную прослойку между пользователем (клиентом) и Zebra-сервером. С точки зрения Zebra-сервера (и, соответственно, разработчика business и UI

logic) все презентации выглядят одинаково, доступ к их возможностям осуществляется через один java интерфейс Presentation. Это существенно облегчает замену одной презентации другой, и, следовательно, смену внешнего вида приложения, поскольку именно в рамках презентаций описывается UI presentation layer. В существующей версии Zebra реализованы JSP, XML и Swing презентации.

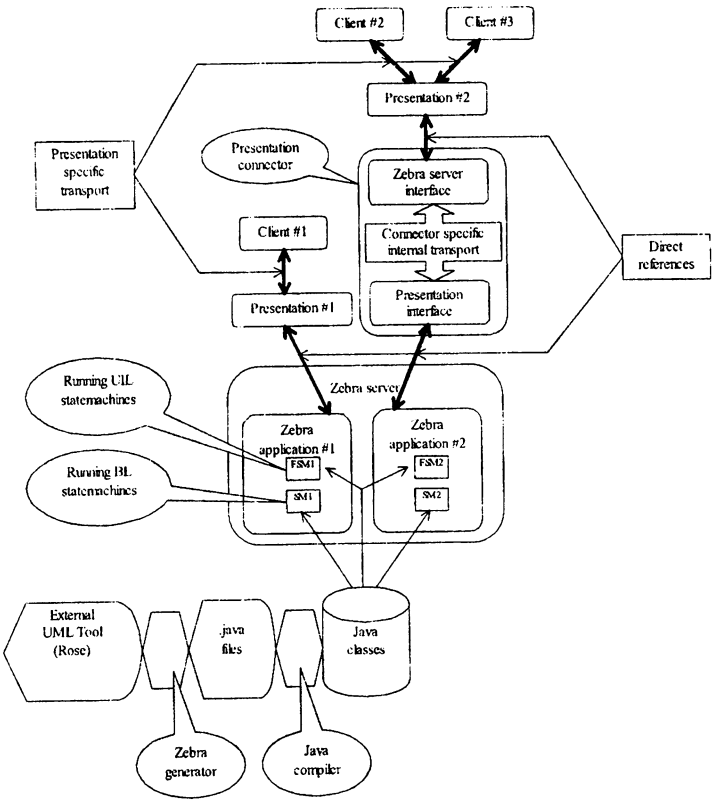


Рис. 1

• Коннекторы презентаций (presentation connectors) обеспечивают возможность работы Zebra-сервера и обычной презентации на разных машинах, что позволяет разрабатывать системы с "толстым" клиентом, основанные на различных протоколах. В существующей версии реализованы коннекторы презентаций для протоколов SOAP и JMS.

• Zebra-клиент полностью управляется приложением, работающим на сервере. В зависимости от презентации это может быть как просто Internet Browser (для JSP презентации), так и полнофункциональное приложение, общающееся с сервером по некоторому протоколу (например, JMS для JMS презентации).

§ 3. Структура приложения "Zebra"

Как уже упоминалось выше, одним из факторов, делающим использование "Zebra" эффективным, является то, что вместо исходного кода все разработчики работают напрямую с UML-моделью, постепенно доводя ее до уровня точной спецификации системы, по которой возможно полностью автоматически сгенерировать исходный код.

Таким образом, Zebra-приложение представляет собой обычную UML-модель, удовлетворяющую всем ограничениям языка UML и расширенную в соответствии с правилами расширения UML некоторыми конструкциями, учитывающими специфику генератора. Более подробно о языке "Zebra" буде рассказано далее, в этой же части статьи мы рассмотрим общую архитектуру приложений "Zebra".

Обычно приложение "Zebra" имеет следующую структуру, показанную на рис. 2.

Здесь:

• BL implementation — слой содержащий классы, реализующие операционную логику. Это обычно классы UML-модели со стереотипом "zbasic" (см. § 5) и другие классы и интерфейсы, возможно использующиеся в приложении, но не генерируемые генератором "Zebra" (например, классы из уже готовых библиотек).

• BL interface — слой интерфейсов, через которые следующий слой получает доступ к операционной логике. В не слишком сложных приложениях этот слой может отсутствовать.

• UI Logic — слой логики пользовательского интерфейса. Этот слой обычно реализуется с помощью классов со стереотипами "zbasic" и "zform". Так же неотъемлемой частью этого слоя получаются объекты классов со стереотипом "zform bean". Они являются посредниками между UI logic layer и UI presentation layer и представляют собой абстракции данных, передаваемых между визуальными компонентами пользовательского интерфейса и объектами, реализующими их поведение.

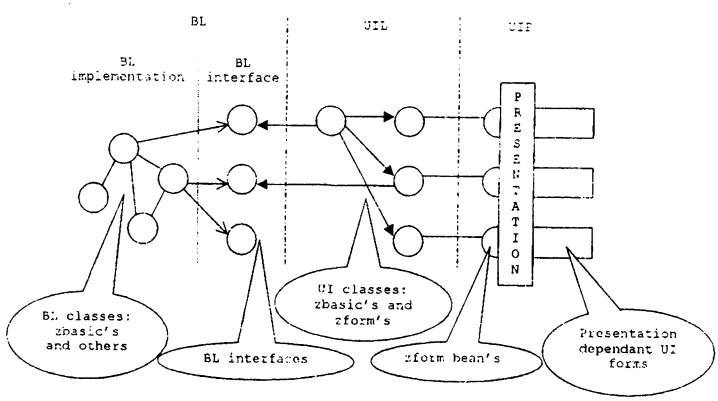


Рис. 2

• UI Presentation — этот слой реализуется презентацией и специфическими для данной презентации объектами-компонентами пользовательского интерфейса. Так, для JSP-презентации — это jsp-страницы, для Swing-презентации — компоненты библиотеки Swing и т.д.

§ 4. Пример создания приложения

Для примера здесь будет описан процесс создания простейшего приложения на "Zebra". Это приложение будет открывать окно (или показывать страничку, в зависимости от того, какая

презентация используется), в котором будет отображаться приветствие "Hello, world!". Более полно поведение приложения может быть описано следующим образом: приложение должно вывести текстовое окно с текстом "Hello, World" в нем. Так же в окне должна быть кнопка ok, при нажатии на которую окно закрывается, а приложение прекращает работу.

На первом этапе создается модель, в которой будет определена структура приложения. Поскольку основным инструментом разработки в "Zebra" сейчас является Rational Rose, просто создается и инициализируется новый документ Rational Rose (более подробно о создании чистой модели для "Zebra" можно прочитать в руководстве для разработчиков "Zebra").

Само приложение "Zebra" это просто несколько классов, поведение которых определено диаграммами состояний. В новой модели создаются два класса, связанных ассоциацией (см.рис. 3).

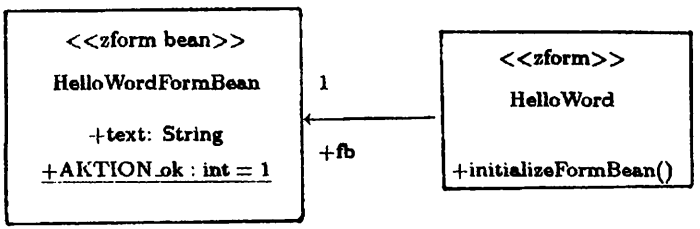


Рис. 3

Один из этих классов, HelloWorld, имеет стереотип "zform" и является простейшим контроллером элемента управления (в данном случае окна или html странички). Другой, HelloWorldFormBean, со стереотипом "zform bean", задает абстрактную структуру данного элемента пользовательского интерфейса.

Класс HelloWorldFormBean имеет два атрибута (поля):

1) атрибут text типа String для задания текста, выводимого в окне;

2) константа ACTION_ok типа int для задания кнопки. Такие атрибуты, начинающиеся на "ACTION_", по соглашению используются для описания того, что в данном элементе управления

пользователь может вызвать единичное действие, каким может, например, являться нажатие на кнопку.

Если класс HelloWorldFormBean примерно определяет внешний вид элемента управления, то класс-контроллер HelloWorld описывает его поведение. Ассоциация между классами HelloWorldFormBean и HelloWorld задает их связь. На классе HelloWorld определяется метод initializeFormBean(), вызываемый при создании объектов HelloWorld и HelloWorldFormBean. Этот метод будет состоять всего из одной строчки: "getFb().setText("Hello World");".

Теперь должно быть определено поведение окна. Для этого в классе HelloWorld задается диаграмма состояний (см. рис. 4).

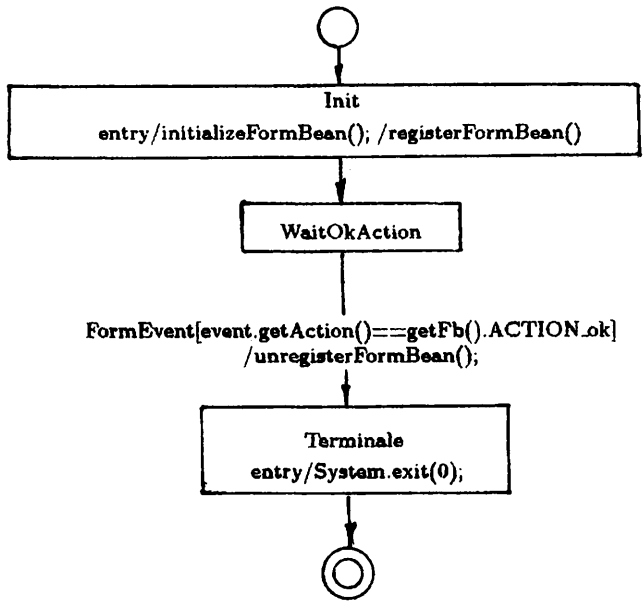


Рис. 4

Из диаграммы видно, каким будет жизненный цикл объекта HelloWorld.

1. Переход из начального состояния в состояние Init не помечен условием, с ним не связано выполнение действий (action).

Переход происходит немедленно. При входе в состояние Init выполняются (последовательно) два действия — вызовы методов initializeFormBean(), который инициализирует содержимое текстового окна, и registerFormBean(), объект класса HelloWorldFormBean регистрируется в презентации, что приведет к показу окна пользователю. В результате создается окно с текстом "Hello World" и кнопкой ok.

2. Происходит переход в состояние WaitOkAction. Никаких действий в этом состоянии не происходит, имеет место только ожидание перехода в состояние Terminate.

3. Переход из состояния WaitOkAction в состояние Terminate имеет стереотип <signal>, что означает, что он происходит только при получении определенного события, выполнении определенного условия и после выполнения определенного действия. В данном случае, переход активизируется, если происходит событие типа FormEvent. В этом случае происходит проверка условия — является ли произошедшее событие событием ACTION_ok класса HelloWorldFormBean. Если условие выполнено, то происходит вызов метода unregisterFormBean(), что приводит к закрытию окна.

4. В состоянии Terminate происходит вызов метода System.exit(0). В результате мгновенно прекращается работа JVM. Приложение завершило работу.

§ 5. Язык системы "Zebra"

Как уже упоминалось, язык "Zebra" это расширение UML + Java. Язык UML используется для описания статической структуры классов приложения (диаграммы классов) и поведения этих классов (диаграммы состояний классов и Java для элементарных операций классов). Язык Java является, с одной стороны, промежуточным языком генерации (именно java–код создается генератором "Zebra" по UML–модели), с другой стороны, Java используется в UML–модели для описания атомарных операций приложения (в полном соответствии с общей спецификацией языка UML, по которой для описания семантики различных элементов можно использовать различные языки, в том числе и естественный). Семантика UML не меняется, используются его стандартные механизмы расширения — стереотипы. Таким образом,

"Zebra" можно использовать совместно с другими инструментами обработки UML–моделей, придерживающихся спецификации UML.

Основными элементами языка являются типы и пакеты, классы, элементы описания обработки сигналов (event dispatching), машины состояний (statemachines) и стереотипы.

1. Типы и пакеты. "Zebra" поддерживает три класса типов — примитивные, параметризованные и типы классов:

- множество примитивных типов совпадает с множеством примитивных типов Java: byte, short, int, long, char, boolean, float и double. Все они должны быть представлены в UML–модели соответствующими UML–классами со стереотипом primitive;
- любой класс, со стереотипом, отличным от primitive, определяет соответствующий тип класса;
- множество параметризованных типов также совпадает с множеством параметризованных типов Java. На данный момент, это только параметризованный тип array, представляющий собой массив элементов. Это тип также должен быть представлен в UML–модели параметризованным классом со стереотипом primitive для нормальной работы генератора "Zebra".

Пакеты — средство, представляющее собой общий механизм организации элементов модели в группы и позволяющее разделять приложение на логически независимые блоки. Каждый такой блок обычно отвечает за определенную функциональность конечного приложения. Например, пакет logger может отвечать за вывод в текстовый файл информации о состоянии тех или иных частей приложения.

Все классы, как в Java, так и в UML, расположены в пакетах (packages), образующих иерархию вложенности. "Zebra" поддерживает естественное соответствие UML пакетов и Java пакетов, если другое соответствие явно не определено программистом.

2. Классы представляют собой совокупности объектов с общими атрибутами, операциями, отношениями и семантикой. Собственно классы — это словарь разрабатываемой системы так как именно с их помощью описываются программные и концептуальные сущности.

Каждый класс (как и любой другой элемент UML) может иметь стереотип. По спецификации UML стереотипы элементов используются для того, что бы определять важные семантические различия в множестве элементов одного метакласса. Так, например, случаи использования в UML могут быть связаны так называемыми зависимостями — ассоциациями специального вида. Что бы различать два существенно разных вида зависимостей, в UML определены стереотипы зависимостей: include и extend.

"Zebra" использует свободные (не зарезервированные в спецификации UML) стереотипы для той же цели: во–первых, наличие соответствующего стереотипа позволяет определить, подлежит ли класс вообще генерации генератором "Zebra", и во–вторых, указывает на назначение класса в рамках приложения и способ его генерации. В описании архитектуры приложения "Zebra" уже упоминались стереотипы zbasic, zform и zform bean. Кроме них для классов "Zebra" зарезервированы еще и стереотипы exception, interface и signal.

Так же как и в UML, в "Zebra" класс может быть связан отношением наследования (Generalization Association) с другими классами. Это означает, что полное описание класса (атрибуты, стереотип, операции) выводится путем объединения его собственного определения и определения наследуемого класса. "Zebra" не поддерживает множественное наследование, это обусловлено выбором языка Java.

Каждый класс характеризуется набором следующих свойств:

• имя — уникальный (в пределах одного пакета) идентификатор. Классы в разных пакетах могут иметь одинаковые имена. Идентифицировать их в этом случае можно по полным именам, составленным из имени самого класса и имен пакетов, в которые он входит, разделенных "::";

• стереотип служит для разделения классов с различной семантикой. В настоящее время "Zebra" поддерживает стереотипы exception, signal, interface, zbasic, zform и zform bean. Более подробно применение стереотипов будет рассмотрено в разделе Стереотипы классов;

• видимость может быть public, protected, private или package в соответствии со спецификацией языка Java;

• признак abstract может быть TRUE или FALSE в соответствии со спецификацией языка Java;

• признак final может быть TRUE или FALSE в соответствии со спецификацией языка Java;

• набор атрибутов определяющих состояние класса. Атрибут — это именованное свойство класса, определяющее часть состояния объекта класса. Проще говоря, — это некоторое свойство моделируемой сущности, общее для всех объектов данного класса. Класс может иметь любое число атрибутов или не иметь их вовсе. Каждый атрибут характеризуется следующими свойствами:

– имя — уникальный (в пределах одного класса) идентификатор атрибута,

– тип атрибута. Это может быть как примитивный или параметризованный тип так и другой класс,

– начальное значение, которое приобретает атрибут автоматически при создании экземпляра класса. Начальное значение может быть не определено,

– видимость, которая может быть public, protected, private или package в соответствии со спецификацией языка Java,

– признак final может быть TRUE или FALSE в соответствии со спецификацией языка Java,

– признак static может быть TRUE или FALSE в соответствии со спецификацией языка Java,

– признак wrappers может быть TRUE или FALSE. Если TRUE, то для этого атрибута в java-классе автоматически будет сгенерирована пара методов get и set для доступа к значению этого атрибута,

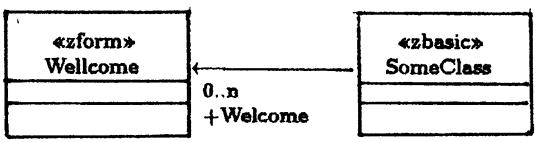
– признак firePropertyChanged может быть TRUE или FALSE. Если TRUE, то при изменении значения этого атрибута в процессе работы приложения объектом будет послано событие PropertyChanged и, таким образом, все объекты, зарегистрировавшиеся для получения этого события, будут извещены об этом.

• Ассоциации — различного рода отношения экземпляра класса с другими объектами. Ассоциация — это структурное отноше-

ние, отражающее тот факт, что объекты одного типа некоторым образом связаны с объектами другого типа. При наличии ассоциации между классами в UML–модели, "Zebra" сгенерирует в java–классах атрибуты, содержащие взаимные ссылки друг на друга. Возможен также случай, когда оба конца ассоциации относятся к одному и тому же классу, так называемая рекурсивная ассоциация. Класс, участвующий в ассоциации играет в ней некоторую РОЛЬ. Один и тот же класс может играть разные роли в различных ассоциациях. При моделировании важно указывать, сколько объектов может быть связано посредством одного экземпляра ассоциации. Это называется КРАТНОСТЬЮ и вводится как выражение, значением которого является диапазон значений. Значение кратности, указанное на одном конце ассоциации, означает, сколько объектов на этом конце может соответствовать одному объекту на противоположном конце.

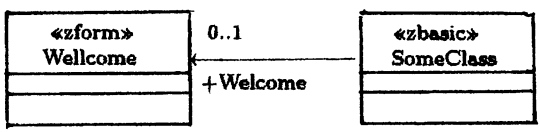
Рассмотрим подробнее использование ролей и кратностей на двух простых примерах.

1. Односторонняя ассоциация — это ассоциация, для которой имя роли указано только для одной стороны ассоциации. На следующем примере кратность этой роли "0..n", т.е. множественная ассоциация



В этом случае для класса SomeClass будет сгенерирован один атрибут с именем welcome типа java.util.Lis и для него два метода доступа getWelcome() и setWelcome(List arg).

Если для роли SomeClass ассоциации указать множественность 0..1,



то для класса SomeClass будет сгенерирован один атрибут welcome типа Welcome (т.е. простая ссылка) и для него два метода доступа getWelcome() и setWelcome(Welcome arg).

• Операции — набор методов, которые могут быть вызваны на объектах класса. "Zebra" предоставляет возможность определять операции как на этапе моделирования, внося java–код прямо в UML–модель, в поле операции semantics, так и позже, модифицируя сгенерированный java–код, а затем автоматически внося изменения в UML–модель.

Для добавления операции в класс, необходимо указать ее сигнатуру (имя, список параметров с типами и тип возвращаемого значения). Тип возвращаемого значения может не указываться, тогда будет сгенерирована java–функция типа void. Имя операции должно быть уникальным в контексте класса.

3. Посылка и обработка сигналов. Некоторые классы "Zebra", а именно классы со стереотипами zbasic и zform, имеющие или наследующие машины состояний, могут получать и обрабатывать сигналы. Все эти классы реализуют (таким образом генерируется их код) интерфейс Statemachine который, в свою очередь наследуется от интерфейса EventObjectListener с единственным методом dispatchEventObject, что позволяет объектам этих классов регистрироваться в качестве получателей сигналов.

То, каким образом объекты класса реагируют на сигналы, определяется машиной состояний класса. Полученное событие может инициировать переход из одного состояния объекта в другое. Более подробно этот механизм описан далее.

4. Машина состояний определяется как множество состояний, некоторые из которых связаны переходами. Машина состояний описывает поведение объекта в терминах последовательности состояний, через которые он проходит в течении своего жизненного цикла. Изменение его состояния (переход) может происходить в результате реакции на те или иные события.

Основные элементы машины состояний описаны ниже.

• Действие это атомарное вычисление, которое, в частности, может привести к изменению полного состояния объекта.

• Под состоянием объекта здесь мы будем понимать элемент множества состояний машины состояний. Состояние может быть

активным (или другими словами, объект находиться в данном состоянии) или неактивным. В отличие от формальных автоматов, в UML–состояния могут быть вложенными, что приводит к тому, что активными в один и тот же момент времени могут быть несколько состояний. Впрочем, это не усложняет структуру машины состояний, а лишь увеличивает выразительность модели. Поскольку состояния могут быть вложенными, они могут быть описаны в виде дерева состояний. Всегда существует корень дерева, или объемлющее состояние верхнего уровня, содержащее все другие состояния. Состояние называется простым, если оно не имеет вложенных подсостояний (или потомков) и составным в противном случае. В UML определено несколько простых ограничений, описывающих формальную правильность дерева состояний. Так, если какое–то состояние активно, то

1) его предок (если он есть) активен,

2) все его братья (если они есть) неактивны (в "Zebra" поддерживается упрощенная модель машин состояний UML, без так называемых параллельных состояний).

Важными элементами описания состояния являются действия, выполняемые при входе в состояние (entry actions), выходе из него (exit actions) и при нахождении в данном состоянии (do actions). "Zebra" реализует выполнение do actions порождая новый поток, в котором выполняются описанные действия. Этот поток либо завершает свою работу нормально, произведя все действия, либо прерывается, если состояние становится неактивным до его нормального завершения.

• Полное состояние объекта — это совокупность всех его свойств и их текущих значений и множество его активных состояний. К свойствам объекта относятся все его атрибуты и агрегированные части.

• Начальное состояние — это состояние активное в первый момент после создания объекта.

• Конечное состояние — это состояние, из которого невозможен переход.

На диаграмме состояний, состояния изображаются в виде прямоугольников с закругленными углами а переходы — в виде

стрелок. На диаграмме состояний должно присутствовать начальное и конечное (последнее не обязательно) состояния. Начальное состояние отображается закрашенным кругом, конечное — закрашенным кругом обведенным кольцом (см. пример приложения).

• Событие — это спецификация определенного факта, имеющего место в пространстве и времени. В контексте машины состояний это некий стимул, инициирующий переход объекта из одного состояния в другое.

• Переход — это элемент бинарного помеченного отношения на множестве состояний. Если существует переход из состояния в состояние, то при некоторых условиях возможна деактивация начального состояния и активация конечного состояния этого перехода. Переход определяется следующими элементами:

1) исходное и целевое состояния (иногда в контексте перехода их называют начальное и конечное состояния),

2) событие–триггер — событие, при получении которого объектом, находящимся в исходном состоянии, может произойти переход (при этом должно быть выполнено условие перехода),

3) условие — предикат, вычисляемый при возникновении возможности срабатывания перехода. Если значение этого предиката истинно, то переход может сработать. В процессе перехода может совершаться некоторое действие. Условие задается в приложении "Zebra" в виде java–выражения типа boolean,

4) действие — атомарное вычисление, которое может непосредственно воздействовать на объект или оказать косвенное воздействие на другие объекты, находящиеся в области видимости. Например, в процессе выполнения действия объект может посылать сообщения как какому либо конкретному объекту, так и широковещательные. Действие задается в приложении "Zebra" в виде Java выражения.

Переход может быть внутренним — в этом случае начальное и конечное состояния совпадают, при срабатывании такого перехода не происходит смена активного состояния объекта, только выполняются некоторые действия.

Переходы делятся на пять групп в зависимости от типов причин, их вызывающих: простые, по сигналу, по исключительной

ситуации, по вызову операции, по времени. В модели эти типы переходов различаются с помощью стереотипов.

1. Простые переходы, это переходы в которых не указано событие–триггер и стереотип. Этот переход происходит если объект находится в исходном (по отношению к этому переходу) состоянии и его условие выполняется.

2. Переходы по сигналу, это переходы со стереотипом signal и для которых указан конкретный тип сигнала в поле событие-триггер. Такой переход происходит если объект находится в исходном по отношению к нему состоянии, тип полученного сигнала совпадает с типом события-триггера и выполняется условие.

3. Переходы по исключению, это переходы со стереотипом exception и для которых указан конкретный тип исключения в поле событие–триггер. Такой переход происходит если объект находится в исходном по отношению к нему состоянии, в каком–либо из подсостояний возникла исключительная ситуация указанного типа и выполняется условие перехода.

4. Переходы по вызову метода, это переходы для которых не указан стереотип и в поле событие–триггер указано имя какого–либо метода данного объекта. Такой переход происходит если объект находится в исходном по отношению к нему состоянии, выполняется сторожевое условие и на данном объекте вызван указанный метод.

5. Переходы по времени, это переходы, для которых не указан стереотип и в поле событие–триггер указано служебное слово after. Аргументом события–триггера должно быть выражение целого типа. Такой переход срабатывает, если объект находится в исходном по отношению к переходу состоянии, выполняется условие и прошло указанное в аргументе число миллисекунд после того, как исходное состояние стало активным.

Теперь, после того как мы описали возможные переходы между состояниями — сделаем несколько дополнительных замечаний, касающихся формального определения работы машины состояний в "Zebra".

• Выполнение перехода является атомарной операцией в работе машины состояний. Поиск перехода, который должен быть выполнен в следующий момент происходит от текущего состоя-

ния к состоянию самого верхнего (имеется в виду вложенные состояния) уровня. При этом возможность выполнения внутренних переходов проверяется в первую очередь.

• Стабильным — называется состояние из которого невозможно выйти без внешнего по отношению к объекту события (т.е. получения объектом сигнала, вызова на объекте функции или истечения некоторого промежутка времени).

• Run–To–Completion Step (RTCS) — последовательность простых и по исключительной ситуации переходов начинающаяся со стабильного состояния и заканчивающаяся тоже стабильным состоянием. Эта последовательность всегда выполняется синхронно, то есть невозможно выполнение в нескольких потоках одновременно различных RTCS для одного и того же объекта. Началом RTSC всегда служит или внешний сигнал или вызов метода или окончание некоторого промежутка времени.

• В случае, если RTCS вызвано внешним сигналом, то в зависимости от условий вызова вся последовательность может выполняться как в вызывающем потоке, так и в дополнительном, созданном "Zebra".

• RTCS, инициированный вызовом функции, ВСЕГДА выполняется в ее потоке. Управление в нее вернется только после полного завершения всех переходов в последовательности RTCS.

• "Zebra" не специфицирует порядок проверки условий переходов, поэтому нежелательно, чтобы процесс проверки условия вызывал побочные эффекты, могущие повлиять на проверку других сторожевых условий. В противном случае поведение машины состояний становится непредсказуемым.

5. Стереотипы. Для классов "Zebra" зарезервированы следующие стереотипы:

• zbasic. Класс, обладающий поведением, но не предназначенный для непосредственного управления некоторым компонентом пользовательского интерфейса. На UML–диаграмме это просто класс, к которому может быть привязана диаграмма состояний. Генератор создаст соответствующий ему java–класс, обладающий поведением, описанным диаграммой.

• zform. Класс, обладающий поведением, и предназначенный для непосредственного управления некоторым компонентом поль-

зовательского интерфейса. На UML–диаграмме это класс, к которому может быть привязана диаграмма состояний и который ассоциирован с классом со стереотипом zform bean. Генератор создаст соответствующий ему java–класс, обладающий поведением, описанным диаграммой и некоторым предопределенным набором операций, необходимых для связи с презентацией.

• zform bean. Этот класс не обладает поведением. Он предназначен только для передачи данных и событий между zform и презентацией и с точки зрения разработчика UI logic layer является абстракцией некоторого компонента UI presentation layer. Более детально использование стереотипов zform и zform bean будет описано в части, посвященной компонентам времени исполнения "Zebra".

• exception. Эти классы используются "Zebra" для генерации java–исключений. Сгенерированный класс будет автоматически унаследован от java–класса java.lang.Exception, если в модели явно не указано иное.

• signal. Эти классы используются для генерации классов сигналов. Сгенерированный класс будет автоматически унаследован от java–класса java.util.EventObject, если в модели явно не указано иное.

• Interface. Эти классы используются для генерации java–интерфейсов.

6. Поддержка многопоточности. Спецификация многопоточности в UML–модели является практически полностью расширением спецификации UML, специфичным для "Zebra". Вызвано это тем, что в существующей версии UML 1.3 нет удовлетворительных и достаточно богатых для реальной разработки средств специфицирования многопоточной обработки нижнего уровня.

"Zebra" предлагает свой гибкий механизм специфицирования многопоточности. Он позволяет разработчику выбрать и реализовать наиболее приемлемый для приложения механизм управления потоками без существенного изменения модели (что позволяет попробовать на приложении несколько вариантов без существенных затрат на переписывание), и, в то же время, позволяет на первых этапах разработки приложения (например, при созда-

нии прототипа) использовать предопределенные механизмы и не тратить время на описание не слишком важных на данном этапе подробностей.

Механизм основан на использовании менеджеров потоков — специальных объектов, управляющих выделением потоков под различные операции — которые могут быть сопоставлены приложению (один менеджер потоков на все объекты приложения), классу или множеству классов (несколько менеджеров потоков на приложение) или даже множеству объектов (в том числе и отдельному объекту).

§ 6. Компоненты времени исполнения

Зная только язык "Zebra" уже возможно создавать java–приложения или их части, но наиболее эффективным использование "Zebra" становится при использовании компонент времени исполнения — нескольких компонент, уже реализующих большой блок функциональности, часто встречающейся в реальных приложениях.

Самыми важными компонентами времени исполнения входящими в состав "Zebra" являются Zebra–сервер, наиболее часто используемые презентации (Swing, JSP, XML) и подсистема безопасности.

1. **Zebra–сервер**, или ZServer, занимает центральное место в архитектуре "Zebra". Zebra–сервером реализуются такие функции как управление сессиями пользователя, управление презентациями, управление потоками по умолчанию и некоторые другие менее важные. Подсистема безопасности также во многом основана на интерфейсах, предоставляемых Zebra–сервером.

2. **Swing–презентация** является полноценной презентацией "Zebra", т.е. может быть использована с любым приложением "Zebra". Создавалась Swing–презентация скорее для целей отладки.

Swing–презентация ставит в соответствие каждому объекту zform bean графическое окно Java библиотеки Swing, в котором можно видеть и изменять значения атрибутов объекта zform bean. Это позволяет отлаживать любое приложение (в том числе и web–приложение) как консольное, исполняющееся на одном

компьютере. Подразумевается, что после отладки swing–презентация заменяется на JSP– или XML–презентацию, и, после окончательного тестирования, устанавливается на web–сервер.

В то же время, Swing–презентация позволяет построить самостоятельное законченное java–приложение. Формы такого приложения могут быть созданы как автоматически (по умолчанию), без дополнительного программирования, так и при помощи специального средства Form Editor, входящего в состав Swing–презентации. В последнем случае, пользовательский интерфейс может быть более насыщенным и сложным. Имеется возможность построения пользовательского интерфейса и полностью "вручную", ручным кодированием на Java с использованием готовых компонент, входящих в состав Swing–презентации.

3. JSP–презентация представляет собой web–приложение на базе JSP и servlet технологий Java. Каждому объекту zform bean JSP–презентация сопоставляет JSP–web–страницу. Таким образом, при подключении JSP–презентации Zebra–приложение становится web–приложением, способным выполняться на любом servlet и JSP совместимом web–сервере.

В функции JSP–презентации входит разбор HTTP запросов и вызов соответствующих методов на Zebra–сервере и объектах zform bean, сопоставление объектам zform bean соответствующих им JSP–страниц, управление HTTP сессиями и сопоставление их Zebra–сессиям, сопоставление сущностей системы безопасности HTTP и servlet/JST сущностям системы безопасности "Zebra", и некоторые другие.

Кроме того, в состав JSP–презентации входит генератор шаблонов JSP–страниц по UML–модели. Поскольку каждому классу zform bean необходимо сопоставить свою JSP–страницу, причем во многих страницах повторяется один и тот же код, был создан инструмент, позволяющий сгенерировать по имеющимся в модели классам zform bean–шаблоны JSP–страниц. Эти шаблоны можно как немедленно использовать при отладке приложения, так и передать дизайнерам для оформления. Затем, отладочные шаблоны просто заменяются на шаблоны, полученные от дизайнеров, и итоговая версия приложения готова к итоговому тестированию и работе.

4. XML–презентация или SOAP–презентация очень похожа на JSP–презентацию как внешне, так и по внутренней структуре. Она также представляет собой servlet/JSP–приложение, предоставляющее HTTP доступ к приложению "Zebra", и выполняет все те же действия, что и JSP–презентация. Но формат представления объектов zform bean иной: вместо HTML используется SOAP. Эта презентация может быть использована, когда приложение предназначено для использования не конечными пользователями, а другими приложениями, как сервиса.

Надо сказать, что первоначально, при разработке, "Zebra" ориентировалась именно на разработку web–приложений, поэтому на данный момент JSP– и XML–презентации являются наиболее развитыми и часто используемыми.

5. JMS–презентация позволяет разработчику Zebra–приложения разделить бизнес логику и логику пользовательского интерфейса так, чтобы их функционирование осуществлялось в разных Java виртуальных машинах (возможно и на физически разных компьютерах).

JMS–презентация состоит из двух взаимодействующих сторон — сервера и клиента. Сервер выполняет всю работу заданную UML–моделью. Он содержит в себе ZServer с присоединенной JMS–презентацией. Клиент — может одновременно взаимодействовать и с какой либо другой презентацией в той же виртуальной машине и с сервером, пересылая последнему все запросы пользователя и получая от него ответы в виде JMS–сообщений. Клиент полностью реализует интерфейс Zebra–сервера, именно с ним и работает реальная (например, Swing или JSP) презентация.

6. Подсистема безопасности "Zebra" позволяет аутентифицировать пользователей и авторизовать их действия. Наиболее важные ее особенности приведены ниже:

• Основана на стандартной Java2 подсистеме безопасности и расширении JAAS.

• Поддержка подключаемых модулей аутентификации (Pluggable Authentication Module — PAM).

• Поддержка single sign–on, что значит, что пользователь может авторизоваться в системе единственный раз и затем пользо-

ваться своими правами доступа в течение всего сеанса работы с системой.

• Гибкая политика контроля авторизации основанной на пользователях, группах или ролях, в зависимости от нужд конкретного приложения.

• Распределение прав доступа вплоть до атрибутов и действий объектов zform bean или действий zform. Так, например, для какого–то атрибута можно указать, что администратор системы может его и читать и менять, а обычные пользователи только читать. После этого на одной и той же странице администратор сможет изменить атрибут (будет видеть поле ввода), пользователь же только прочитать его (увидит только текст).

• В комплекте поставки имеется web–приложение, демонстрирующее все возможности системы безопасности "Zebra".

З а к л ю ч е н и е

Технология "Zebra" уже успешно использовалась в нескольких программных проектах и показала неплохие результаты. В дальнейшем предполагается продолжать развитие технологии в следующих направлениях:

1) автоматизация других этапов процесса разработки либо интеграция с системами и/или технологиями, их автоматизирующими. Так, возможно включение в разрабатываемую систему, упрощающую бизнес-анализ путем полуавтоматической обработки текстов на естественном языке, описывающих предметную область и требования заказчика к системе;

2) введение новых компонент, упрощающих разработку приложений отдельных типов (в первую очередь web–приложений). Планируется большая ориентация технологии на разработку именно web–приложений, для чего будут введены новые или оптимизированы существующие компоненты времени исполнения, появятся новые стереотипы классов, ориентированные на реализацию web–приложений.

В более далеких планах развитие

3) возможностей прототипирования, для чего должна быть реализована возможность генерации прототипов не только по

достаточно детальным диаграммам классов, но и по менее детализированным use–case диаграммам и диаграммам взаимодействий,

4) поддержки других языков, кроме Java.

Литература

1. BOOCH G. Object-Oriented Analysis and Design with Applications, 2nd edition. ISBN The Benamin Cummings. – Redwood City. — 1994.

2. JACJBSON I. The Unified Software Development Process, ISBN 0201571692, Addison–Wesley, 1999.

3. BOOCH G. and others, The Unified Modeling Language User Guide/ G.Booch , J.Rumbaugh, I.Jacobson, J.Rumbaugh. – Addison–Wesley, 1998.

4. RUMBAUGH J. Object-Oriented Modeling and Design, Prentice Hall. – 1991.

5. JACOBSON I. and others, Software Reuse: Architecture Process and Organization for Business Success/ I.Jacobson, M.Griss, P.Jonsson. – Addison–Wesley: ACM Press, 1997.

6. JACOBSON I., BYLUND S. The Road to the Unified Software Development Process. – Cabridge: Cambridge University Press, 2000.

7. PAGE–JONES M. Fundamentals of Object–Oriented Design in Uml (Addison Wesley Object Technology Series). – Addison–Wesley. — 1999.

8. ИВАНОВА Г.С. и др. Объектно–ориентированное программирование: Учебник (под ред. Ивановой Г.С.) / Иванова Г.С., Ничушкина Т.Н., Пугачев Е.К. – М: МГТУ им. Н.Э.Баумана, 2001. – 320 с.

9. ФАУЛЕР М., СКОТТ К. UML: Основы: Краткое руководство по унифицированному языку моделирования: Пер. с англ. Изд. 2–е. – С.-Петербург: Символ–Плюс, 2002. – 192 с.

10. КВАТРАНИ Т. Rational Rose 2000 и UML: Визуальное моделирование: Пер. с англ.) Объектно–ориентированные технологии в программировании — М: ДМК Пресс, 2000. – 176 с.

11. БУЧ Г. и др. UML: Специальный справочник: Пер. с англ./ Рамбо Дж., Якобсон А., Буч Г. – С.-Петербург: Питер, 2002. – 656 с.

Поступила в редакцию
5 июня 2002 года