Статья опубликована в журнале “Информационно-управляющие системы”. 2005. № 6, с. 9 – 22.

УДК 602-507

ВИЗУАЛЬНОЕ КОНСТРУИРОВАНИЕ ПРОГРАММ

Ф. А. Новиков,

канд. физ.-мат.наук

Санкт-Петербургский государственный политехнический университет

В статье описывается подход к проектированию и реализации набора компонентов, составляющих в совокупности инструментальное средство, основанное на UML и поддерживающее все фазы процесса разработки приложений.

Введение

В настоящее время эффективность и результативность процесса разработки программного обеспечения в целом оставляет желать лучшего. Несмотря на заметные успехи технологии программирования, остается весьма значительной доля проектов по разработке программного обеспечения, которые нельзя считать вполне успешными. Наряду с эффектными достижениями имеются и сравнительно многочисленные досадные неудачи (обзор современного состояния, например, в работе [1]). К сожалению, до сих пор слишком часто приходится делать вывод, что программирование – это рискованный бизнес, программы ненадежны, а программисты неуправляемы.

Дальнейший прогресс в области совершенствования процесса разработки зависит от множества факторов, таких как совершенствование аппаратной базы компьютеров, внедрение современных форм организации коллективного труда разработчиков, разработка и использование новых информационных технологий и так далее.

Многие полагают, и автор разделяет это мнение, что одним из ключевых факторов является совершенствование методов и инструментов визуального моделирования и (в перспективе) конструирования законченных приложений. Ожидается, что языки визуального моделирования должны стать такой же абстракцией над языками программирования, какой языки программирования являются над аппаратной платформой. Именно эта идея положена в основу инициативы, названной MDA (Model Driven Architecture) – архитектура, управляемая моделью [2]. Разработку на основе моделей будем называть модельно-ориентированной разработкой.

Наиболее многообещающим в этой области за последние годы представляется появление и распространение унифицированного языка моделирования *UML* [3]. Разумеется, *UML* – это исторически далеко не первая попытка ввести в программировании чертежи, понятные всем. Введение же в повсеместную практику программирования чертежей (именно *чертежей*, как определяющих документов, а не просто иллюстраций, относящихся непонятно к чему) является признаком того, что программирование действительно является инженерной областью деятельности. Пока же, по мнению автора, термин "инженер-программист" является скорее благим пожеланием, нежели констатацией квалификации.

Эта статья не является апологией *UML*. Очевидная перегруженность языка, рыхлое описание и туманная семантика бросаются в глаза. Но нельзя отрицать тот факт, что в *UML* в унифицированном виде удалось аккумулировать множество разнообразных плодотворных идей, а сообщество носителей языка давно переросло критическую отметку известности. *UML* является признанным лидером в обсуждаемой области и останется таковым в обозримом будущем. Семантика и нотация *UML* считаются здесь данностью, известной читателю, и не обсуждаются. Нас интересует, прежде всего, вопрос о том, что и как нужно сделать, чтобы потенциальные достоинства *UML* действительно дали ощутимые практические результаты. Наиболее важной является инструментальная поддержка *UML* – чтобы с пользой применять *UML* для визуального конструирования, нужно иметь адекватные инструментальные средства, поддерживающие *UML*. В настоящее время инструментальная поддержка *UML* не вполне удовлетворительна. На рынке присутствует множество инструментов, и все время появляются новые, но при более пристальном рассмотрении оказывается, что нам предлагают еще одно CASE средство традиционной архитектуры, поддерживающее еще одну неудобную нотацию. По мнению автора, вопрос о разумной поддержке *UML* не так прост, и требует исследования. Именно из-за плохой инструментальной поддержки *UML* используется в основном для рисования (необязательных!) диаграмм, и не более того. Только при наличии адекватной инструментальной поддержки, можно будет надеяться на распространение модельно-орентированного процесса разработки и получение выгод, которые обещает эта идея.

Целью данной работы является исследование вопроса о принципиальной реализуемости и о необходимых свойствах инструментальных средств, поддерживающих модельно-центрированную разработку приложений на основе *UML*.

План статьи состоит в следующем. В первом разделе обсуждается само понятие визуального конструирования приложений в сравнении с традиционными процессами разработки. Второй раздел посвящен обсуждению некоторых важных тенденций в методологии программирования и их связи с *UML*. В третьем разделе обсуждаются те общие требования к инструменту визуального конструирования, которые вытекают из наблюдений, сделанных в первых двух разделах. Последний раздел посвящен изложению известных автору способов реализации общих требований, выдвинутых в третьем разделе.

Визуальное моделирование и конструирование приложений

Прежде всего, охарактеризуем область применимости дальнейших построений. Многообразие различных типов приложений необозримо, и нельзя объять необъятное. Предметная область, характер использования, жизненный цикл приложения и другие факторы оказывают значительное влияние на характер процесса разработки. *UML – унифицированный* язык моделирования, но отнюдь не универсальный, и не единый. Утверждать, что *UML* является панацеей от всех детских болезней программирования было бы явным преувеличением – "серебряной пули нет" [4].

Для определения области применимости нужно выявить влияние *UML* на процесс разработки. Существует множество взглядов на процесс разработки – разнообразие подходов и моделей указывает на то, что инженерия программирования (software

engineering) является инженерной дисциплиной только по названию, а на практике используются, в основном, субъективные предпочтения и эмпирические правила. Это обстоятельство позволяет нам использовать здесь для формулировки тезиса о влиянии *UML* свой взгляд на технологию программирования, опуская для краткости детальные обоснования.

Все процессы, связанные с разработкой, можно разделить на три группы:

- процессы на уровне организации, то есть порядок проведения типового проекта (отчетность, бюджет, и так далее);
- процессы на уровне проекта, то есть отношения между людьми в процессе разработки (распределение ролей, оперативное планирование и так далее);
- процессы на уровне работника, то есть технология программирования в собственном смысле данного термина.

Оценивать качество процессов можно с помощью различных показателей – надежность разрабатываемого программного обеспечения, затраты на разработку, управляемость процесса и так далее. Для различных типов приложений применяются разные системы показателей. Здесь мы рассматриваем заказную разработку отчуждаемых программных продуктов, то есть предполагается, что в процессе разработки взаимодействуют, по меньшей мере, две стороны, которые для простоты мы обозначим устаревшими, но привычными терминами "заказчики" и "разработчики". Кроме того, ограничимся рассмотрением таких проектов, в которых вершины "треугольника качества" (ресурсы, качество, функциональность) не являются жестко фиксированными. Другими словами, мы рассматриваем "типичные" приложения, в которых требования являются достаточно гибкими в определенных пределах: например, допустимо пожертвовать некоторыми не приоритетными функциями, чтобы уложиться в заданные ресурсы по времени, или добавить трудовые ресурсы, чтобы обеспечить приемлемый для заказчика уровень качества. Для таких проектов нас интересует один интегральный показатель – продуктивность, которую здесь мы упрощенно определим как отношение прибыли к трудозатратам. Образно говоря, нас интересует процесс разработки достаточно полезного и надежного приложения за приемлемые деньги. В этих условиях, по субъективным наблюдениям автора, рост продуктивности от вложений в улучшение процессов трех выделенных выше групп относятся примерно как 1 : 3 : 10. Другими словами, рубль вложений в технологию программирования на уровне работника дает в десять раз больший прирост продуктивности по сравнению с вложением в "стандартный процесс". Еще раз оговоримся, что все эти утверждения не имеют универсального характера и не являются законами природы – это не более чем эмпирическое обобщение наблюдений автора, и они вряд ли могут быть распространены за пределы оговоренных ограничений.

Что же влияет на продуктивность программирования на уровне работника? По убеждению автора, таких факторов два:

- сокращение количества внеплановых изменений артефактов;
- увеличение объема повторно использованных артефактов.

Первый фактор, попросту говоря, означает уменьшение количества грубых ошибок, допускаемых на фазах анализа и проектирования (прочими ошибками можно пренебречь – стоимость их устранения при использовании современных технологий незначительна).

Второй фактор означает увеличение повторного использования сделанного ранее. Речь идет о повторном использовании всех видов артефактов: законченных компонентов, фрагментов кода, архитектурных решений и так далее. В частности, применение образцов проектирования мы относим к повторному использованию.

Рассмотрим упрощенную (но не противоречащую наиболее распространенным моделям, например, [5]) схему процесса разработки (рис. 1).

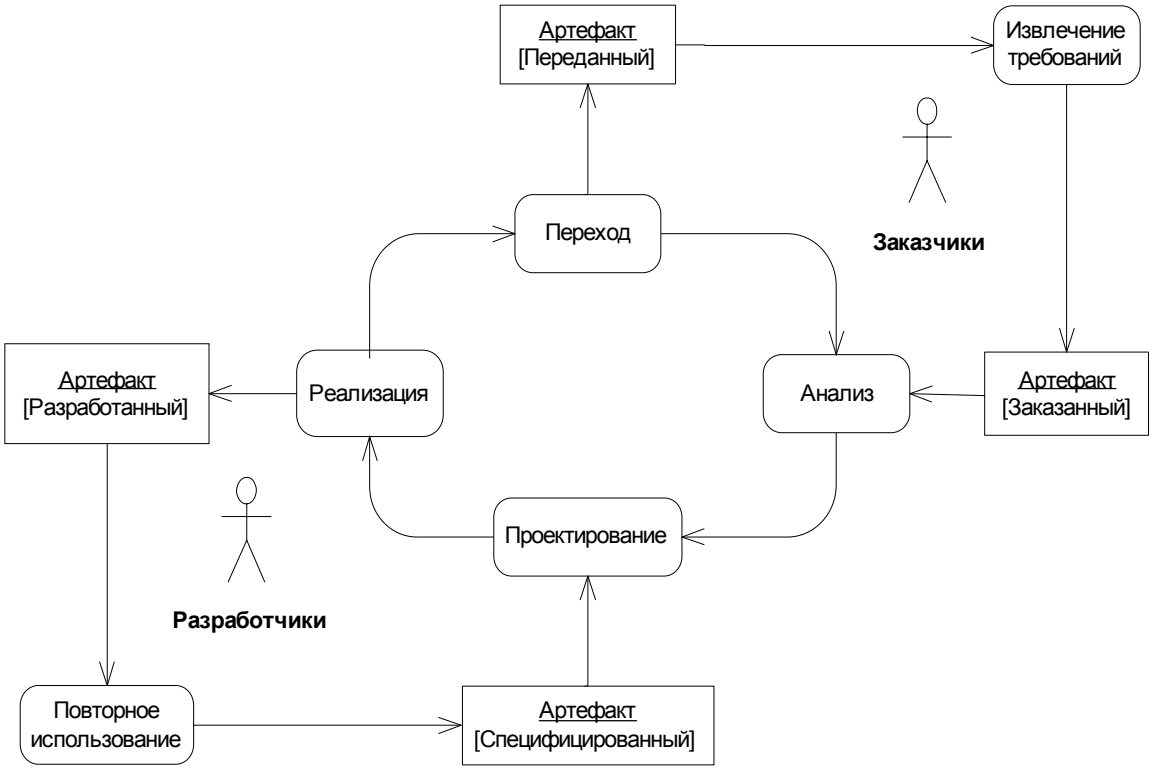

Рис. 1. Циклы повышения продуктивности

Обсуждение последовательности и названий основных фаз процесса, а также связи между фазами по данным в процессе и условия переходов между фазами мы оставляем в стороне, поскольку для наших целей здесь это несущественно (в этой статье мы не намерены предлагать вниманию читателю новый "стандартный" процесс).

Выделим два "боковых" цикла движения артефактов в процессе, которые мы называем циклами повышения продуктивности.

- На стороне заказчика происходит анализ и извлечение требований. В нормальной ситуации требования возникают не "из головы" заказчика, а в результате анализа предметной области и, в частности, анализа использования существующего программного обеспечения. Движение артефактов в этом цикле определяет величину первого фактора повышения продуктивности:

неадекватные требования порождают ошибки проектирования, адекватные требования позволяют их избежать.

- На стороне разработчика вновь созданные (или модифицированные) артефакты подготавливаются для повторного использования: документируются, обобщаются, помещаются в репозиторий и так далее. Движение артефактов в этом цикле определяет величину второго фактора повышения продуктивности: практика показывает, что эффективно повторно использовать удается только те решения, которые были для этого специально подготовлены.

Третий (не указанный на рис. 1) цикл движения артефактов – это стандартный цикл в процессе разработки: требования – проект – реализация – поставляемый продукт.

Наш основной тезис о влиянии *UML* на процесс разработки состоит в том, что *UML* позволяет унифицировать представление всех артефактов во всех циклах, в том числе, прежде всего в циклах повышения продуктивности. (Заметим, что унификация представления артефактов в основном цикле – это, по нашему мнению, центральная идея модельно-орентированной разработки). Если участникам процесса разработки приходится знать и применять десяток (а то и два) различных способов описания артефактов и технологий их создания, то трудно ожидать высокой продуктивности. Действительно, преобразования форматов, изучение специальных приемов, "ручная заточка" инструментов в каждом проекте трудоемки, замедляют скорость движения артефактов и чреваты ошибками. Если же в руках участников процесса есть универсальный инструмент, знакомый всем участникам и адекватный для всех фаз процесса, то индивидуальная производительность (и продуктивность), очевидно, должны иметь тенденцию к росту. В сущности, наш тезис является парафразой известного положения "о пользе чертежей", которое нам представляется бесспорным. Таким образом, *UML* оказывает положительное влияние на продуктивность процесса разработки, если он действительно применяется, причем в соответствии со своим назначением: "для спецификации, документирования, конструирования и визуализации всех артефактов в процессе разработки программного обеспечения" [5].

Рассмотрим, как соотносится сегодняшняя практика использования *UML* с декларированным выше назначением языка.

По мнению автора, можно выделить три основных варианта использования *UML* (см. также [9]), представленные на рис. 2.

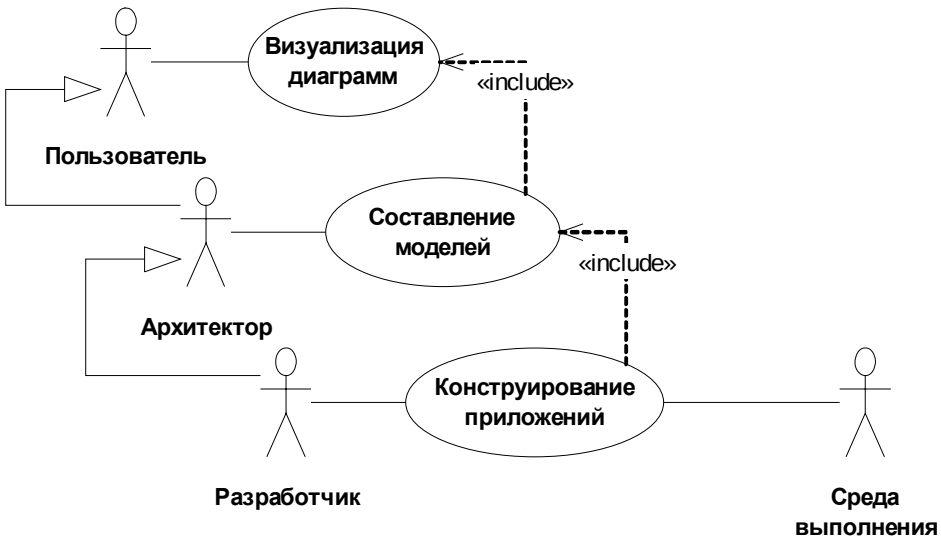

Рис. 2. Варианты использования *UML*

Вариант использования "Визуализация диаграмм" подразумевает изображение диаграмм *UML* с целью обдумывания, обмена идеями между людьми, документирования и тому подобного. Значимым для пользователя результатом в этом случае является само изображение диаграмм. Вообще говоря, в этом варианте использования языка поддерживающий инструмент не очень нужен. Иногда рисование диаграмм от руки фломастером с последующим фотографированием цифровым аппаратом может оказаться практичнее.

Вариант использования "Составление моделей" подразумевает создание и изменение модели системы в терминах тех элементов моделирования, которые предусматриваются метамоделью *UML* [7]. Значимым результатом в этом случае является машинно-читаемый артефакт с описанием модели. Мы будем для краткости называть такой артефакт просто моделью, деятельность по составлению модели называть моделированием, а субъекта моделирования называть архитектором (потому что соответствующие существительные – модельщик, модельер – в русском языке уже заняты).

Вариант использования "Конструирование приложений" подразумевает детальное моделирование, проектирование, реализацию и тестирование приложения в терминах *UML*. Значимым для пользователя результатом в этом случае является работающее приложение, которое может быть, например, скомпилировано во входной язык конкретной системой программирования или сразу интерпретировано средой выполнения инструмента. Этот вариант использования наиболее сложен в реализации, но именно он лежит в основе концепции модельно-орентированной разработки. Термин "конструирование приложений" означает по существу то же самое, что и "разработка приложений". Мы используем этот новый термин, чтобы отличать модельно-орентированную разработку от традиционных подходов к разработке. Кроме того, "конструирование приложений" больше похоже по звучанию на термин "construction", которые используют в этом контексте авторы *UML*.

Однако современные инструменты поддерживают указанные варианты использования далеко не в равной степени. Все инструменты умеют (плохо или хорошо) ви-

зуализировать все типы диаграмм *UML*, некоторые инструменты позволяют построить модель, допускающую какое-то дальнейшее использование, но только немногие инструменты могут генерировать исполнимый код и отнюдь не для всех диаграмм. Имеется множество практических и организационных причин, по которым указанные выше варианты использования неравноправны и в разной степени поддержаны в современных инструментах.

Практические причины заключаются в том, что визуальное конструирование наряду с очевидными достоинствами имеет и столь же очевидные недостатки, на которые не стоит закрывать глаза. В табл. 1 указаны наиболее существенные, по нашему мнению причины.

Таблица 1. Достоинства и недостатки визуального конструирования приложений

Достоинство	Недостаток
Диаграммы наглядны	Диаграммы не компактны
Высокий уровень абстракции	Плохо выразимы некоторые простые программистские конструкции
Высокий уровень формализации	Сложная для освоения нотация

Достоинства *UML* не во всех случаях перевешивают его недостатки, поэтому существуют (и, скорее всего, будут существовать) категории пользователей языка, не заинтересованные в поддержке варианта использования "Конструирование приложений". Такие пользователи не согласны платить за ненужные им (и дорогостоящие!) возможности. Производители инструментов не вправе игнорировать потребности рынка. (По чисто субъективному впечатлению автора, количества пользователей, жизненно заинтересованных в трех перечисленных выше вариантах использования, соотносятся примерно как 10 : 3 : 1). Например, диаграммы в этой статье подготовлены с помощью *Microsoft Visio*, а этот инструмент и не претендует на визуальное конструирование приложений.

С другой стороны, довольно распространенным является отношение к *UML* как к модной графической надстройке над обычным языком программирования "высокого" уровня (речь идет о так называемых языках третьего поколения *3GL*), которая имеет (частично) графический, а не только текстовый синтаксис. Несколько упрощая, можно сказать, что разработка сводится к рисованию диаграммы классов, после чего тела методов вписываются вручную. При таком использовании *UML* фактически оказывается средством автоматического добавления графических комментариев к программе. Действительно, структура наследования (для небольших проектов!) наглядно видна на диаграмме классов, но, по большому счету, этим визуальное моделирование и ограничивается. Мы не склонны считать такую практику визуальным конструированием приложений, во всяком случае, рассчитывать на значительное повышение продуктивности точно не стоит. *UML можно* использовать как еще один традиционный язык программирования, но, как мы рассчитываем показать в оставшейся части статьи, делать этого *не нужно*!

Организационные причины заключаются в том, что процесс стандартизации *UML* неоправданно затянулся (например, стандарт *UML* 2.0 находится в состоянии "финализации" уже более года). Данную ситуацию нетрудно объяснить. *UML* был задуман и определен как язык-оболочка, унифицирующий как можно больше известных и зарекомендовавших себя на практике приемов визуального моделирования и конструирования. Такой подход с необходимостью ведет к сложности и громоздкости языка и его описания. Некоторые полагают, что "*UML* рухнет под собственной тяжестью" [10]. Мы надеемся показать ниже, в третьем разделе статьи, что хотя риск такого печального исхода действительно велик, но он распространяется в основном на производителей инструментов, но не на пользователей языка, а потому, принимая предупредительные меры, есть надежда избежать худшего.

Сложность и громоздкость языка привели к тому, что подавляющее большинство пользователей знакомится с языком не по оригинальным источникам и утвержденным спецификациям стандартов, которые слишком сложны для восприятия, а с помощью популяризаторов и интерпретаторов, среди которых, к сожалению, немало недобросовестных и некомпетентных людей. Наблюдения автора показывают, что многие пользователи *UML* владеют языком поверхностно. В результате поставщики инструментов также не утруждают себя следованием стандарту – например, не проверяют правила непротиворечивости, определенные в стандарте [11], не поддерживают (или хуже того, искажают) стандартные конструкции и канонические диаграммы – да и зачем напрягаться, если имеется в виду только рисование необязательных картинок?

Если сопоставить нынешние инструменты, поддерживающие *UML*, с традиционными системами программирования, то вариант использования "Рисование диаграмм" соответствует наличию простого текстового редактора, вариант использования "Моделирование систем" соответствует синтаксическому анализатору, вариант использования "Конструирование приложений" соответствует генерации кода и наличию административной системы времени выполнения. Языки "программирования", которые не подразумевают выполнения, иногда используются – например, для публикации алгоритмов. Однако чаще производители систем программирования, заявляя о поддержке языка, подразумевают полную поддержку, включая точное соответствие стандартам и многое сверх того, например, наличие развитых библиотек готовых к использованию компонентов. В случае с *UML* ситуация прямо противоположная: заявляя о поддержке *UML* некоторые производители инструментов не считают себя связанными серьезными обязательствами. Действительно, правомерно ли утверждать, что инструмент поддерживает описание поведения с помощью *UML*, если он позволяет *нарисовать* диаграмму состояний и не более того? Между тем даже известнейшие инструменты (например, *Rational Rose* и *Together Control Center*), декларируя поддержку *UML* различных версий, *ничего* не умеют делать с упомянутым базовым средством описания поведения в *UML* (то есть, с конечными автоматами).

Таким образом, сравнение инструментов по критерию степени поддержки языка расплывчато и субъективно. Можно предложить следующие объективные критерии того, можно ли считать инструмент полностью поддерживающим какой-либо вариант использования *UML*.

- Инструмент полностью поддерживает вариант использования "Визуализация диаграмм", если он позволяет отобразить все графические конструкции, предусмотренные описанием нотации в стандарте.
- Инструмент полностью поддерживает вариант использования "Составление моделей", если он позволяет построить объекты и их комбинации для всех классов метамодели, предусмотренные описанием семантики в стандарте.
- Инструмент полностью поддерживает вариант использования "Конструирование приложений", если он позволяет преобразовать модель приложения в работающее приложение.

Заметим, что варианты использования очевидным образом включают друг друга и если инструмент полностью поддерживает разработку, то он, скорее всего, поддерживает также моделирование и рисование. Поставим вопрос: как можно определить, что предложенные критерии выполнены, и инструмент действительно поддерживает тот или иной вариант использования? Нетрудно проверить, что инструмент умеет рисовать все, что нужно – для этого хватит получаса при условии знания языка. Можно проверить, что инструмент создает все элементы модели, предусмотренные стандартом, хотя это и хлопотно – счет тестовых сценариев пойдет на сотни, так как в языке *много* отдельных конструкций. Но с проверкой третьего варианта использования дела обстоят сложнее. Операционная семантика *UML* туманна, если не сказать хуже (за что недоброжелатели подвергают язык постоянной и справедливой критике). В то же время, по нашему мнению, плохое определение или даже отсутствие операционной семантики в стандарте *UML* еще не является непреодолимым препятствием для реализации варианта использования "Конструирование приложений". Язык устроен таким образом, что семантику можно *доопределить* прямо в инструменте. Стандартные механизмы расширения и точки вариации семантики оставляют достаточный простор. Похожая проблема (но менее остро) строит и перед разработчиками традиционных систем программирования. Обычно ее решают чисто практически: договариваются о представительном наборе *примеров*, и если прогон примеров дает ожидаемые результаты, то система программирования считается приемлемой. Однако стандартного набора примеров визуальной разработки пока нет, и приходится их выбирать и придумывать.

Традиционно в качестве теста для инструмента разработки используют сам этот инструмент разработки. Например, язык программирования хорош, если на этом языке можно написать хороший компилятор данного языка. Такой прием обычно называют раскруткой (bootstrapping). Мы считаем достаточно представительным примером приложения сам инструмент визуального конструирования. Таким образом, окончательно наш критерий можно сформулировать следующим образом: *инструмент полностью поддерживает UML, если он допускает полную раскрутку*. Другими словами, если на вход инструмента подать модель инструмента, созданную в этом инструменте, то на выходе мы получим тот же инструмент (тот же в функциональном смысле, то есть так же работающий). Проверить это (до некоторой степени) можно с помощью сценария, приведенного на рис. 3. Здесь предполагается, что инструмент умеет выполнять последовательности команд, и что процесс построения любой модели можно описать такой последовательностью команд (трудно представить себе обратное). Тогда если инструменту подать последовательность команд, описывающую построение его модели (на рис. 3 она обозначена

M0), то инструмент должен построить некоторую модель самого себя (на рис. 3 она обозначена M1). Если теперь выполнить модель M1 (запустить интерпретацию или же сгенерировать код и запустить его) и еще раз подать на вход этому сеансу выполнения ту же последовательность команд M0, то должна быть построена еще одна модель (на рис. 3 она обозначена M2). Если окажется, что M1 и M2 совпадают, то, по нашему мнению, это позволяет с достаточной степенью уверенности считать, что инструмент функционально тождественен своей модели и вариант использование "Конструирование приложений" поддерживается.

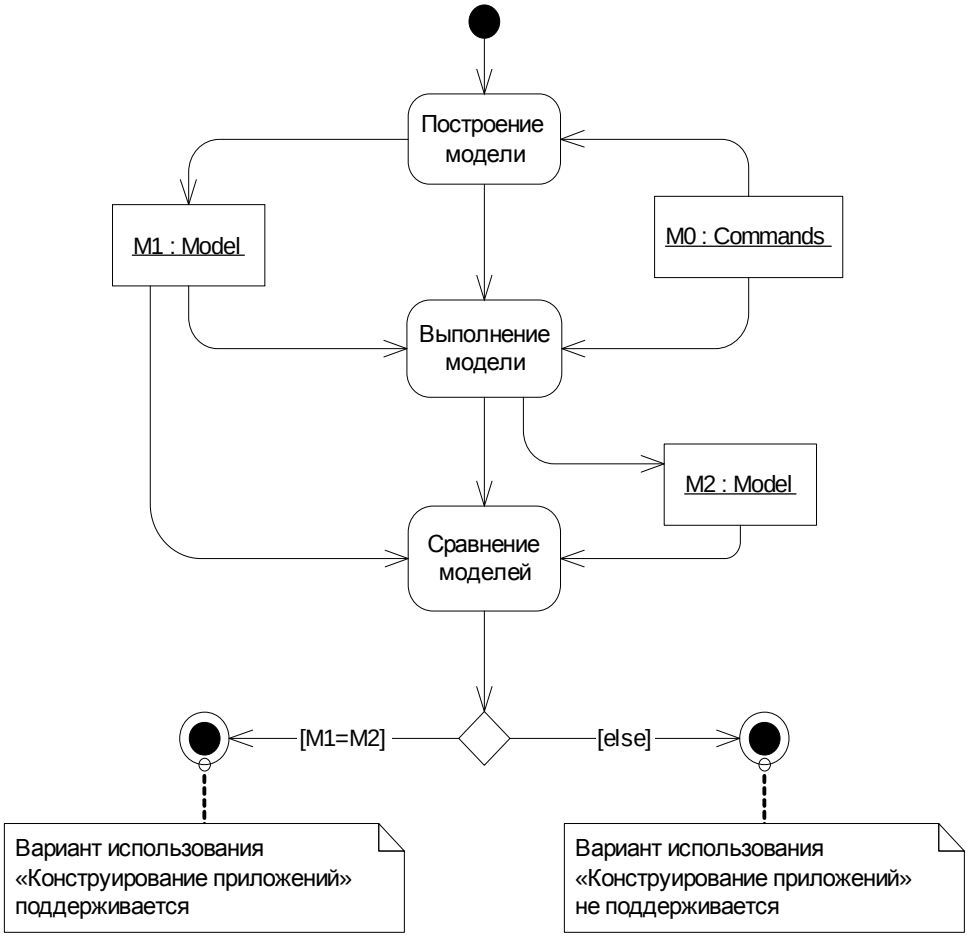

Рис. 3. Сценарий проверки варианта использования "Конструирование приложений"

Отметим, что в настоящее время не существует инструментов, полностью поддерживающих *UML* согласно приведенному критерию.

Современные тенденции в методологии программирования

В этом разделе мы рассмотрим некоторые концепции методологии программирования, про которые точно известно, что их применение в практическом программировании положительно влияет на продуктивность. Обозначим их:

- компонентно-ориентированное программирование;
- предметно-ориентированное программирование;

- аспектно-ориентированное программирование.

Мы не собираемся детально излагать эти концепции – предполагается, что они хорошо известны читателю (возможно, под другими названиями, поэтому основные идеи во избежание недоразумений мы все-таки повторим). Наша цель в этом разделе состоит в том, чтобы установить, насколько адекватен *UML*, лежит ли этот язык в русле основных тенденций или "идет против течения".

Компонентно-ориентированное программирование старо как само программирование, если не старо как мир, поскольку является реализацией принципа "разделяй и властвуй". Во всяком случае, указать первоисточник этой идеи затруднительно. Сама же идея очевидна: целесообразно конструировать программы не из мелких деталей (операторов языка программирования), а из более крупных блоков (компонентов). Выгоды очевидны:

- составить большую программу из готовых компонентов намного легче, чем написать "с нуля" (такую же по возможностям) программу;
- готовые компоненты являются тем, что можно и нужно повторно использовать.

Понятие "компонент" довольно расплывчатое, были предложены и использованы многочисленные технологии и механизмы, поддерживающие компонентное программирование, и компоненты в них понимаются и описываются немного поразному. Грубо все разнообразие видов программных компонентов можно поделить на три типа:

- неструктурированные фрагменты исходного кода на языке программирования;
- структурные части исходного кода программ (их часто называют модулями);
- скомпилированные исполнимые части программ (например, библиотеки динамического связывания).

В первом случае решение прямолинейное и понятное, но многого ожидать от него не приходится: если текст программы собирается из компонентов путем копирования и вставки, то очень вероятны ошибки, связанные с "настройкой по месту" скопированного фрагмента. Впрочем, в отношении компонентного программирования методом копирования и вставки инструменты, основанные на *UML*, не хуже других: фрагменты моделей можно также легко копировать и вставлять, как и тексты.

Компоненты на уровне модулей исходного кода удобная вещь, если используется один язык программирования с хорошо определенным понятием модуля (например, *Pascal* или *Java*). Если же требуется использовать компоненты, написанные на разных языках и для разных платформ, то возникают проблемы. В *UML* имеется хорошо определенное понятие модуля (пакет), а поскольку язык по самой своей сути претендует на независимость от платформы, число и сложность проблем при сборке моделей из пакетов *UML* оказываются меньше. Заметим, что при описании *UML* 2.0 [8] введена и активно используется для описания самого языка своеобразная операция слияния пакетов («merge»). Этот вид компонентного программирова-

ния специфичен для модельно-центрированной разработки. Трудно представить себе столь же избирательную операцию по сборке на уровне текстов. Опуская детали, можно сказать, что слияние позволяет использовать пакет, как компонент, извлекая из него ровно то, что нужно при сборке модели. Таким образом, компонентное программирование на уровне исходных кодов поддержано в *UML* по меньшей мере не хуже, чем у конкурентов.

Компонентное программирование на уровне исполнимых компонентов имеет наибольшее практическое применение в настоящее время. Разрабатываются и применяются весьма изощренные механизмы, призванные обеспечить компонентное программирование на объектном уровне. Наиболее яркий пример последнего времени – это, несомненно, .NET [12]. Общая среда выполнения, обмен метаинформацией и другие механизмы обеспечивают достаточно гибкие возможности сборки программ из компонентов. Заметим, что разработчики .NET были связаны искусственным ограничением, состоящим в том, что исходный код компонента считается недоступным во время использования компонента.

По мнению автора, защита интеллектуальной собственности разработчиков путем сокрытия исходных кодов программ – это укоренившийся, но общественно вредный анахронизм. Практика сокрытия исходного кода невыгодна самим разработчикам. Действительно, сокрытие исходного кода затрудняет его повторное использование, и упущенная выгода намного перевешивает ущерб от злонамеренного нарушения авторских прав. Кстати говоря, в современной открытой информационной среде контрафактное использование кода программ технически вполне возможно выявить и пресечь. В модельно-центрированной разработке сокрытие моделей – нонсенс, противоречащий самой сути подхода. Общественные движения, направленные на расширение публичного доступа к информации экономически обоснованы, и потому победа будет за ними. В этой связи, по нашему мнению, особого упоминания заслуживает движение за открытую проектную документацию [13], поскольку оно в наибольшей степени соответствует по духу модельно-ориентированной разработке.

Возвращаясь к компонентному программированию на *UML*, отметим следующее. Диаграммы компонентов *UML* предоставляют достаточно наглядные средства для описания состава и структуры взаимодействия компонентов (особенно с учетом симметричной нотации для требуемых и предоставляемых интерфейсов, введенной в *UML* 2.0), а диаграммы взаимодействия являются достаточным ресурсом для описания самого взаимодействия. Важным, на наш взгляд, является появление в *UML* 2.0 специального вида машин состояний, описывающих протокол взаимодействия компонентов.

Мы видим, что применение *UML* ничуть не противоречит компонентному программированию в любом его толковании. Более того, мы рискнем предположить, что применение *UML* в ближайшем будущем позволит рассматривать еще одно толкование понятия компонента: компонент – это экземпляр образца проектирования, то есть решение задачи путем применения образца проектирования в конкретном контексте.

Подводя итоги обсуждению компонентного программирования, подчеркнем еще раз, что компонентное программирование явно поощряет повторное использование, а потому ведет к повышению продуктивности. Приятно отметить, что в качестве средства компонентного программирования *UML* не только не отстает, но даже несколько опережает распространенную на сегодня практику.

Обратимся теперь к предметно-ориентированному программированию. Основная идея здесь также банальна: программировать нужно в терминах конкретной предметной области, тогда программы для этой предметной области будут простыми, понятными, надежными и хорошими во всех отношениях. Мы не будем обсуждать здесь предметно-ориентированное программирование во всех деталях, тем более что они значительно разнятся у различных авторов. Очень краткий, но емкий обзор приведен в работе [14]. Отметим только одно обстоятельство, важное в контексте нашего рассмотрения. Программировать в терминах предметной области могут и должны специалисты в этой предметной области, которые, хотя, может быть, и не являются профессиональными программистами, но показывают очень высокую продуктивность. Личные наблюдения автора полностью согласуются с этим тезисом, более того, у нас есть объяснение: если в циклах повышения продуктивности (рис. 1) заказчики и разработчики – одно действующее лицо, то движение артефактов заметно ускоряется. С других исходных позиций, но к тому же выводу приходят экстремальные программисты настаивая на постоянной вовлеченности заказчика в процесс разработки.

Чтобы оценить пригодность *UML* для предметно-ориентированного программирования, заметим следующее. Обычно апологеты предметно-ориентированного конструирования приложений акцентируют внимание на выразительности предметно-ориентированных программ и простоте их составления, но это только верхушка айсберга! Квалифицированная формализация предметной области, разработка развитых пакетов прикладных программ для нее, изобретение и реализация специфических интерфейсных средств остаются как бы за кадром. Системы предметно-ориентированного программирования просты в использовании, но сложны в реализации. Например, известная программа *Excel*, которую мы считаем классическим примером предметно-ориентированной системы программирования, намного превосходит по богатству возможностей современные системы программирования на обычных универсальных языках программирования. Сложную задачу по обработке числовых данных, которую опытный пользователь *Excel* решает "в два счета" с помощью сводных таблиц и многочисленных надстроек *Excel* (например, надстройка "поиск решения"), команда обычных профессиональных программистов на *C++*, скорее всего, вообще не сможет запрограммировать за разумное время – не хватит квалификации в предметной области.

Многие справедливо отмечают, что *UML* слишком сложный язык, и конечному пользователю – специалисту в предметной области – трудно было бы построить содержательную модель так же легко, как это делается с помощью специализированного предметно-ориентированного языка. Критика направлена мимо цели. Во-первых, построить модель предметной области на *C++* ничуть не проще. Во-вторых, делать этого не нужно. Действительно, зачем насильно втискивать предметную область в *UML*? Наоборот, целесообразно воспользоваться гибкостью *UML*

и создать специализированный предметно-ориентированный язык, в котором модель запишется легко и просто. И здесь *UML* может, по нашему мнению, составить конкуренцию традиционным подходам. Главным конкурентным преимуществом *UML* при этом является возможность метамоделирования.

Метамоделирование – это возможность изменять метамодель, то есть тот язык (включая семантику), на котором пользователь строит свои модели. Поскольку мы договорились, что рассматриваем инструмент, которому полностью подвластна его собственная модель (метамодель), то возможности метамоделирования на *UML* намного богаче, чем на обычных языках программирования. Поясним примером: обычная система программирования, может быть, позволяет переопределить стандартную библиотеку встроенных функций, меняя до некоторой степени семантику языку, но вряд ли позволит вмешиваться в протокол вызова функций или в механизм проверки типов. Для инструмента, полностью поддерживающего *UML*, все возможно.

Последняя тенденция, которую мы хотим затронуть – аспектно-ориентированное программирование. Это сравнительно новое изобретение, но все на ту же тему повторного использования кода. Идея, как обычно проста и базируется на следующем наблюдении: большая часть кода в реальных программах имеет рутинный характер – это повторы практически одинаковых фрагментов, написание которых требует не глубоких раздумий, а только сугубого внимания. Ярким примером, на котором часто объясняют аспектно-ориентированное программирование, является ведение программой протокола своего выполнения. Проблема состоит в том, похожие операторы записи в файл протокола (или вызов соответствующей процедуры) должны быть помещены в сотни разных мест исходной программы. Эти операторы, может быть, зависят от контекста, в котором выполняется протоколирование, но, в общем, очень похожи. Идея состоит в том, чтобы написать заготовку нужных операторов один раз ("определить аспект"), а также один раз определить правило модификации аспекта в соответствии с контекстом и один раз определить правило нахождения мест в тексте программы, куда эти операторы должны быть вставлены. Система аспекто-ориентированного программирования автоматически вставит нужные операторы везде, где требуется. Экономия трудозатрат, повышение надежности, улучшение читабельности, удобство повторного использования очевидны.

Классические системы программирования реализуют идею аспектов с помощью специального модуля (препроцессора или прохода транслятора), который обычно называют "ткач", поскольку это модуль как бы "вплетает" аспекты в основной код программы. Насколько же *UML* приспособлен для реализации этой идеи? Заметим, во-первых, что в *UML* имеется зависимость со стандартным стереотипом «extend», которая прямо реализуют идею расширения варианта использования "извне". Правда, такая зависимость предусмотрена только для вариантов использования, но можно определить соответствующие стереотипные зависимости и для других элементов моделирования. Мы хотели бы обратить внимание на то, что можно пойти гораздо дальше. В сущности, аспекты – это пример *нелокальной* операции над текстом программы (другой пример – так называемый "рефакторинг"). Такого рода нелокальные операции с текстами не слишком широко распространены и применяются реже, чем они того заслуживают, из-за сугубой бедности структуры текста

программы. В лучшем случае мы можем опираться на древовидную синтаксическую структуру. Между тем в модели *UML*, особенно при наличии средств метамоделирования, для формулировки правил нелокальных преобразований модели доступно значительно больше информации: наряду с отношением владения между элементами модели, которое соответствует обычной синтаксической иерархии, можно использовать причинно-следственные связи, выраженные зависимостями со стереотипом «refine», строгую типизацию элементов, специфические отношения на диаграммах и многое другое. Вот пример подобного преобразования модели: поиск и выделение в конечном автомате (диаграмме состояний) составных состояний, которые могли бы объединять состояния исходного автомата. Обсуждение подобных нелокальных систематических преобразований модели только начинается: здесь пока больше неясных вопросов, чем готовых ответов.

Таким образом, мы видим, что концепции и средства *UML* в основном хорошо согласованы с современными тенденциями в программировании, и дело за тем, чтобы обеспечить должную инструментальную поддержку.

Общие требования к инструменту визуального конструирования

Первое требование к инструменту визуального конструирования состоит в том, что инструмент должен отвечать своему названию, то есть он должен позволять конструировать приложения. В первом разделе статьи мы сформулировали критерий того, что инструмент поддерживает вариант использования "Разработка приложений" – инструмент должен быть достаточен для воспроизведения себя самого по своей модели. Это требование главное, но не единственное.

Другие требования вытекают из объективных обстоятельств и ограничений. Заметим, что мы обсуждаем инструменты, поддерживающие язык *UML*, который, в свою очередь, является предметом международной стандартизации. Далее, напомним, что визуальное конструирование не только академически интересно, но и нацелено на повышение продуктивности разработки, а значит, инструмент должен быть согласован с современными тенденциями методологии программирования. Для определения вытекающих требований целесообразно использовать тот же принцип самоприменимости. Отсюда следует, что инструмент на своем примере должен демонстрировать следование принципам компонентного, предметно-ориентированного и модельно-орентированного программирования. (Аспекто-ориентированное программирование мы не включили в число основных требований, так как оно полезно, но не первостепенно). Таким образом, наши требования к инструменту суть:

- соответствие стандартам;
- компонентная архитектура;
- предметная ориентация;
- модельная ориентированность.

Обсудим эти требования чуть подробнее. Стандартизация вещь, безусловно, полезная, а в инженерных областях необходимая. Но само по себе формальное соответ-

ствие стандартам еще не является существенной ценностью для пользователей инструмента. Стандарт языка нужен для обеспечения унифицированного использования, совместимости разных инструментов и переносимости на различные платформы. Выше мы отметили, что возможность метамоделирования ведет к высокой гибкости инструмента визуального конструирования, намного превосходящей модифицируемость традиционных систем программирования. Особенности (например, механизмы расширения и точки вариации семантики) и, к сожалению, недоговоренности стандартного определения языка ведут к появлению диалектов. Модели, созданные разными разработчиками и с помощью различных инструментов, могут быть совершенно не похожи друг на друга, и мы не видим в таком разнообразии ничего плохого, если модели отвечают основному назначению: визуализации, спецификации, конструированию и документированию программных систем. Применительно к *UML* разумно выдвинуть требование о соответствии скорее духу, нежели букве стандарта: язык инструмента не должен *противоречить* стандарту (но может быть расширен и доопределен в нужных инструменту направлениях). Что касается совместимости различных инструментов, то владелец процесса стандартизации *UML* – Object Management Group – уделяет, конечно, внимание этому вопросу, но явно недостаточное. В качестве средства обеспечение совместимости предлагается использовать *XMI* (XML Metadata Interchange) [**Ошибка! Источник ссылки не найден.**] – специальное приложение *XML*, предназначенное для этой цели. Но со стандартизацией *XMI* дело обстоит еще хуже, чем с *UML* – отставание версий *XMI* от *UML*, несоответствие версий и так далее. Реальные промышленные инструменты кое-как понимают друг друга, но с явными ошибками. Впрочем, производителей инструментов это не очень беспокоит: каждый из них претендует на то, чтобы стать единственным нужным пользователю поставщиком инструментов моделирования. Требовать невозможного неразумно: достаточно обеспечивать совместимость в тех пределах, в которых (будет) определен *XMI*. Наконец, многоплатформенность для самого инструмента необязательна, так как одна из основных идей MDA – это кросс-платформенная генерация кода. Не стоит требовать, чтобы инструмент работал в любой среде: достаточно, чтобы он работал где-нибудь и умел конструировать приложения для выполнения в *нужной* среде.

Обратимся теперь к требованию компонентной архитектуры. Прежде всего, констатируем, что инструмент компонентного конструирования должен быть примером компонентного конструирования, а значит должен быть не "монолитным" программным продуктом, а конструктором инструментов. Другими словами, мы выдвигаем требование не просто компонентной архитектуры, а требование компонентой архитектуры, динамически изменяемой (расширяемой) пользователем. В любой момент компоненты могут быть изменены, удалены или добавлены в соответствии с потребностями конкретного случая применения инструмента. Такое требование не является чем-то исключительным: многие современные приложения используют идею надстроек. Надстройка – это компонент, реализующий специфические функции и включаемый в состав приложения по желанию пользователя. Особенностью выдвигаемого здесь требования является то, что инструмент должен являться инструментом разработки своих надстроек.

Обсудим теперь требование предметной ориентации. Предметная ориентация подразумевает использование специального языка (или языков), ориентированного на

предметную область. В данном случае предметной областью является метамоделирование и соответствующий предметно-ориентированный язык – MOF (Meta Object Facility) [16] – уже предложен той же организацией, что и *UML*. Однако применительно к MOF можно повторить сказанное об *XMI* – ход процесса стандартизации отстает от потребностей практики. В результате во многих ведущих проектах используются свои, альтернативные варианты предметно-ориентированных средств метамоделирования (см., например, [17]). Поэтому мы не склонны формулировать наше требование предметной ориентации в жестком формальном виде "MOF Compliant". Любые средства метамоделирования допустимы, если они достаточны и удобны. Можно использовать и свою метамодель, и свой язык, даже текстовый!

Наконец, последнее наше общее требование – модельная центрированность – является, по-видимому, самым важным, необычным и трудно выполнимым требованием. Необычность проявляется даже в том, что сама формулировка требования режет слух. Мы используем буквальный перевод английского термина model centric, просто потому, что пока не нашли более подходящего выражения по-русски. Речь идет о том, что модель является важнейшим и (в идеале) единственным абсолютно необходимым артефактом процесса разработки. Поясним это требование противопоставлением. При разработке приложения с помощью традиционной системы программирования исходный код программы имеет решающее значение. В процессе разработки могут использоваться какие-то другие типы артефактов (текстовые описания, диаграммы, тестовые примеры и тому подобное), но исходный код является главным: нет кода – нет и разработанного приложения. И наоборот, есть работающий код – есть и приложение. Все остальное необязательно. При модельно-орентированной разработке роль кода играет модель, наличие исполнимой модели является необходимым и достаточным условием, а исходный код на традиционном языке программирования является только возможным, но необязательным артефактом разработки. Отсюда следует, что все в инструменте должно быть представлено в форме модели *UML*, но отнюдь не следует, что в этой модели не должно быть ничего, кроме "стрелочек и квадратиков". Так же как и любой другая система программирования, система визуального конструирования опирается на набор предопределенных примитивов. По нашему мнению, эти примитивы отнюдь не обязаны ограничиваться только теми понятиями, через которые (фрагментарно!) определена операционная семантика *UML* – их явно недостаточно. В качестве примитивов можно использовать все что угодно, готовые компоненты любой сложности, лишь бы они были описаны в модели, а реализованы они могут быть вовне произвольным образом. В этой связи заметим, что вопрос об алгоритмической полноте *UML* в контексте нашего рассмотрения представляется хотя и любопытным, но достаточно праздным – напомним, что нашей главной целью является повышение продуктивности разработки программного обеспечения, а не построение еще одной модели вычислимости, тем паче "единой теории программирования".

Принципы реализации инструмента визуального конструирования

Имея поставленную цель, сформулированный критерий ее достижения и требования к решению, попробуем определить, что и как нужно сделать для достижения цели.

Прежде всего, следует определить, *что* нужно сделать. Мы вслед за А. Якобсоном считаем, что состав компонентов (надстроек) должен управляться вариантами использования. Например, на рис. 4 представлена одна из возможных моделей использования инструмента, полученная уточнением исходной модели, показанной выше на рис. 2.

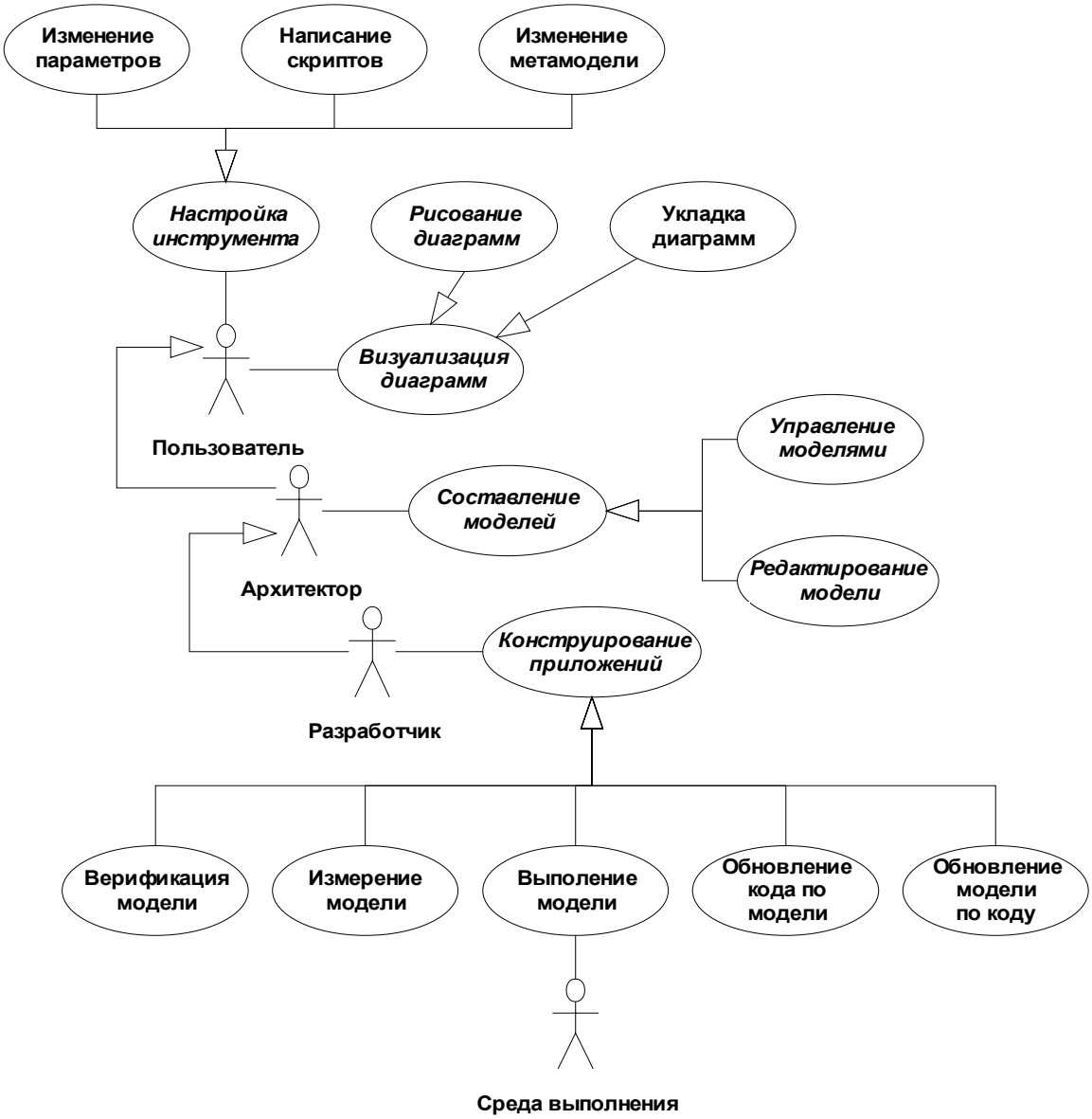

Рис. 4. Уточненная модель использования инструмента

По поводу модели использования необходимо сделать два важных замечания, относящихся к процессу разработки и специфичных для визуального конструирования приложений.

Во-первых, модель использования действительно управляет процессом визуального конструирования, но отнюдь не обязательно является описанием компонентной структуры и состава реализуемых настроек. Может быть так, что вариант использования однозначно соответствует отдельному компоненту или надстройке, но ча-

ще однозначного соответствия не наблюдается: компонент поддерживает несколько вариантов использования, а вариант использования реализуется в нескольких компонентах. В этом случае вариант использования следует считать скорее описанием аспекта, "размазанного" по структуре кода.

Во-вторых, при визуальном конструировании модель использования не является жестко фиксированной, подобно тому, как не являются жестко фиксированными требования в традиционном процессе разработки. Собственно, модель использования – это и есть требования. Поэтому не требуется отдельных процессов и инструментов для управления требованиями, поскольку требования, по крайней мере, функциональные, являются частью основного артефакта – модели приложения – и управляются совместно с остальными элементами модели. По нашему мнению, итеративный процесс визуального конструирования в целом можно представить диаграммой на рис.5Рис. 5.

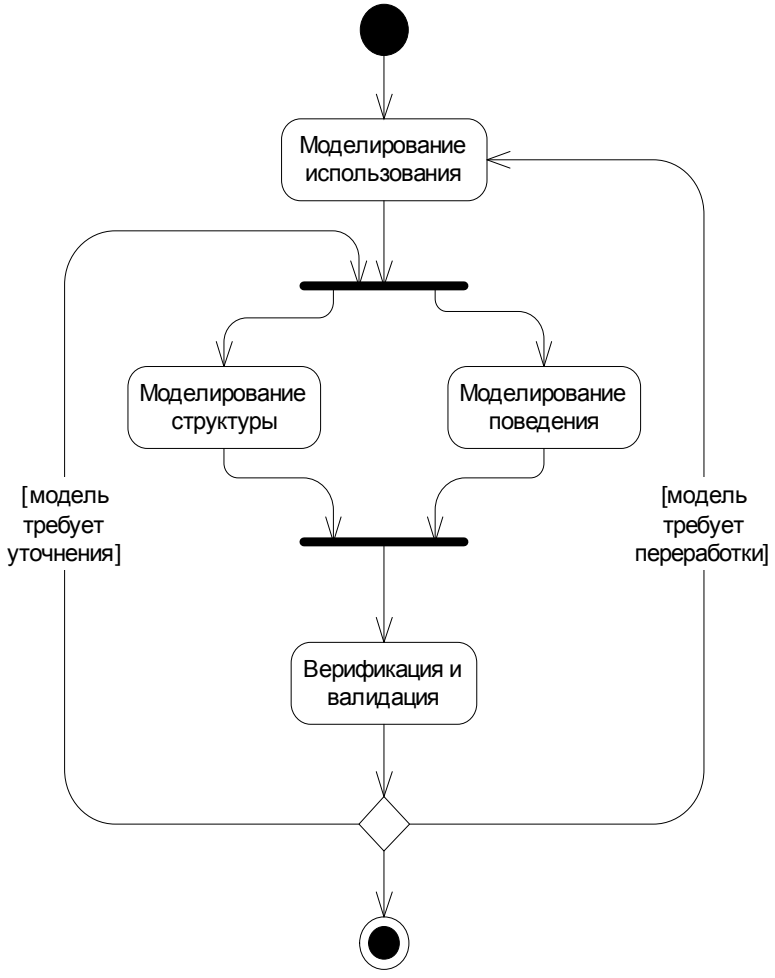

Рис. 5. Итеративный процесс визуального конструирования

Блоки деятельности на рис. 5 названы не совсем привычно, и это отражает убеждения автора: внедрение визуального конструирования окажет серьезное влияние на привычный процесс разработки. Место анализа требований займет моделирование использования, проектирование и реализация сольются в моделирование, отладка и

тестирование претерпят радикальные изменения, превратившись в верификацию и валидацию моделей. Но изменятся не только названия фаз процесса – это второстепенно. Важнее то, что кардинально изменится соотношение трудозатрат на разные фазы в процессе. Львиную долю будет занимать проектирование (в форме визуального конструирования), а непродуктивные сейчас процессы отладки, тестирования и документирования перестанут довлеть над менеджерами программных проектов, систематически недооценивающих трудозатраты при составлении планов-графиков.

Как же построить такой инструмент визуального конструирования? Как обычно – постепенно, итерационно (с помощью "инкрементального" процесса разработки). Для начала годится что угодно, но лучше сразу начать раскручивать инструмент, концептуально ориентированный на визуальное конструирование приложений, а не только на рисование диаграмм. Многие из "больших" инструментов визуального моделирования и проектирования, которые сейчас заявляют о своей поддержке *UML* 2.0, MDA и визуального конструирования, на самом деле изначально не были ориентированы на эти новые идеи. Конечно, эти инструменты прогрессируют в данном направлении, но лишь с той скоростью, с которой позволяет богатый (и тяжелый!) груз унаследованных компонентов.

Новые проекты выглядят динамичнее. Среди последних наибольшего внимания, по нашему мнению, заслуживает проект *UniMod* [18]. Автор наблюдает развитие проекта с близкого расстояния, и скорость роста возможностей этого инструмента впечатляет.

Проект *UniMod* опирается на ясную парадигму – автоматное программирование [19, 20]. В соответствии с парадигмой автоматного программирования приложение мыслится как совокупность некоторых объектов предметной области и управляющих ими конечных автоматов. Объекты управления могут выполнять некоторые действия и в них могут происходить события. Автоматы при поступлении событий вызывают соответствующие действия объектов. Эта простая и ясная парадигма является необычайно широко применимой. В данной статье автор не намерен отстаивать и разъяснять концепцию автоматного программирования (автору эта идея представляется бесспорной), отсылая читателя за блестящими аргументами и многочисленными примерами к первоисточникам [21]. Для нас достаточно того, что автоматное программирование удовлетворяет нашему главному критерию полной раскрутки, выдвинутому в первом разделе статьи. В качестве показательного примера, мы предлагаем вниманию читателя автоматную модель интерпретатора автоматных моделей.

На рис. 6 и 7 показана упрощенная модель интерпретатора моделей *UniMod*, реализованная на *UniMod*. В качестве исходного материала использовано описание работы интерпретатора *UniMod*, взятое из работы [22]. Для краткости здесь сделаны существенные, но не принципиальные упрощения: не рассматриваются составные состояния и вложенные автоматы (а значит, опущена проблема хранения контекста выполнения автомата), интерфейс объектов управления унифицирован (что нетрудно достичь на практике с помощью соответствующей оболочки) и объекты управления немедленно реагируют событиями на управляющие воздействия.

Концептуально модель приложения в *UniMod* состоит из диаграмм двух типов:

- специализированных диаграмм классов, показывающих участвующие в приложении источники событий, объекты управления и автоматы (такие диаграммы авторы *UniMod* называют диаграммами связей);
- диаграмм состояний, описывающих поведение автоматов.

На рис. 6 представлена диаграмма связей для нашего упрощенного интерпретатора моделей. Диаграмма выполнена с использованием стилистических соглашений *UniMod*: слева источники событий, справа объекты управления, а в центре автоматы. Имена выбираются в соответствии с очень жесткой дисциплиной имен, принятой в *UniMod*. Объекты полагается именовать "o1", "o2" и так далее; управляющие воздействия – "z0", "z1"; имена событий начинаются с буквы "e"; имена функций, участвующих в сторожевых условиях на переходах – с буквы "x".

Такая дисциплина имен кажется непривычной и даже архаичной, но у нее есть важные для визуального конструирования достоинства. Во-первых, имена получаются короткие, а значит, даже сложные выражения легко размещаются, например, на стрелках переходов диаграмм состояний. Во-вторых, имена типизированы на лексическом уровне, что очень важно для читабельности диаграмм. Такой стиль действительно противоречит якобы общепринятому правилу обязательного использования длинных содержательных идентификаторов. Заметим, однако, что содержательность легко достигается комментариями и всплывающими подсказками, и при этом не требуется использовать уродливых паллиативов наподобие "венгерской нотации".

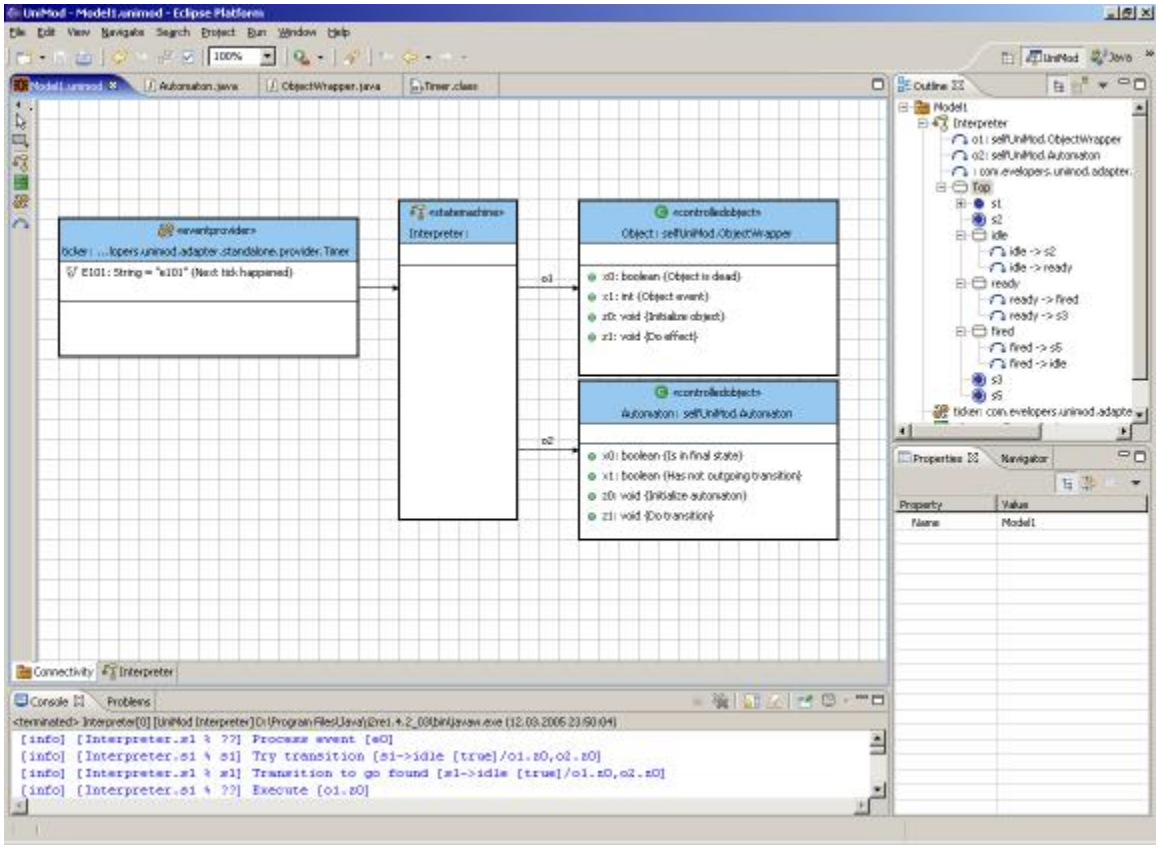

Рис. 6. Диаграмма связей интерпретатора

Структура связей интерпретатора предельно проста: интерпретатору нужен тактовый генератор, который бы побуждал его работать (на рис. 6 слева), и два объекта управления. Первый из них представляет объект или объекты предметной области (на рис. 6 обозначен o1), а второй – сам интерпретируемый автомат (на рис. 6 обозначен o2).

На рис. 7 представлена диаграмма состояний интерпретатора. Диаграмма сделана как можно более простой, с тем, чтобы еще раз показать на примере, что интерпретация моделей не является чем-то запредельно сложным.

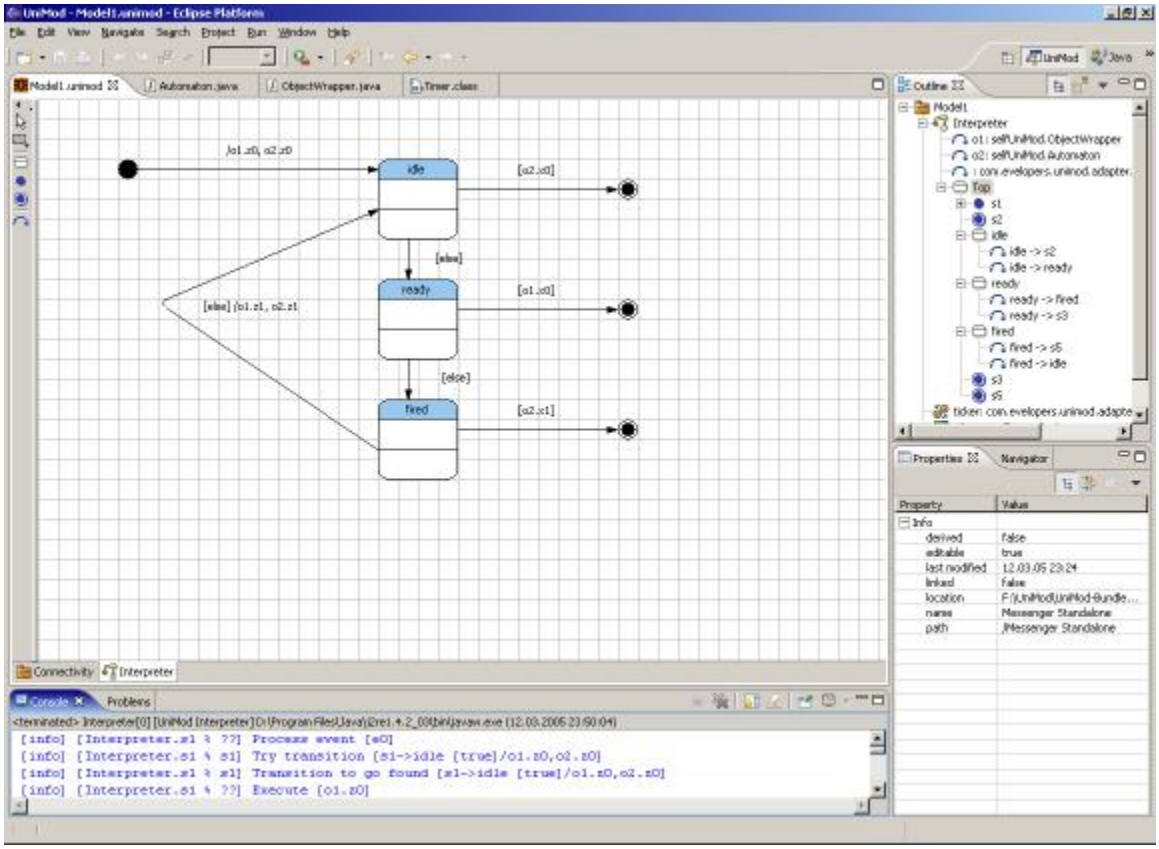

Рис. 7. Диаграмма состояний интерпретатора

Фактически, на этой диаграмме нет устойчивых состояний, поскольку все переходы спонтанны. Таким образом, данная диаграмма может быть прочитана как диаграмма деятельности, и в таком прочтении хорошо видна ее связь с первоисточником, заимствованным из [22].

Из приведенного примера видно, что для визуального конструирования совсем не обязательно "поддерживать" весь язык *UML*. Например, диаграммы связей *UniMod* являются весьма узким и к тому же специализированным подмножеством диаграмм классов *UML*. С другой стороны, если некоторая концепция поддерживается, то она должна поддерживаться в полной мере, например, так, как поддержаны диаграммы состояний в *UniMod*.

Заключение

Автор считает, что предшествующее изложение является достаточным основанием для следующего вывода.

Визуальное конструирование программ возможно, более того, оно реально осуществимо здесь и сейчас.

Помимо теоретических рассуждений и благих пожеланий имеются конкретные примеры инструментов, поддерживающих визуальное конструирование приложений. Самым показательным из известных примеров автор считает *UniMod*. Будучи с самого начала ориентированным на визуальное конструирование приложений,

данный инструмент сконцентрирован на главном: исполнимой визуальной модели приложения.

Статья подготовлена при поддержке «Лаборатории автоматного программирования», организованной корпорацией Borland и Санкт-Петербургским государственным университетом информационных технологий, механики и оптики.

Литература

1. **Брауде Э**. Технология разработки программного обеспечения. Питер, 2004.
2. http://www.omg.org/mda/
3. **Буч Г., Рамбо Д., Якобсон А**. *UML*: специальный справочник. Питер, 2002.
4. **Брукс Ф**. Мифический человеко-месяц. Символ, 2000.
5. **Якобсон А., Буч Г., Рамбо Дж**. Унифицированный процесс разработки программного обеспечения. Питер, 2002.
6. **Буч Г., Рамбо Д., Якобсон А**. *Я*зык *UML*. Руководство пользователя. ДМК, 2000.
7. OMG Unified Modeling Language Specification. Version 1.4. http://www.omg.org
8. http://www.uml.org/#UML2.0
9. **Фаулер М., Скотт К**. *UML*. Основы. 2-е издание. Символ, 2002.
10. **Thomas D**. *UML* – Unified or Universal Modeling Language? //Journal of Object Technology, 1, 2003.
11. **Андреев Н.Д**. Автоматическая верификация модели *UML* / IV Международная молодежная школа-семинар "БИКАМП-2003", СПбГПУ.
12. **Уоткинз Д., Хаммонд М., Эйбрамс Б**. Программирование на платформе .NET. Вильямс, 2003.
13. **Шалыто А.** Новая инициатива в программировании. Движение за открытую проектную документацию. Мир ПК. 2003, № 9, с.52-57. http://is.ifmo.ru/works/open_doc/
14. **Thomas D., Barry B**. Model Driven Development – The Case for Domain Oriented Programming. http://www.oopsla.org/oopsla2003/files/ddd-session-vision.html
15. http://www.omg.org/technology/documents/formal/xmi.htm
16. http://www.omg.org/technology/documents/formal/mof.htm
17. http://www.eclipse.org
18. **Гуров В.С. , Мазин М.А. , Нарвский А.С., Шалыто А.А**. *UML*. SWITCH-технология. Eclipse // Информационно-управляющие системы. 2004. № 6, с.12-17. http://is.ifmo.ru/works/*UML*-SWITCH-Eclipse.pdf

19. **Шалыто А.А**. SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука. 1998, 628 с. http://is.ifmo.ru/books/switch/1

20. **Шалыто А.А**. Автоматно-ориентированное программирование / Материалы IX Всероссийской конференции по проблемам науки и высшей школы «Фундаментальные исследования в технических университетах». СПбГПУ, 2005, с.44-52. http://is.ifmo.ru/works/_politeh.pdf

21. http://is.ifmo.ru/

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». Т.1, с.74-76. http://tm.ifmo.ru/tm2005/src/224as.pdf