Статья опубликована в журнале «Информационно-управляющие системы», 2004, № 6, с.12-17.

УДК 681.3.06

UML. SWITCH-технология. Eclipse

В. С. Гуров,

ведущий разработчик

М. А. Мазин,

разработчик

А. С. Нарвский,

канд. техн. наук, исполнительный директор

компания eVelopers Corporation

А. А. Шалыто,

д-р техн. наук, профессор

Санкт-Петербургский государственный университет информационных технологий, механики и оптики

В статье описаны метод и процесс моделирования поведения программы с явным выделением состояний, основанные на SWITCH-технологии и UML-нотации. Также описано создание для платформы Eclipse инструмента, поддерживающего этот метод.

This article describes method and process of state-based modeling of program behavior, based on SWITCH-technology and UML notation. Creation of plug-in for Eclipse platform, that supports this method, is also described.

ВВЕДЕНИЕ

В последнее время идея *запускаемого UML* [1] приобретает все большую популярность. Это связано с тем, что практическое использование *Unified Modeling Language* (*UML)* [2], 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)» [3] 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*) [4], 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 [5]. 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. Отметим, что в новой редакции языка *UML* (*UML 2*) такая семантика должна появиться.

   Отсутствие однозначной указанной 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) [6-8].

Основной идеей такого подхода является независимое рассмотрение моделей, создаваемых при проектировании системы, от деталей их реализации на конкретной программно-аппаратной платформе. Проектирование на базе моделей должно привести к появлению универсальных графических языков программирования.

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. При этом отметим, что идея применения *UML* как запускаемого языка для таких систем не нова и описана, например, в работах [9, 10]. Однако подход, предлагаемый авторами, как будет показано ниже, более универсален и лучше формализован.

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

Для создания графического языка диаграммы необходимо наделить операционной семантикой.

В работе [11] предложен метод проектирования событийных объектно-ориентированных программ с явным выделением состояний, названный «*SWITCH*-технологией». Особенность этого подхода состоит в том, что поведение в таких программах описывается с помощью графов переходов структурных конечных автоматов с нотацией, предложенной в работе [12].

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 [13], из числа сущностей выделяются источники событий, объекты управления и автоматы. Источники событий активны — они по собственной инициативе воздействуют на автомат. Объекты управления пассивны — они выполняют действия по команде от автомата. Объекты управления также формируют значения входных переменных для автомата. 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 осуществляет 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.

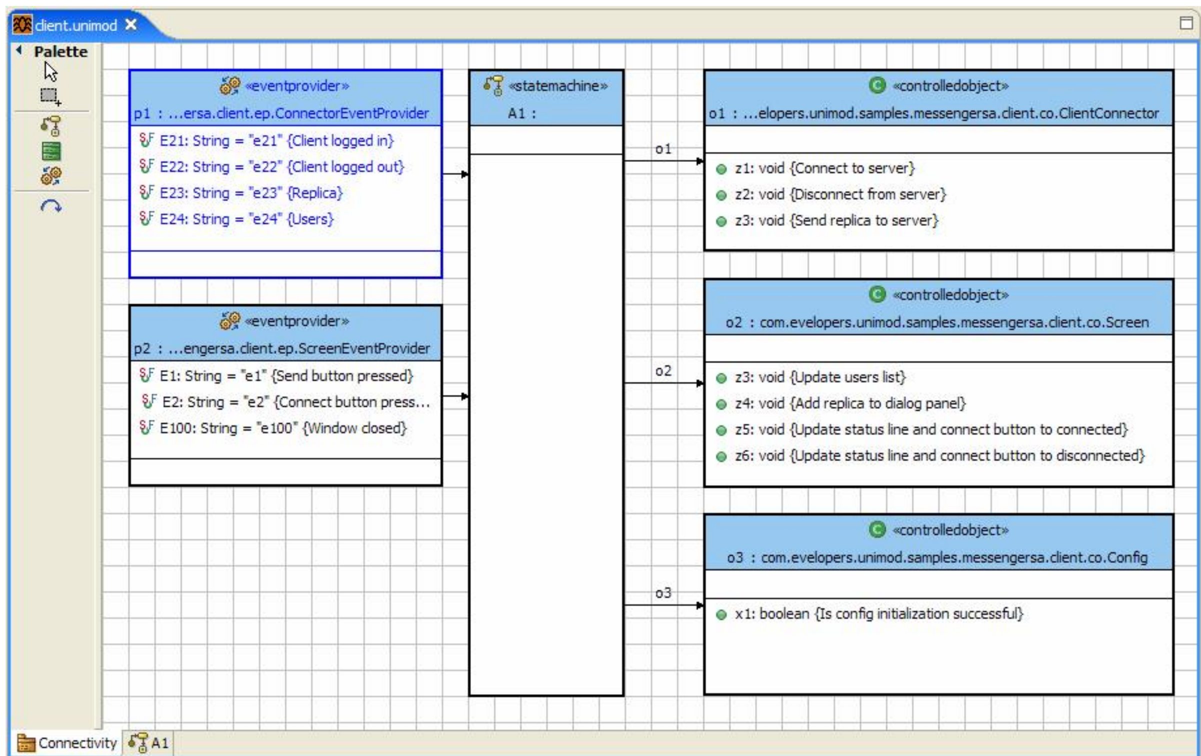

Рис. 1. Пример схемы связей автомата

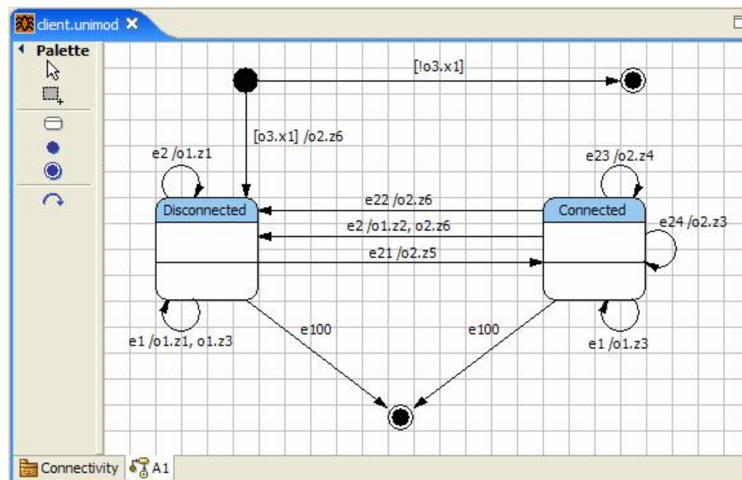

Рис. 2. Пример графа переходов автомата

Зададим операционную семантику для модели системы:

- при запуске модели, инициализируются все источники событий. После этого они 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 переход не найден, то автомат продолжает поиск перехода у состояния, в которое вложено текущее состояние;
- при переходе в конечное состояние автомат останавливает все источники событий. После этого работа системы завершается.

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.

UNIMOD — ПАКЕТ ДЛЯ АВТОМАТНО-ОРИЕНТИРОВАННОГО ПРОГРАМИРОВАНИЯ

Пакет *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, и поддерживает два типа реализации — на основе интерпретации и компиляции. В первом случае имеется возможность:

- преобразовывать диаграммы в формат *XML;*
- исполнять полученное *XML*-описание с помощью интерпретатора, созданного на основе набора разработанных базовых классов. Эти классы реализуют, например, такие функции как обработка событий, сохранение текущего состояния, протоколирование.

Во втором случае диаграммы непосредственно преобразуется в код на целевом языке программирования, который впоследствии компилируется и запускается.

Проектирование программ с использованием пакета *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 задаются кодом на целевом языке программирования.

При использовании языка *Java* источникам событий и объектам управления соответствуют классы. Для этого языка в пакете *UniMod* реализован интерпретатор *XML*-описаний структурных конечных автоматов, которые пакет строит на основе указанных диаграмм. При запуске программы интерпретатор загружает в оперативную память *XML*-описание и создает экземпляры источников событий и объектов управления. В процессе указанные источники формируют события и направляют интерпретатору, который обрабатывает их в соответствии с логикой, описываемой автоматом. При этом он вызывает методы объектов управления, реализующие входные переменные и выходные воздействия.

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

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 [14] кода;
- запуск и отладка программы внутри среды разработки.

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

Проверка синтаксиса и семантики

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*). Данный 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. Так на рис. 3 приведен пример диаграммы с недостижимым состоянием.

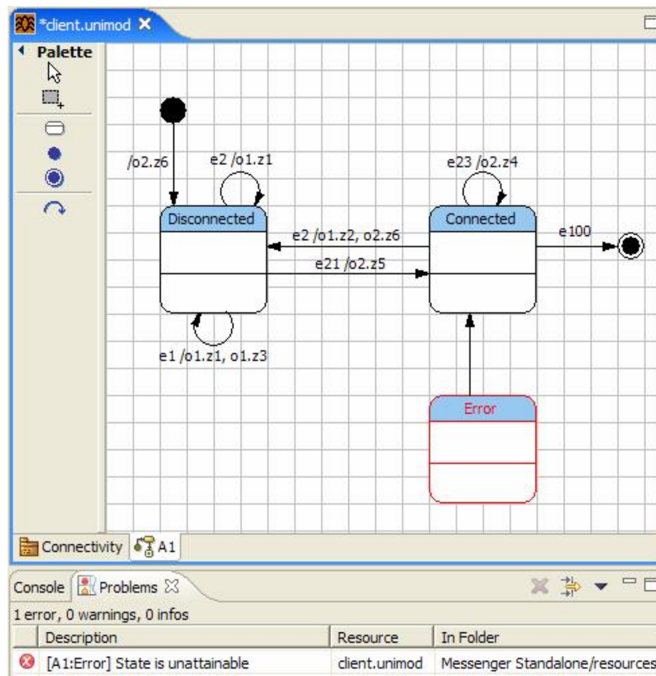

Рис. 3. Недостижимое состояние на графе переходов

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. 3, пользователю будет предложено добавить переход в это состояние из любого достижимого (рис. 4).

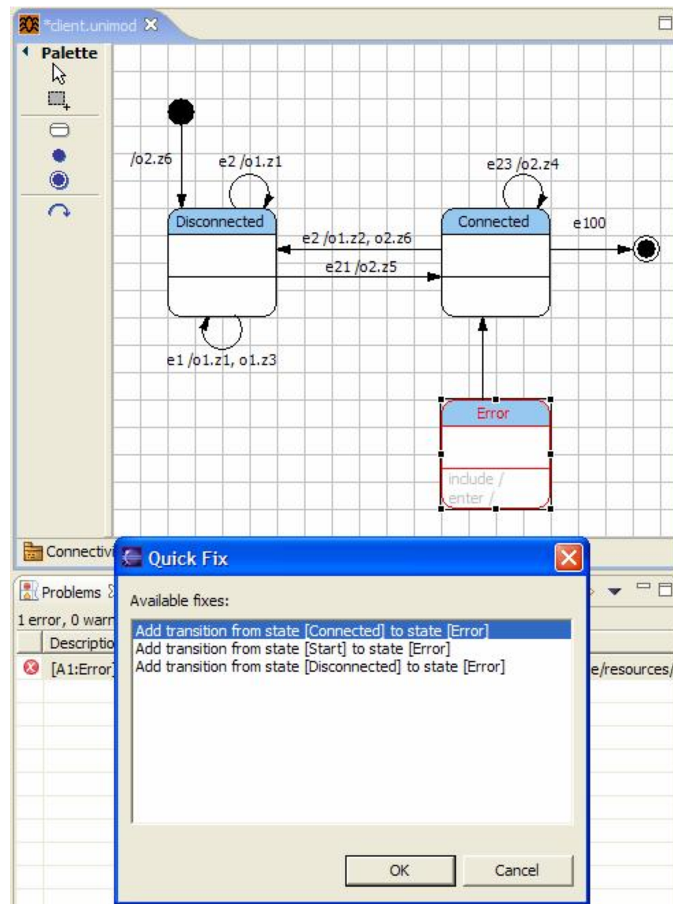

Рис. 4. Предлагаемые варианты исправления ошибки на диаграмме

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 [15].

Запуск программы

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.

Отладка

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.

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. Опыт разработки показывает, что приложения, работающие в одной предметной области, различаются не столько выполняемыми атомарными действиями, сколько логикой выполнения этих действий. Этот факт позволяет надеяться, что, создав для некоторой предметной области библиотеку источников событий и объектов управления, при программировании приложений из этой предметной области можно будет полностью отказаться от написания кода на текстовом языке программирования.

ЗАКЛЮЧЕНИЕ

В статье излагается подход к созданию графического языка для автоматно-ориентированного программирования.

Этот подход позволяет:

- сокращать объем кода на текстовом языке программирования;
- отказаться от текстового программирования для некоторой предметной области при наличии библиотеки источников событий и объектов управления для нее;
- 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 [16];
- формально и наглядно описывать поведение программ и модифицировать их, изменяя, в большинстве случаев, только графы переходов;

- упростить сопровождение проектов вследствие повышения централизации логики программ.

Исходные тексты, документация и примеры использования программного пакета *UniMod* представлены на сайте http://unimod.sourceforge.net.

ЛИТЕРАТУРА

1. **Mellor S. et al.** Executable UML: A Foundation for Model Driven Architecture. MA: Addison-Wesley, 2002. — P. 258.
2. **Буч Г., Рамбо Г., Якобсон И.** UML. Руководство пользователя. М.: ДМК, 2000. — 358 с.
3. **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
4. **Якобсон И., Буч Г., Рамбо Дж.** Унифицированный процесс разработки программного обеспечения. СПб.: Питер, 2002. — 458 с.
5. **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 — по адресу http://www.javable.com/columns/aop/workshop/02/)
6. *1st European Conference on Model-Driven Software Engineering*. Germany. 2003. http://www.agedis.de/conference/
7. *International Workshop* "e-Business and Model Based in System Design". IBM EE/A. SPb.: SPb ETU, 2004.
8. *OMG Model Driven Architecture*. http://www.omg.org/mda/
9. **Harel D., Gery E.** Executable Object Modeling with Statecharts. // In Proceedings of the 18th International Conference on Software Engineering, IEEE Computer Society Press, January 1996, pp. 246-257. http://citeseer.ist.psu.edu/harel97executable.html
10. *XJTek AnyState*. http://www.xjtek.com/anystates/
11. **Шалыто А.А., Туккель Н.И.** Танки и автоматы //BYTE/Россия. 2003. №2, с. 69-73. http://is.ifmo.ru/ (раздел «Статьи»).
12. **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/ (раздел «Статьи»).
13. **Грэхем И.** Объектно-ориентированные методы. Принципы и практика. М.: Вильямс, 2004. — 768 с.
14. **Фаулер М.** Рефакторинг. Улучшение существующего кода. М.: Символ-Плюс, 2003. — 623 с.
15. **Fruchterman T. M. J., Reingold E. M.** Graph Drawing by Force Directed Placement. // Software - Practice and Experience. 1991, № 21(11), pp. 1129-1164.
16. **Шалыто А.А.** Новая инициатива в программировании. Движение за открытую проектную документацию //PC Week/RE. 2003. № 40, с. 38-42.