**Статья опубликована в ”Научно-техническом вестнике. Вып. 30. Фундаментальные и прикладные исследования информационных систем и технологий”. СПбГУ ИТМО. 2006, с. 32–44.**

# UNIMOD – ИНСТРУМЕНТАЛЬНОЕ СРЕДСТВО ДЛЯ АВТОМАТНОГО ПРОГРАММИРОВАНИЯ

**В. С. Гуров, М. А. Мазин, А. А. Шалыто**

*Санкт-Петербургский государственный институт информационных технологий, механики и оптики*

*В статье предлагается метод проектирования и реализации реактивных объектно-ориентированных программ с явным выделением состояний. Метод основан на использовании SWITCH-технологии и UML-нотации. Описывается инструментальное средство с открытым исходным кодом UniMod, являющееся встраиваемым модулем для платформы Eclipse, которое предназначенное для поддержки этого метода.*

## Введение

В последнее время идея *запускаемого UML* [1, 2] приобретает все большую популярность. Это связано с тем, что практическое использование *Unified Modeling Language* (*UML)* [3] в большинстве случаев ограничивается моделированием статической части программы с помощью диаграммы классов. Моделирование динамических аспектов программы на языке *UML* затруднено в связи с отсутствием в стандарте формального однозначного описания правил интерпретации поведенческих диаграмм. Также следует отметить, что связь поведенческих диаграмм с кодом на целевом языке программирования в современных широко известных средствах моделирования, например *IBM Rational Rose*, реализована слабо, либо вообще не реализована.

Ивар Якобсон, один из создателей языка моделирования *UML*, в докладе «Четыре основные тенденции в разработке программного обеспечения (ПО)» [4] перечислил важнейшие, по его мнению, направления развития процесса разработки ПО.

Он отметил, что технологическая база разработки объектно-ориентированного ПО, состоящая из языка *UML* и стандартного процесса разработки *Rational Unified Process* (*RUP*) [5], приобрела устойчивое состояние. По его мнению, следующим шагом должно стать их широкое внедрение. Уже сегодня можно говорить что «процесс пошел»: И. Якобсон утверждает, что язык *UML* преподается более чем в 1000 университетах мира, а, в присутствии авторов, представители фирмы *IBM* в одном из своих докладов в течение четырех часов демонстрировали использование *UML* и *RUP*.

И. Якобсон обозначил следующие тенденции разработки ПО.

1. Аспектно-ориентированное программирование [6]. Аспекты упрощают добавление в систему сквозной функциональности, такой, например, как логирование или проверка прав доступа. Отметим, что И. Якобсон отождествляет понятие аспекта с вариантом использования, что позволяет моделировать аспекты с помощью языка *UML*.
2. Исполняемый *UML*. В настоящее время *UML* применяется, в основном, как язык спецификации моделей систем. Существующие *UML*-средства позволяют строить различные диаграммы и автоматически создавать по диаграмме классов «скелет» кода на целевом языке программирования (языки *Java* и *C#*). Некоторые их этих

средств также предоставляют возможность автоматически генерировать код логики программы по диаграммам состояний.

Однако можно считать, что в настоящее время указанная функциональность существует в «зачаточном состоянии», так как известные инструменты не позволяют в полной мере эффективно связывать модель поведения, которую можно описывать с помощью четырех типов диаграмм (состояний, деятельностей, кооперации или последовательностей), с генерируемым кодом. Это во многом определяется отсутствием в языке *UML* формального однозначного описания операционной семантики для поведенческих диаграмм.

Отсутствие однозначной операционной семантики при традиционном написании программ приводит к дублированию описания поведения, как в модели, так и на целевом языке, а также к произвольной интерпретации поведенческих диаграмм программистом. Более того, описание поведения в модели часто носит неформальный характер. Появление операционной семантики зафиксирует однозначность понимания диаграмм и приведет к появлению исполняемого *UML*, для которого код, в привычном смысле этого слова, может не генерироваться вообще.

3. Процесс разработки ПО должен быть активным. Существующие средства разработки требуют длительного времени для их изучения. И. Якобсон считает, что средства разработки должны предсказывать действия разработчика и предлагать варианты решения возникших проблем в зависимости от текущего контекста. Отметим, что подобный подход реализован во многих современных средах разработки (например, *Borland JBuilder*, *Eclipse*, *IntelliJ IDEA*) для текстовых языков программирования, но не для языка *UML*.
4. Разработанное ПО также должно быть активным, однако, не для разработчика, а для конечного пользователя.

По мнению авторов, наиболее интересными и востребованными на сегодняшний день тенденциями являются вторая и третья.

Признание многими ведущими в области разработки ПО фирмами того факта, что программы необходимо не писать «на авось» (как сказал по-русски на конференции «Microsoft Research Academic Days in St.-Petersburg, April 21-23, 2004» создатель языка *Eiffel* Бертран Мейер), а проектировать, привело к появлению такого направления в программной инженерии как «проектирование на базе моделей» (Model-Driven Design) [7–9].

## 1. Проектирование на базе моделей

Основной идеей проектирования на базе моделей является абстрагирование от целевого языка программирования и целевой программно-аппаратной платформы — уход на более высокий уровень абстракции. Такой подход позволяет инженеру сосредоточиться на предметной области разрабатываемой системы, не вникая и не описывая детали реализации на стадии моделирования. При этом очень важно выбрать правильный язык и средство моделирования. Выбор должен производиться, основываясь на предметной области разрабатываемой системы. Так, например, если необходимо разработать компилятор, то в качестве языка моделирования логично выбрать язык формальных грамматик, а в качестве инструмента – один из существующих пакетов для разработки компиляторов [10].

### 1.1. Реактивные системы

Другим, достаточно широким, классом программных систем является класс реактивных систем – систем, выполняющих определенные действия в ответ на внешние события. В работе Д. Харела [11] отмечено, что для моделирования таких систем хорошо подходит

расширение конечных автоматов за счет применения вложенных состояний. Для построения таких автоматов и генерации кода по ним созданы инструментальные средства, многие из которых перечислены в [12], в которой, в частности, упомянуты инструменты как *I-Logix Statemate* (http://ilogix.com/sublevel.aspx?id=74), *XJTek AnyState* (http://www.xjtek.com/anystates/), *StateSoft ViewControl* (http://www.statesoft.ie/products.html), *SCOPE* (http://www.itu.dk/~wasowski/projects/scope/), *IAR Systems visualSTATE* (http://www.iar.com/p1014/p1014_eng.php), *The State Machine Compiler* (http://smc.sourceforge.net/).

Недостатком этих инструментов является то, что они позволяют строить и реализовывать только поведенческую часть модели программы, не рассматривая ее структуру (статику). Поэтому с помощью этих инструменты программу в целом не построить.

### 1.2. ЗАПУСКАЕМЫЙ *UML*

Дальнейшее развитие подход Харела получил в рамках языка *UML* в виде диаграмм состояний (*statechart*). В *UML,* помимо этого типа диаграмм, входит еще ряд других. Все типы диаграмм разделены на две группы: структурные и поведенческие. Для поведенческих диаграмм в стандарте *UML* неформально описана их операционная семантика, что привело к появлению таких понятий как *запускаемый UML* [1, 2] и *виртуальная машина UML* [13]. Появились также инструментальные средства [14, 15] для их реализации.

В [14] модель программной системы предлагается строить следующим образом: структура моделируется с помощью *UML*-диаграммы классов, а поведение – с помощью описания каждого метода каждого класса в виде *UML*-диаграммы последовательности вызовов (sequence). Такой подход крайне неудобен, так как приводит к очень громоздким моделям.

В [15] предлагается расширить *UML* текстовым платформенно-независимым императивным языком для описаний действий, что приводит к перегрузке графических диаграмм текстовой информацией.

В рамках проекта [16] планируется создать инструмент для преобразования *UML*-диаграмм состояний в код на языке *Java*.

Отметим, что некоторые инструменты для изображения *UML*-диаграмм, такие как [17], не ассоциирующие себя с описанными выше понятиями, также в некоторой степени их реализуют.

Далее в статье, на примере разработанного авторами проекта с открытым исходным кодом *UniMod* (http://unimod.sourceforge.net), описаны применение *UML*-нотации для создания графического языка описания систем со сложным поведением и инструмент моделирования для этого языка, который поддерживает активный процесс разработки ПО.

Проект *UniMod* обладает концептуальной целостностью и позволяет строить компактные модели программных систем в целом, а не только их поведенческой части.

## 2. ИСПОЛНЯЕМЫЙ ГРАФИЧЕСКИЙ ЯЗЫК НА ОСНОВЕ *SWITCH*-ТЕХНОЛОГИИ И *UML*-НОТАЦИИ

В работе [18] предложен метод проектирования и реализации программ с явным выделением состояний, названный «*SWITCH*-технологией» или «автоматным программированием». Особенность этого подхода состоит в том, что программы предлагается строить так же, как осуществляется автоматизация технологических процессов, в ходе которой первоначально строится схема связей, содержащая источники информации, систему управления и объекты управления, а также обратные связи. В предлагаемом подходе систему управления реализуется в виде системы взаимосвязанных структурных конечных автоматов, каждый из которых имеет несколько входов и выходов. Поведение автоматов описывается графами переходов с нотацией, предложенной в работе [19].

Метод проектирования и реализации реактивных объектно-ориентированных систем с явным выделением состояний предложен в работе [20].

*SWITCH*-технология определяет для каждого автомата два типа диаграмм (схему связей и граф переходов) и их операционную семантику. При наличии нескольких автоматов, также строится схема их взаимодействия. *SWITCH*-технология задает свою нотацию диаграмм. Предлагается, сохранив автоматный подход, использовать *UML*-нотацию при построении диаграмм в рамках *SWITCH*-технологии. При этом, используя нотацию диаграмм классов языка *UML*, строятся схемы связей автоматов, определяющих их интерфейс, а графы переходов строятся с помощью нотации диаграммы состояний *UML*.

Предлагаемый процесс моделирования системы состоит в следующем:

- на основе анализа предметной области разрабатывается концептуальная модель системы, определяющая сущности и отношения между ними;
- в отличие от традиционных для объектно-ориентированного программирования подходов [21], из числа сущностей выделяются источники событий, объекты управления и автоматы. Источники событий активны – они по собственной инициативе воздействуют на автоматы. Объекты управления пассивны — они выполняют действия по командам от автоматов. Объекты управления также формируют значения входных переменных для автоматов. Каждый автомат активируется источниками событий и на основании значений входных переменных и текущего состояния воздействует на объекты управления, переходя в новое состояние;
- используя нотацию диаграммы классов, строится схема связей автомата, задающая его интерфейс. На этой схеме слева отображаются источники событий, в центре – автоматы, а справа – объекты управления. Источники событий с помощью *UML*-ассоциаций связываются с автоматами, события которым они поставляют. Автоматы связываются с объектами, которыми они управляют;
- схема связей, кроме задания интерфейса автомата, выполняет функцию, характерную для диаграммы классов – задает объектно-ориентированную структуру программы;
- каждый объект управления содержит два типа методов, реализующих входные переменные (`xj`) и выходные воздействия (`zk`);
- для каждого автомата с помощью нотации диаграммы состояний строится граф переходов типа *Мура-Мили*, в котором дуги могут быть помечены событием (`ei`), булевой формулой из входных переменных и формируемыми на переходах выходными воздействиями. В вершинах могут указываться выходные воздействия и имена вложенных автоматов. Каждый автомат имеет одно начальное и произвольное количество конечных состояний;
- состояния на графе переходов могут быть простыми и сложными. Если в состояние вложено другое состояние, то оно называется сложным. В противном случае состояние простое. Основной особенностью сложных состояний является то, что наличие дуги, исходящей из такого состояния, заменяет однотипные дуги из каждого вложенного состояния;
- все сложные состояния неустойчивы, а все простые, за исключением начального – устойчивы. При наличии сложных состояний в автомате появление события может привести к выполнению более одного перехода. Это происходит в связи с тем, что сложное состояние является неустойчивым и автомат выполняет переходы до тех пор, пока не достигнет первого из простых (устойчивых) состояний. Отметим, что если в графе переходов сложные состояния отсутствуют, то, как и в *SWITCH*-технологии, при каждом запуске автомата выполняется не более одного перехода;
- каждая входная переменная и каждое выходное воздействие являются методами соответствующего объекта управления, которые реализуются вручную на целевом языке программирования;

- использование символьных обозначений в графах переходов позволяет весьма компактно описывать сложное поведение проектируемых систем. Смысл таких символов задает схема связей. При наведении курсора на соответствующий символ на графе переходов во всплывающей подсказке отображается его текстовое описание.

На рис. 1 приведена схема связей автомата, а на рис. 2 – его граф переходов, построенные в *UML*-нотации описанным выше способом.

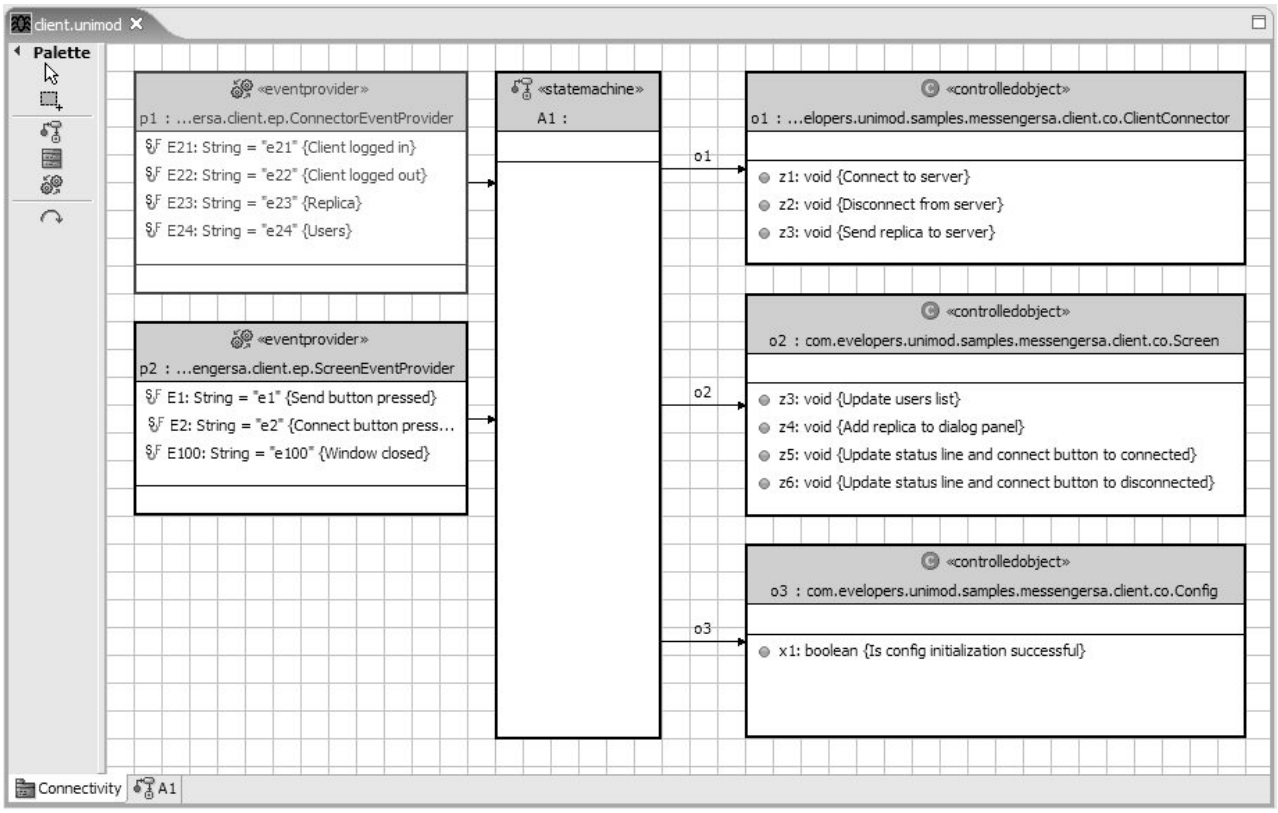


Рис. 1. Пример схемы связей автомата

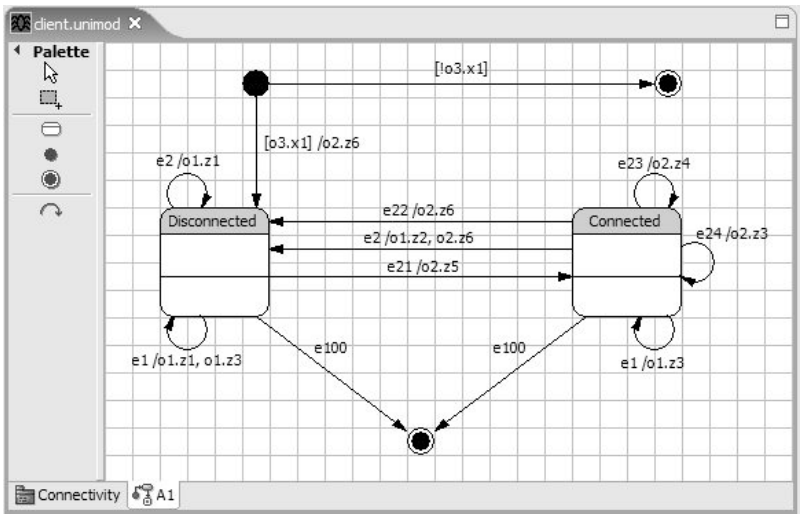


Рис. 2. Пример графа переходов автомата

Зададим операционную семантику модели системы, построенной описанным выше образом:

- при запуске модели, инициализируются все источники событий и объекты управления. После этого источники событий начинают воздействовать на связанные с ними автоматы;
- каждый автомат начинает свою работу из начального состояния, а заканчивает – в одном из конечных;
- при получении события, автомат выбирает все исходящие из текущего состояния переходы, помеченные символом этого события;
- автомат перебирает выбранные переходы и вычисляет булевы формулы, записанные на них, до тех пор, пока не найдет формулу со значением *true*;
- если переход с такой формулой найден, автомат выполняет выходные воздействия, записанные на дуге, и переходит в новое состояние. В нем автомат выполняет выходные воздействия, а также запускает вложенные автоматы. Если новое состояние оказалось составным, то выполняется переход из начального состояния, находящегося внутри данного составного состояния;
- если переход не найден, то автомат продолжает поиск перехода у родительского состояния – состояния, в которое вложено текущее состояние;
- при переходе в конечное состояние автомат останавливает все источники событий. После этого работа системы завершается.

Более подробное описание операционной семантики приведено в работе [22].

Описав исполняемый графический язык на основе *UML*-нотации и его операционную семантику, перейдем к описанию процесса создания инструмента моделирования, который будет поддерживать активный процесс разработки программ на его основе.

## 3. *UNIMOD* — ИНСТРУМЕНТАЛЬНОЕ СРЕДСТВО ДЛЯ АВТОМАТНО-ОРИЕНТИРОВАННОГО ПРОГРАМИРОВАНИЯ

Инструментальное средство *UniMod* обеспечивает разработку и выполнение автоматно-ориентированных программ. Он позволяет создавать и редактировать *UML*-диаграммы классов и состояний, которые соответствуют схеме связей и графу переходов.

Проектирование программ с использованием инструментального средства *UniMod* предполагает следующий подход: логика приложения описывается структурным конечным автоматом, заданным в виде набора указанных выше диаграмм, построенных с использованием *UML*-нотации. Источники событий и объекты управления реализуются вручную на целевом языке программирования.

Пакет *UniMod* поддерживает два типа реализации построенных диаграмм – на основе интерпретации и компиляции.

Интерпретатор является *виртуальной машиной UML.*

На рис. 3 приведена структурная схема для интерпретационного подхода.

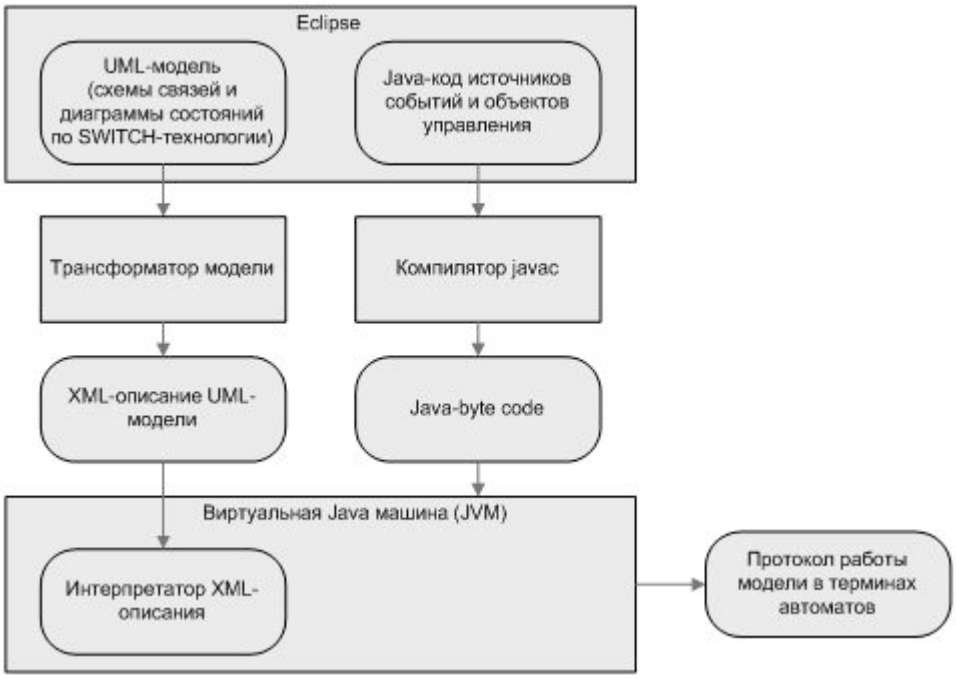


Рис. 3. Структурная схема интерпретационного подхода

Как следует из приведенного схемы, при использовании интерпретационного подхода, **исходным кодом являются *UML*-модель** (схемы связей и диаграммы состояний по *SWITCH*-технологии) **и *Java*-код источников событий и объектов управления**.

При запуске программы интерпретатор загружает в оперативную память *XML*-описание и создает экземпляры источников событий и объектов управления. В процессе работы указанные источники формируют события и направляют интерпретатору, который обрабатывает их в соответствии с логикой, описываемой автоматами. При этом он вызывает методы объектов управления, реализующие входные переменные и выходные воздействия.

На рис. 4. приведена структурная схема для компилятивного подхода.

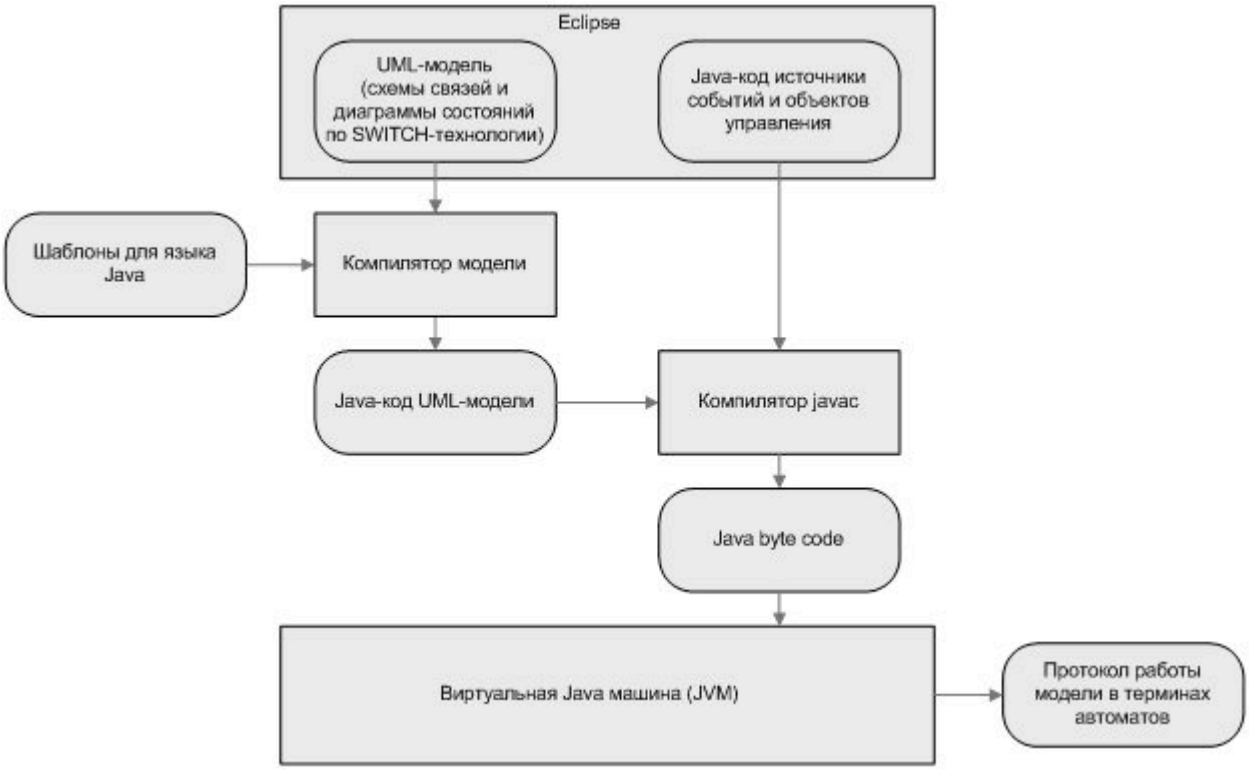


Рис. 4. Структурная схема компилятивного подхода

При использовании компилятивного подхода *UML*-модель непосредственно преобразуется в код на целевом языке программирования, который впоследствии компилируется и запускается.

Компилятивный подход целесообразно применять для устройств с ограниченными ресурсами, например для мобильных телефонов. Такой подход является типичным для «классической» *SWITCH*-технологии.

Можно утверждать, что рассмотренное инструментальное средство позволяет совмещать декларативный и процедурный подходы для построения программ.

## 4. РЕАЛИЗАЦИЯ РЕДАКТОРА ДИАГРАММ НА ПЛАТФОРМЕ *ECLIPSE*

Инструмент для создания указанных диаграмм является встраиваемым модулем (*plug-in*) для платформы *Eclipse* (http://www.eclipse.org). Эта платформа обладает рядом преимуществ перед такими продуктами, как, например, *IntelliJ IDEA* или *Borland JBuilder:*

- является бесплатным продуктом с открытым исходным кодом;
- содержит библиотеку для разработки графических редакторов – *Graphical Editing Framework*;
- активно развивается фирмой *IBM* и уже сейчас обладает не меньшей функциональностью, чем упомянутые выше аналоги.

Для обеспечения процесса активной разработки программ на текстовых языках в перечисленных выше средствах разработки реализованы:

- подсветка семантических и синтаксических ошибок;
- автоматическое завершение ввода и автоматическое исправление ошибок;
- форматирование и рефакторинг [23] кода;
- запуск и отладка программы внутри среды разработки.

В английском языке эти возможности называются "*code assist*". При создании описываемого модуля для платформы *Eclipse* авторы перенесли указанные подходы на процесс редактирования диаграмм.

### *4.1. Валидация модели*

Для текстовых языков программирования редакторы осуществляют проверку принадлежности программы к заданному языку и выделяют (подсвечивают) места в коде, содержащие синтаксические ошибки. К семантическим ошибкам для текстовых языков программирования относится, например, использование необъявленных переменных, вызовы несуществующих методов, некорректное приведение типов.

В стандарте на язык *UML* синтаксис и семантика диаграмм определяется набором ограничений, записанных на языке объектных ограничений (*Object Constraint Language*). Этот набор ограничений должен удовлетворяться для любой правильно построенной диаграммы. Именно на этих ограничениях и основана проверка синтаксиса и семантики диаграмм.

Авторами предлагается расширить множество ограничений следующим образом:

- все состояние на диаграмме состояний должны быть достижимы;
- множество исходящих переходов для любого состояния должно быть полно и непротиворечиво. Это означает, что при обработке любого события не должно быть альтернативных переходов и хотя бы один переход должен выполняться.

Проверка корректности диаграмм происходит следующим образом. В фоновом режиме запускается процесс, который при любом изменении диаграммы, проверяет ее на корректность. При нахождении ошибки некорректный элемент на диаграмме выделяется цветом. На рис. 5 приведен пример диаграммы с недостижимым состоянием.

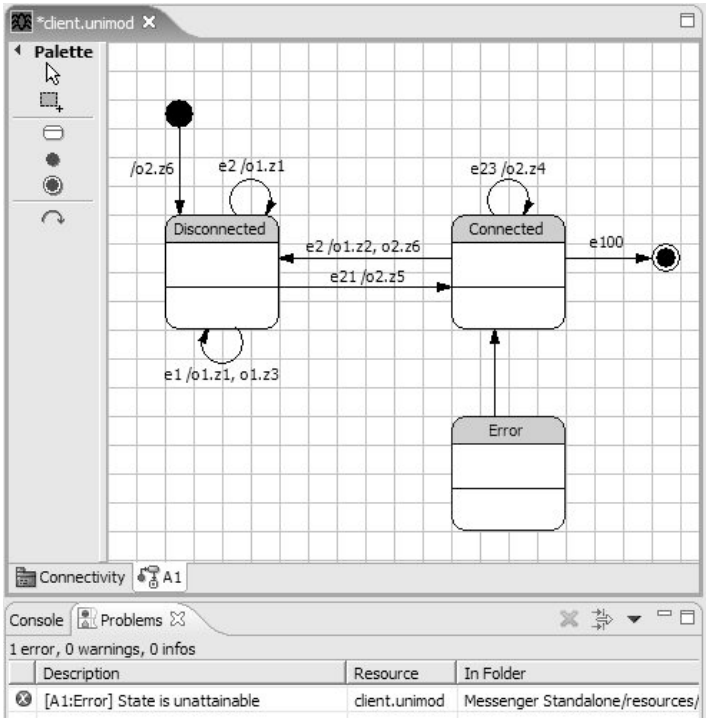


Рис. 5. Недостижимое состояние на графе переходов

### *4.2. Автоматическое завершение ввода и автоматическое исправление ошибок*

Традиционно, автоматическим завершением ввода называется подход, благодаря которому среда по заданному началу лексемы определяет набор допустимых конструкций, префиксом которых данное начало является, и предлагает пользователю выбрать одну из них. Автоматическое исправление ошибок предполагает, что редактор для каждой найденной ошибки указывает пользователю варианты ее исправления.

В случае текстового редактора оба подхода основываются на знании грамматики языка и наборе семантических правил.

В предлагаемом графическом редакторе диаграмм в *UML*-нотации эти подходы реализованы авторами на базе ограничений, определенных стандартом языка *UML* и описанных выше дополнительных ограничений. Так, для недостижимого состояния, представленного на рис. 5, пользователю будет предложено добавить переход в это состояние из любого достижимого (рис. 6).

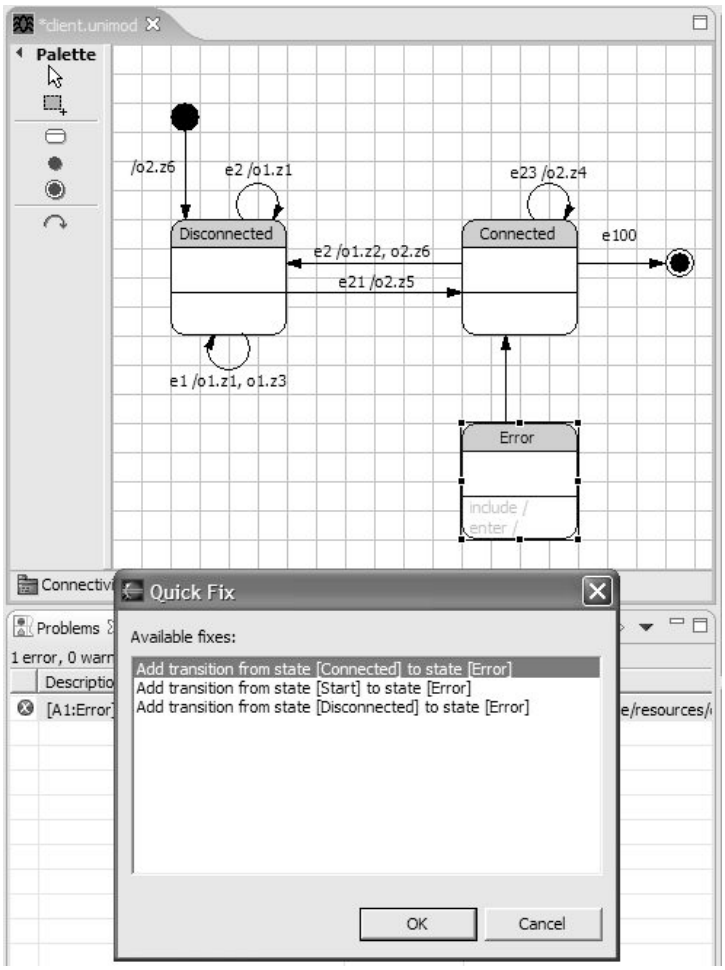


Рис. 6. Предлагаемые варианты исправления ошибки на диаграмме

### *4.3. Форматирование*

Форматирование кода облегчает его чтение. Многие текстовые редакторы позволяют автоматически форматировать код.

Аналогом форматирования кода применительно к диаграммам, по мнению авторов, является их укладка (*layout*). Задача укладки диаграмм является существенно более сложной, чем форматирование кода, так как общепринятые эстетические критерии качества укладки отсутствуют. В проекте *UniMod* раскладка диаграмм осуществляется методом отжига [24].

### *4.4. Запуск модели*

Для запуска программы, написанной на текстовом языке программирования, ее текст либо компилируется в код, исполняемый операционной системой (*C++, Pascal*) или виртуальной машиной (*Java, C#*), либо непосредственно исполняется интерпретатором (*JavaScript*, *Basic*).

Подобные решения доступны и для графического языка программирования. Например, для интерпретационного подхода при запуске диаграммы ее содержимое преобразуется в *XML*-описание, которое передается интерпретатору. Интерпретатор в соответствии с операционной семантикой, изложенной выше, «выполняет» *XML*-описание. Это описание является изоморфным представлением содержимого диаграмм, и поэтому можно говорить о «запуске» диаграмм, как программ.

### *4.5. Отладка модели*

Обычно, после локализации ошибки, отладка программ представляет собой трассировку программного кода оператор за оператором с одновременным анализом значений переменных.

Для графической автоматной модели отладка — это трассировка графа переходов с анализом текущего состояния, событий и значений входных переменных. При необходимости возможна текстовая отладка кода выходных воздействий.

### *4.6. Библиотеки*

Большинство существующих языков программирования поддерживают идеологию библиотек и каркасов (*frameworks*). Библиотека — программный модуль, реализующий функциональность в рамках некоторой предметной области. Каркас — это набор программных базовых сущностей из некоторой предметной области, уточняя и дополняя которые следует строить программу. Для текстовых языков библиотеки и каркасы, как правило, представляют собой скомпилированный код, подключаемый к программе в процессе ее компиляции (статические библиотеки) или во время исполнения (динамические библиотеки).

В рамках автоматного подхода в библиотеки и каркасы следует включать заранее скомпилированные источники событий и объекты управления, так как опыт разработки показывает, что приложения, работающие в одной предметной области, различаются не столько выполняемыми атомарными действиями, сколько логикой выполнения этих действий.

## 5. ПРИМЕНЕНИЕ ПРЕДЛАГАЕМОГО ПОДХОДА ДЛЯ МОБИЛЬНЫХ УСТРОЙСТВ

Примеры использования предлагаемого подхода приведены на сайте http://is.ifmo.ru в разделе “UniMod-проекты”.

Несмотря на то, что язык *Java* может быть использован для создания приложений для мобильных устройств, часто возникает необходимость писать эти приложения на языке *C++*. В этом случае целесообразно применять компилятивный подход. На рис. 7 показано, как изменяется структурная схема этого подхода при переходе на язык *C++* для мобильной платформы *Symbian* (http://www.symbian.com, http://is.ifmo.ru/science/MD-Mobile.pdf).

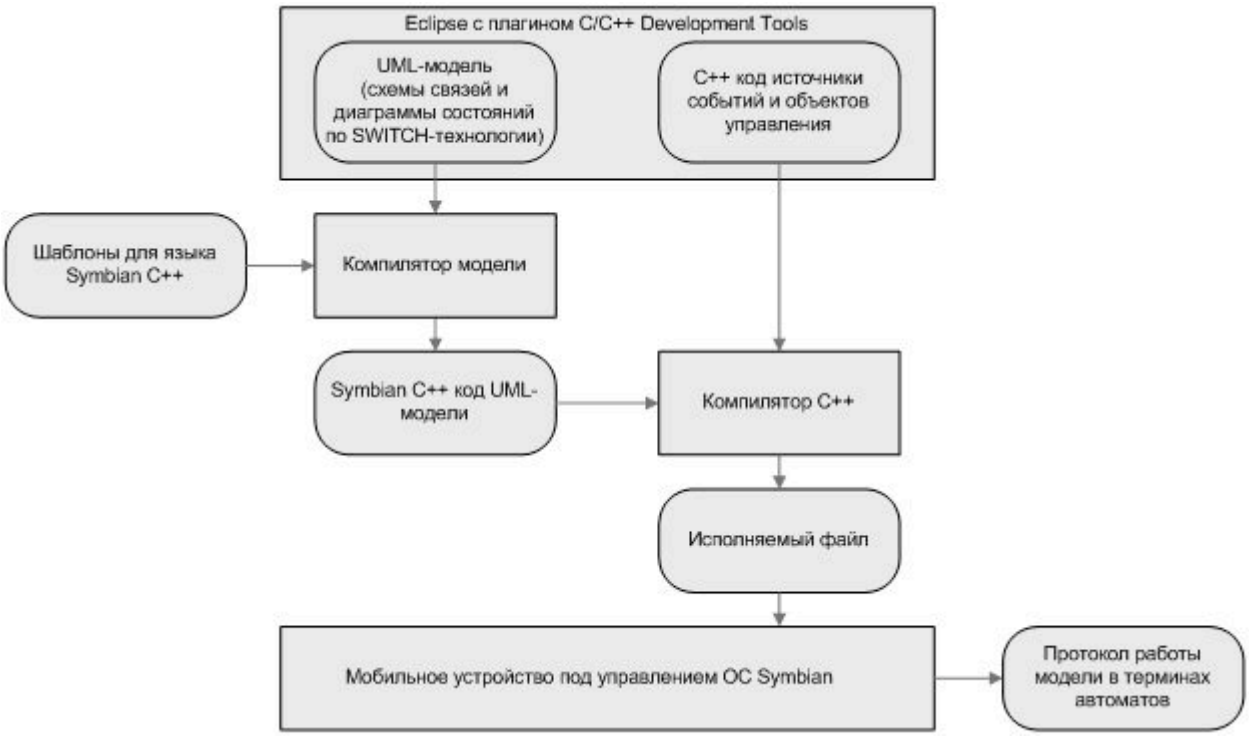


Рис. 7. Структурная схема компилятивного подхода при использовании *Symbian C++*

Таким образом, можно утверждать, что, несмотря на то, что *Eclipse* и *UniMod* ориентированы на язык программирования *Java*, использование шаблонов при компиляции модели делает *UniMod* многоязыковой платформой. Правда, для языков, отличных от *Java*, происходит потеря функциональности, например возможности графической отладки модели.

## ЗАКЛЮЧЕНИЕ

В статье описывается графический язык для поддержки автоматного программирования и инструментальное средство на его основе.

Из изложенного следует, что предлагаемый подход обладает следующими преимуществами по сравнению аналогами, на которые даны ссылки во введении:

- в модели допускается использовать систему взаимосвязанных автоматов, что позволяет декомпозировать поведение сложной задачи на подзадачи. При этом отметим, что каждое состояние также осуществляет декомпозицию подзадачи, выделяя только те входные и выходные воздействия, которые с ним связаны;
- наряду с вложенными состояниями используются также вложенные автоматы, число и глубина вложения которых не ограничены.
- помимо компиляционного подхода, используется также и интерпретационный подход, при котором исходным кодом являются сами диаграммы;
- использование платформы *Eclipse*, языков *Java* и *XML* позволяет безболезненно переносить модели и программу в целом с одной операционной системы на другую (например, с *ОС Windows* на *ОС Linux*, и наоборот.

Предложенный подход позволяет:

- сокращать объем кода на текстовом языке программирования;
- строить предложенные в *SWITCH*-технологии схемы связей и графы переходов в *UML*-нотации диаграмм классов и диаграмм состояний соответственно, и включать их в проектную документацию [25];
- формально и наглядно описывать поведение программ и модифицировать их, изменяя, в большинстве случаев, только графы переходов;

- упростить сопровождение проектов за счет повышения централизации логики программ.

Предлагаемая операционная семантика является детерминированной за счет проверки отсутствия противоречивых переходов, что не выполняется, например, в таких инструментальных средствах, как *Rational Rose* [26] и *Borland Together* [27]. Указанный недостаток также присутствует в инструменте *VisualSTATE* [29], но устраняется с помощью дополнения его исследовательским инструментом *SCOPE* [28].

Исходные тексты, документация и примеры использования инструментальное средство *UniMod* представлены на сайте http://unimod.sourceforge.net. При этом обратим внимание, что такие известные инструментальные средства как *I-Logix Statemate* [30] и *Telelogic TAU G2* [15] являются коммерческими продуктами.

Проекты, выполненные с использованием инструментального средства *UniMod*, приведены http://is.ifmo.ru/unimod-projects/

В заключение отметим, что данная работа базируется на работе [19] и развивает подход, описанный в работах [31, 32].

Отметим также, что в работе [33] сказано, что для языка *UML* выполняется закон «20–80». Данная статья подтверждает этот закон: из всех типов *UML*-диаграмм авторы используют только два типа диаграмм для построения программ в целом. Это соответствует «лезвию Оккама», в соответствии с которым не следует приумножать сущности без необходимости.

Работы выполнена в рамках Федеральной целевой научно-технической программы «Исследования и разработки по приоритетным направлениям науки и техники» на 2002-2006 годы по контракту ИТ 13.4/004/015 «Технология автоматного программирования: применение и инструментальные средства».

Изложенное позволило назвать работу [33] “Исполняемый *UML* из России”.

## ЛИТЕРАТУРА

1. *Mellor S. et al.* Executable UML: A Foundation for Model Driven Architecture. MA: Addison-Wesley, 2002. p. 258.
2. *Raistrick C. et al.* Model Driven Architecture with Executable UML. Cambridge University Press, 2004. p. 412.
3. *Буч Г., Рамбо Г., Якобсон И.* UML. Руководство пользователя. М.: ДМК, 2000. 358 с.
4. *Jacobson I.* Four Macro Trends in Software Development Y2004. http://www.ivarjacobson.com/postnuke/html/modules.php?op=modload&name=UpDownload&file=index&req=getit&lid=9
5. *Якобсон И., Буч Г., Рамбо Дж.* Унифицированный процесс разработки программного обеспечения. СПб.: Питер, 2002. 458 с.
6. *Kiczales G., Lamping J., Mendhekar A. et al.* Aspect-oriented programming / In ECOOP'97 – Object-Oriented Programming. 11th European Conference. 1997. LNCS 1241. pp. 220–242. http://citeseer.ist.psu.edu/kiczales97aspectoriented.html (русский перевод — http://www.javable.com/columns/aop/workshop/02/)
7. *1st European Conference on Model-Driven Software Engineering*. http://www.agedis.de/conference/
8. *International Workshop* “e-Business and Model Based in System Design”. IBM EE/A. SPb.: SPb ETU, 2004.
9. *OMG Model Driven Architecture*. http://www.omg.org/mda/
10. *Создание* компиляторов на *Java*. http://www.kulichki.net/kit/tools/java.html

11. *Harel D.*, Statecharts: A Visual Formalizm for Complex Systems //Science of Computer Programming 8, 1987, pp. 231–274.
12. *Wikipedia.* List of state machine CAD tools. http://en.wikipedia.org/wiki/List_of_state_machine_CAD_tools
13. *Riehle D., Fraleigh S., Bucka-Lassen D., Omorogbe N.* The Architecture of a UML Virtual Machine /Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01). ACM Press, 2001.
14. *Matilda UML Virtual Machine.* http://dssg.cs.umb.edu/projects/umlvm/
15. *Kennedy Carter iUML.* http://www.kc.com/products/iuml/index.html
16. *Jia X. et al*. Using ZOOM Approach to Support MDD. http://se.cs.depaul.edu/ise/zoom/papers/zoom/SERP_ZOOM.pdf
17. *Telelogic TAU G2.* http://telelogic.com/corp/products/tau/g2/index.cfm
18. *Шалыто А.А.* SWITCH-технология. Алгоритмизация программирования задач логического управления. СПб.: Наука, 1998. 628 с. http://is.ifmo.ru/books/switch/1
19. *Шалыто А.А., Туккель Н.И.* SWITCH-технология — автоматный подход к созданию программного обеспечения "реактивных" систем //Программирование. 2001. № 5, с. 45–62. http://is.ifmo.ru/works/switch/1/
20. *Шалыто А.А., Туккель Н.И.* Танки и автоматы //BYTE/Россия. 2003. № 2, с. 69–73. http://is.ifmo.ru/works/tanks_new/.
21. *Грэхем И.* Объектно-ориентированные методы. Принципы и практика. М.: Вильямс, 2004. 768 с.
22. *Гуров В.С., Мазин М.А., Шалыто А.А.* Операционная семантика *UML*-диаграмм состояний в программном пакете *UniMod* /Труды XII Всероссийской научно-методической конференции "Телематика- 2005". СПб.: СПбГУ ИТМО. Т.1, с.74–76. http://tm.ifmo.ru/tm2005/src/224as.pdf
23. *Фаулер М.* Рефакторинг. Улучшение существующего кода. М.: Символ-Плюс, 2003. 623 с.
24. *Fruchterman T. M. J., Reingold E. M.* Graph Drawing by Force Directed Placement. // Software - Practice and Experience. 1991, № 21(11). pp. 1129–1164.
25. *Шалыто А.А.* Новая инициатива в программировании. Движение за открытую проектную документацию //PC Week/RE. 2003. № 40, с. 38–42. http://is.ifmo.ru/works/open_doc/
26. *IBM Rational Rose*. http://www-306.ibm.com/software/awdtools/developer/modeler/
27. *Borland Together.* http://www.borland.com/us/products/together/index.html
28. *SCOPE*. http://www.itu.dk/~wasowski/projects/scope/
29. *IAR Systems visualSTATE.* http://www.iar.com/p1014/p1014_eng.php
30. *I-Logix Statemate.* http://ilogix.com/sublevel.aspx?id=74
31. *Горшкова Е.А., Новиков Б.А.* Использование диаграмм состояний и переходов для моделирования гипертекста //Программирование. 2004. № 1, с. 64–80.
32. *Горшкова Е.А., Новиков Б.А. Белов Д.Д., Гуров В.С., Спиридонов С.В.* Моделирование контроллера Web-приложений с использованием UML //Программирование. 2005, № 1, с. 44–51.
33. *Эккель Б.* Философия Java. СПб.: Питер. 2003. 976 с.

34. *Гуров В.С., Нарвский А.С., Шалыто А.А.* Исполняемый UML из России // PC Week/RE". 2005. № 26, с. 18, 19.