Обзор реализации механизма циклической разработки диаграмм классов и программного кода в современных UML-средствах*

Н. А. Холтыгина Nadezhda.Kholtygina@lanit-tercom.ru
Д. В. Кознов dkoznov@yandex.ru

Механизм циклической разработки (round-trip engineering) является важной чертой модельно-ориентированного подхода к созданию ПО. Это обусловлено тем, что очень часто полная генерация кода по моделям невозможна, и сгенерированный код приходится дорабатывать «вручную». Циклическая разработка подразумевает, что связь сгенерированного и измененного кода с его моделями поддерживается в автоматическом режиме. В данной статье исследуются реализации механизма циклической разработки в ведущих UML-пакетах — Enterprise Architect (Sparx Systems), UModel (Altova), Rational Software Modeler/Architect (IBM). Рассматриваются диаграммы классов как самые востребованные на практике виды диаграмм UML. Предложена система критериев оценки, делаются выводы о применимости механизма циклической разработки в процессе создания промышленного ПО.

Введение

UML (Unified Modeling Language, унифицированный язык моделирования) — это язык для создания графических объектно-ориентированных описаний программного обеспечения, создаваемых при разработке и сопровождении программных продуктов [1].

*Работа выполнена при финансовой поддержке РФФИ (грант 11-01-00622а).

В связи с широким распространением UML было разработано большое количество программных средств для его поддержки. В настоящий момент существуют и используются около 30 UML-средств[1], почти во всех крупных средах разработки встроены собственные UML-редакторы.

UML-пакеты предоставляют не только возможность создания и редактирования UML-моделей, но также поддерживают генерацию программного кода по UML-моделям (прямая инженерия, forward engineering) и обратную процедуру — генерацию UML-моделей по существующему коду (обратная инженерия, reverse engineering). Важной компонентой UML-средств оказывается механизм циклической разработки (round trip engineering) — синхронизация двух или более программных артефактов, таких как исходный код, модели, конфигурационные файлы и пр. Необходимость в циклической разработке возникает в связи с тем, что часто полная генерация кода по моделям невозможна, и сгенерированный код приходится дорабатывать «вручную». Циклическая разработка подразумевает, что синхронизация сгенерированного и измененного кода и его моделей поддерживается в автоматическом режиме. Циклическую разработку часто ошибочно определяют как поддержку только механизмов прямой и обратной инженерии, однако, как отмечалось в [9], это не так. Отличительной характеристикой циклической разработки является распространение локальных изменений зависимых артефактов (например, UML-модели и программного кода) после изменений пользователем одного из них (change propagation), а не полная их перегенерация. Это важно, потому что обычно такие артефакты содержат, кроме общей части (так называемого семантического пересечения), еще и уникальную для каждого из них информацию — иначе не было бы надобности в различных артефактах! Важной задачей UML-пакетов является автоматизированная поддержка обновления артефактов после обнаружения несогласованности. Артефакты могут обновляются немедленно после обнаружения несогласованности, либо владелец артефакта может запускать процесс синхронизации и контролировать разрешение воз-

[1] См. список на сайте Object Management Group [2], список-рейтинг на UML Forum [6], а также списки, представленные компаниями Objects by Design [3] и Q-Success [4].

можных конфликтов по своему усмотрению. Наличие автоматической поддержки циклической разработки позволяет создателям ПО свободно переходить от UML-диаграмм к коду и обратно, не придерживаясь какой-то одной жесткой схемы работы.

На данный момент существует много различных обзоров UML-средств. В [5] представлена сравнительная характеристика UML-средств с оценкой различной функциональности. В [6] содержится подробный обзор наиболее популярных средств с оценкой (по шкале от 1 до 5), складывающейся из таких параметров, как удобство использования, соответствие стандарту и т. п. Работа [7] является русскоязычным обзором UML-средств с точки зрения наличия и качества генерации из UML в Java. В [8] рассматриваются средства прямого и обратного проектирования в различных UML-средствах. Однако отсутствует сравнительный обзор UML-пакетов с точки зрения поддержки циклической разработки, хотя некоторые обзоры для отдельных продуктов в этом направлении имеются — см., например, обзор механизма циклической разработки в продукте Rational Software Architect [10].

В данной статье представлен обзор поддержки механизма циклической разработки для случая UML-диаграмм классов и Java-кода. Диаграммы классов сильнее всего связаны с архитектурой ПО, поддержка их связи с программным кодом является, соответственно, основной задачей циклической разработки. В данной работе предлагаются критерии сравнения различных реализаций механизма циклической разработки и рассматриваются следующие UML-средства: Enterprise Architect (Sparx Systems), UModel (Altova), Rational Software Architect (IBM). Эти средства являются наиболее зрелыми UML-продуктами — согласно [6], они занимают второе, третье и четвертое место в рейтинге UML-средств. На основе данного анализа делаются выводы о готовности использования механизма циклической разработки в индустриальных проектах и о проблемах, которые требуют разрешения.

1. Критерии сравнения

Для сравнения реализаций механизма циклической разработки в различных UML-пакетах необходимо понять, как и по каким критериям проводить сравнение. Мы выделили следующие критерии:

- используемый механизм идентификации (mapping);
- поддержка простых изменений;
- поддержка сложных изменений;
- возможности настроек;
- удобство использования.

Рассмотрим эти критерии более подробно.

1.1. Механизмы идентификации

Одной из наиболее важных особенностей механизма циклической разработки является идентификация (mapping) — способ сопоставления элементов синхронизуемых моделей [11]. Способ идентификации важен, так как от него зависит качество реализации механизма циклической разработки — сколь тонкие изменения он может распознавать и распространять. Следуя [9], обозначим следующие способы идентификации.

1. По имени. Соответствие между элементами моделей устанавливается по именам. Недостатком данного механизма является то, что при переименовании элемента в одной модели связь между соответствующими элементами в другой теряется.
2. По уникальному идентификатору (по GUID). Элементы, как модели, так и кода, имеют некоторый идентификатор, в случае, если идентификаторы элемента модели и элемента кода совпадают, между ними устанавливается соответствие. Благодаря тому, что идентификатор не зависит от имени объекта, удается преодолеть недостаток предыдущего подхода. Кроме того, так как идентификатор обычно спрятан и не может быть изменен пользователем, удается достичь большей надежности идентификации. Таким образом, при изменении элементов модели можно однозначно определить, какие элементы связанной модели должны быть синхронизованы. Недостаток данного подхода заключается в том, что не всегда удается устано-

вить однозначное соответствие между элементами синхронизируемых моделей, например, если при копировании элемента модели копируется и идентификатор. Если же при копировании генерируется новый идентификатор, теряется связь между объектом и его копией. Кроме того, идентификаторы нужно хранить (в случае, если речь идет об элементах кода, это может представлять проблему), а также обновлять.

3. Вынесенная идентификация. В этом случае пары сущностей из модели и кода хранятся отдельно и от кода и от модели. В принципе, это могут быть не пары, а две группы — как с одной, так и с другой стороны может быть задана группа сущностей. Отметим, что речь идет о конкретных «живых» элементах модели и кода, а не об отношении на уровне метамодели и языка программирования.
4. По имени с различными эвристиками (дополнительный к [9] способ идентификации).

Не смотря на важность этого критерия, для существующих средств часто бывает не просто определить, какой способ идентификации они используют, так как эта информация является know-how.

1.2. Простые изменения

Здесь речь идет об операциях добавить/удалить/модифицировать элементы модели: классы, атрибуты классов, методы классов, параметры методов и т. п. — как в коде, так и в UML-модели. При этом модификация не означает смены имени и владельца, а только изменение атрибутов сущности, например, параметров операций. Для реализаций механизма циклической разработки важно, чтобы все эти простейшие операции работали надежно. В противном случае возможность пользоваться данным механизмом в промышленных проектах значительно снижается.

1.3. Сложные изменения

Авторы работы [13], рассматривая задачу нахождения разницы деревьев и представляя эту разницу в виде набора операций по изменениям одного дерева с тем, чтобы получить второе дерево, не ограничивались распознаванием простых операций — доба-

вить/удалить узел. Их алгоритм определял, были ли выполнены более сложные операции, как, например, перенесение ветви из одного поддерева в другое. Чем более семантически сложные операции распознает механизм синхронизации, тем более «тонко» он способен работать, в большей степени распространяя изменения и в меньшей степени перегенерируя заново фрагменты целевой спецификации. При перегенерации теряется та информация, которая в эти фрагменты была добавлена пользователем, но не содержится (и, соответственно, не синхронизирована) в исходной спецификацией. Мы выделяем следующие подобные операции.

- Переименование элементов. Важно, что при переименовании элемента в одной модели не потеряется связь с соответствующим ему элементом другой модели, иначе некоторая важная информация (например, реализация метода) может быть утрачена.
- Копирование элементов. Хотелось бы, чтобы при копировании элемента модели копировался не только элемент в данной модели, но и соответствующий ему элемент другой модели. Например, при копировании метода или класса в UML-диаграмме желательно, чтобы соответствующая реализация в коде также была скопирована. Естественно, копии в диаграмме и в коде должны быть синхронизованы между собой. Кроме того, после завершения процесса копирования, элемент и его копия должны существовать самостоятельно и не зависеть друг от друга.
- Перенос элементов (например, перемещение метода в другой класс). В данном случае важно, чтобы при перемещении элемента не потерялась связь с соответствующим ему элементом другой модели. Зачастую перемещенный элемент воспринимается не как уже существующий элемент, а как новый элемент, что приводит к потере связанной с ним информации.
- Изменение связей между элементами: добавление, удаление ассоциаций, изменение типа ассоциаций. Важно не только отражение внесенных изменений, но и сохранение каждой модели целостной и согласованной. Возможность получить некомпилируемый код после синхронизации делает механизм циклической разработки намного менее надежным.

1.4. Удобство использования

Сюда входит и простота инструмента в использовании, и возможность его тонкой настройки, и баланс между этими свойствами. Важно также, на сколько логика работы инструмента интуитивно понятна. Наконец, необходимо учитывать такие свойства, как надежность и хорошее качество реализации, отсутствие ошибок и сбоев.

2. Enterprise Architect

Данный продукт создан компанией Sparx Systems[2] — австралийской компанией, основанной в 1996 году. С самого начала своего существования компания занимается разработкой UML-продукта Enterprise Architect. Первый коммерческий выпуск продукта состоялся в 2000 году. Enterprise Architect является ведущим продуктом в данной компании (flagship product) и поддерживает кроме UML такие языки моделирования, как BPMN, SysML и др. Enterprise Architect используется для разработки ПО в самых разных прикладных областях — в аэрокосмической индустрии, при создании финансовых, банковских, медицинских систем, Web-приложений, исследовательских систем и т. д. Продукт используется как в крупных корпорациях, так и в небольших консалтинговых компаниях. Всего продано более 200 000 лицензий продукта. Enterprise Architect является одним из ведущих UML-пакетов (по рейтингу UML Forum он является вторым). Текущая версия продукта — восьмая. В данном обзоре мы рассматриваем Enterprise Architect 7.1.

Механизм слияния в Enterprise Architect реализован группой функций, доступ к которым можно получить из пункта главного меню Project/Source Code Engineering. Команда Import, подпункта Source Directory позволяет загрузить структуру классов из кода приложения. Классы могут быть созданы и непосредственно в Enterprise Architect. Для связи с Java нужно установить UML-проекта соответствующие опции (в частности, указать, что он является Java-проектом) и сгенерировать код по диаграммам с по-

[2] http://www.sparxsystems.com.au/

мощью команды Generate Package Source Code. Дальнейшие манипуляции с кодом и диаграммами синхронизируются с помощью команды Synchronize Package Contents: выбирается опция Forward Engineer (model–>source) для синхронизации изменений в модели с кодом или Reverse Engineer (source–>model) — для синхронизации изменений в коде или UML-моделью. Однако, если добавляются новые файлы в Java-проект, то для того, чтобы их содержимое попало в Enterprise Architect, нужно воспользоваться командой Import Java Files. Также при добавлении нового класса в Enterprise Architect он попадает в код, только если использована команда Generate Package Source Code. Тот факт, что оба этих действия вынесены из Synchronize Package Contents достаточно неудобен на практике. Никаких дополнительных опций у процедуры синхронизации нет.

2.1. Механизм идентификации

В качестве способа идентификации используется связь по имени в рамках файла с неизменным именем. При переименовании файла извне Enterprise Architect теряется связь содержимого этого файла с моделью классов в Enterprise Architect. То, что связь между кодом и моделью, в рамках этих ограничений, отслеживается именно по именам, следует из того факта, что при попытке переименовать метод или атрибут класса из кода/модели, соответствующие атрибут и метод в модели/коде удаляются и создаются заново.

2.2. Поддержка простых изменений

При добавлении атрибутов и методов классов, а также новых классов из Enterprise Architect все они соответственно появляются в программном коде. При добавлении атрибутов, методов классов и классов в коде они также появляются и в Enterprise Architect.

С удалениями ситуация немного иная. При удалении атрибутов и методов из кода при синхронизации они автоматически удаляются и в Enterprise Architect. А вот при удалении их из Enterprise Architect удаления в коде не происходит — соответствующие сущности в коде остаются. Такое поведение синхронизатора отчасти оправдано, так как не очень хорошо удалять сущности программы из модели — программа может перестать быть корректной, так

как в коде могут оставаться ссылки на удаленные сущности. Кроме того, пользователь Enterprise Architect не видит «тел» методов, а они могут быть весьма значительными, на их реализацию могло быть затрачено значительное время. Таким образом, методы и классы вполне целесообразно удалять только из кода. Хотя такое поведение синхронизатора оставляет модель и код несинхронизированными.

Отметим особо, что нам не удалось удалить класс в Enterprise Architect из кода — попытка это сделать не приводит ни к каким результатам, класс в модели остается, несмотря на то, что в коде его уже нет. Дополнительными возможностями Enterprise Architect, связанными с импортом файлов, также не удается достичь искомого результата — даже если в коде остается файл, из которого мы удалили класс (во многих случаях это не так — класс часто строго соответствует файлу, и при удалении класса удаляется также и файл), то при выполнении команды Import Java Files удаленный из кода класс продолжает существовать в модели.

Рассмотрим, как Enterprise Architect поддерживает модификацию элементов. Если у атрибута класса изменить тип и видимость, как из Enterprise Architect, так и из кода, все синхронизируется надлежащим образом. Если в методе класса изменить тип возвращаемого результата и его видимость (как из Enterprise Architect, так и из кода), то все также синхронизируется надлежащим образом. Однако при изменении в Enterprise Architect типа у какого-либо параметра метода, а также при добавление/удаление параметров во время синхронизации с кодом происходит создание нового метода с пустым телом в коде. То же действие, но из кода приводит к нормальной синхронизации. Специальное поведение синхронизатора при модификациях методов, выполненных из Enterprise Architect, можно объяснить заботой о корректности кода в той его части, которая не видна из UML — параметры могут использовать в «теле» метода, и если их изменить, то этот код может стать некорректным. При изменении имен параметров метода из кода/модели синхронизация проходит успешно.

2.3. Поддержка сложных изменений

Если мы переименовываем в коде атрибуты и методы, то все хорошо: Enterprise Architect просто удаляет их и генерирует новые. Тот факт, что происходит именно удаление, а не изменение, мы проверяли следующим образом: изменяли свойство атрибута в Enterprise Architect под названием constraints (оно не видно из кода) и при переименовании этого атрибута из кода наше значение этого свойства исчезло. Обратно, при переименовании атрибутов и методов из Enterprise Architect происходит дублирование — старые сущности из кода не удаляются, но добавляются еще и новые.

Переименование классов из кода также не работает нормально, так как синхронизатор этого не видит, что что-то было изменено не видит, а при загрузке нового класса из кода с помощью функции Import Java Files в модели оказывается два класса — старый и новый. При переименовании класса из модели синхронизатор, наконец, работает, но генерирует новый класс в тот же файл, где находится спецификация старого, а старый класс при этом сохраняет!

Копирование атрибутов и методов класса поддержано Enterprise Architect, но при этом в код добавляются новые сущности — у переносимых атрибутов в коде исчезают комментарии, а у методов — их «тела». То есть синхронизатор не фиксирует копирование, а воспринимает это действие как создание абсолютно новой сущности.

Перенос классов (вместе с файлами) в новый namespace (package) в коде не воспринимается в Enterprise Architect — прежние классы в UML-модели остаются там же, где они и были, а в новый namespace добавляются новые классы с такими же именам. Модель и код оказываются рассинхронизованными, а пользователь может менять в UML-модели старые файлы думая, что меняет новые.

2.4. Удобство использования

«Размазанность» функциональности синхронизации по нескольким командам доставляет неудобства при использовании и приводит к очевидным коллизиям, например, к невозможности удалить класс так, чтобы механизм синхронизации это учел.

История с переименованиями и удалениями, а также отсутствие поддержки переноса классов с файлами в новый namespace (package) сильно ограничивает использование механизма циклической разработки в Enterprise Architect и вызывает скептическое к нему отношение, может поставить под сомнение целесообразность использования его в индустриальном проекте.

3. UModel

Компания Altova была основана в 1992 году и имеет два юридических лица — в Австрии и в США. Компания является автором широко известного продукта XMLSpy, предназначенного для редактирования XML-спецификаций. Среди компаний, пользующихся услугами Altova, присутствуют такие компании, как Bank of America, Lufthansa, Motorola, Philips, Nokia и др. Первая версия продукта UModel была выпущена в 2005 году, последняя версия — в 2011 году. По рейтингу UML Forum это продукт является четвертым UML-средством. Мы исследовали версию UModel выпуска 2011 года.

Механизм слияния в UModel реализован группой функций, доступ к которым можно получить из пункта главного меню Project. Команда Import Source Directory позволяет загрузить в UModel структуру классов из кода Java-приложения. Классы могут быть созданы и непосредственно в UModel. Дальнейшие манипуляции с кодом и диаграммами синхронизируются с помощью команд Merge Program Code from UModel Project для синхронизации изменений в модели с кодом и Merge UModel Project from Program Code для синхронизации изменений в коде с UML-моделью. Эти команды и генерируют код, и «поднимают» новые файлы из Java в UModel, и проводят более мелкие изменения. Удобно, что вся эта функциональность не распределена между несколькими командами, как это сделано Enterprise Architect. При синхронизации можно выбрать следующие опции: при удалении кода его можно закомментировать или удалить (при генерации кода из модели); модель с кодом можно сливать (Merge) или перекрывать то, во что происходит синхронизация (Overwrite).

3.1. Механизм идентификации

В качестве способа идентификации в UModel используется связь по имени. Это становится ясным из того, что код приложений, сгенерированных по UML-моделям, не содержит никакой дополнительной информации, относящейся к UML-модели. Однако, в отличие от Enterprise Archtitect, UModel при идентификации использует многочисленные дополнительные эвристики, например, имя файла, в котором находится Java-спецификация класса. В целом идентификация оказывается качественной и справляется не только с типичными ситуациями, но и с различными сложными случаями. Нам не удалось найти изъянов в ее реализации, как впрочем и точно определить алгоритм ее работы.

3.2. Поддержка простых изменений

Все простые изменения модели классов и кода механизм синхронизации UModel поддерживает согласно ожиданиям, без сюрпризов (в отличие от Enterprise Architect). В частности, из UML-модели можно беспрепятственно удалять атрибуты и методы классов, а также сами классы. В последнем случае остается только имя файла, в котором располагалась спецификация класса в коде. Отметим также, что при добавлении класса в код он появляется в UML-модели, но не отображается ни на одной диаграмме, хотя мог бы добавляться на текущую, активную диаграмму, как это делается, например, в том же Enterprise Architect. Можно удалять классы из кода вместе с файлами — в модели эти изменения также будут отражены!

3.3. Поддержка сложных изменений

Переименование атрибутов, методов и имен параметров методов как из UModel, так и из кода прекрасно работает. При этом производится именно переименование, а не удаление и перегенерация. В частности, в сторону кода это легко понять, если, например, в середину определения атрибута вставить комментарий (между типом и именем). При переименовании в UModel данного атрибута с

последующей синхронизацией в коде этот комментарий останется, а атрибут будет переименован.

При переименовании класса в коде (вместе с конструкторами) в UModel все переименовывается после синхронизации. Обратно тоже все работает, включая автоматическое переименование конструкторов в UML-модели. Если в коде переименовать класс и имя файла, в котором он содержится, то при синхронизации UModel предлагает связать новый класс с прежним. Данная связь может строиться в режиме слияния двух классов (опция Merge) или в режиме переписывания старого класса новым (опция Overwrite).

В UModel есть возможность «перетащить» атрибут или метод из одного класса в другой. В последнем случае «тело» метода не перетаскивается, также как и в Enterprise Architect: в коде создается пустой метод. То же самое происходит, если в UModel скопировать класс — все его методы в коде будут пустыми. В UModel не поддерживается «перевешивание» ассоциаций. Если «перевесить» ассоциацию в коде, то после синхронизации UModel создаст в модели классов две новые однонаправленные ассоциации.

3.4. Удобство использования

Механизм синхронизации UModel работает предсказуемо, надежно, он прост в использовании и в настройке.

4. Rational Software Architect

Продукт Rational Software Architect является наследником знаменитого UML-пакета Rational Rose, в разработке которого участвовал Грэди Буч — один из основоположников UML — и который в конце 90-х ним начале 2000 годов был одним из самых покупаемых UML-продуктов. На базе UML в компании Rational Software Corp. был создан процесс разработки RUP (Rational Unified Process) [14] и собрана линейка продуктов, поддерживающих, фактически, весь жизненный цикл разработки ПО — от сбора и формализации требований до конфигурационного управления и тестирования.

В настоящее время компания IBM, купившая Rational Rose вместе с Rational Software Corp., продвигает несколько родственных

средств моделирования — Rational Software Modeler, Rhapsody, ErWin и др. Продукт Rational Software Architect отличается тем, что кроме поддержки UML он включает в себя также Java-IDE[3]. Специалисты IBM считают, что программистам трудно использовать что-либо, находящееся вне их IDE, в том числе и средства визуального моделирования. Поэтому вместе в UML-средством IBM предоставляет IDE.

Это IDE имеет браузер проекта — общий как для Java-IDE, так и для UML-модели, при этом и проект в Rational Software Architect для обоих частей создается общий. Имеются также отладчик, хороший рефакторинг, Java-компилятор и пр. В IDE хранится история изменений Java-файла и есть встроенный компаратор.

Rational Software Architect глубоко интегрирован с многочисленными продуктами IBM, поддерживающими различные аспекты разработки ПО (в первую очередь, с линейкой продуктов бывшей компании Rational Software Corp.). Rational Software Architect является мощной средой разработки и может быть по-разному интегрирован в процесс разработки в зависимости от потребностей. Тот факт, что он создан на базе Eclipse и глубоко интегрирован еще и туда, несомненно добавляет ему новые возможности и новых пользователей.

В данной работе мы рассматриваем IBM Rational Software Architect 8.0.

Связи UML-моделей с программным кодом в Rational Software Architect определяются только для диаграмм классов UML и задаются в трансформациях. Трансформация может быть создана как UML-to-Java или как Java-to-UML. То, как трансформация создана, считается в дальнейшем ее основной функциональностью. Обратный вариант трансформации также поддерживается, но управляется выбором одной из следующих трех опций: Reconciled (есть синхронизация), Conceptual (есть генерация и синхронизация, не только в одну, главную сторону), Mixed (смешанный вариант, определен только для UML-to-Java, позволяет синхронизировать не все классы, а лишь некоторые, считая, что остальные классы в UML-модели

[3]Integrated Development Environment — общее название интегрированных сред разработки ПО.

являются абстрактными, концептуальными и не имеют проекций в код).

В трансформации можно задать также особенности проекций UML в Java — варианты связей ассоциации и атрибуты UML с кодом. В трансформации определяется связывание UML-проекта и Java-проекта, а также используемые коллекции при генерации множественных ассоциаций Java-коллекции и многое другое. Имеется специальный мастер по созданию трансформаций. В одном проекте можно задать сколько угодно трансформаций. Они сохраняются в специальных файлах, которые можно сохранять как в модели, так и в Java-части проекта (код находится прямо в модели в специальном подпроекте). Доступные трансформации становятся видны при активации контекстного меню как на отдельных классах, так и на самом проекте в браузере проекта, и содержатся в пункте контекстного меню transform. Можно запускать их на отдельном классе как из модели, так и из Java-кода. В любом случае при активизации конкретной трансформации отображаются две команды — Run Java to UML и Run UML to Java, которые, как указывалось выше, занимаются как синхронизацией, так и генерацией и возвратным проектированием.

4.1. Механизм идентификации

Тот факт, что Java-код находится в одном проекте с моделью, открывает дополнительные возможности. Например, становится возможным решать задачу синхронизации параллельных изменений модели и кода в виде задачи слияния (Merge), как это описано в [10]. Кроме того, это позволяет осуществить возможность задания прямого mapping между разными элементами кода и моделей, явно указав, какие сущности с одной стороны соответствуют сущностям с другой стороны. Например, класс `A1` в модели соответствуют классу `A2` в коде. И далее при синхронизации это соответствие будет учитываться. К сожалению, нам не удалось «опустить» этот механизм внутрь классов и поставить в соответствие, например, атрибуту в модели пару функций в коде класса (`Get` и `Set`). Вся идентификация в Rational Software Architect осуществляется по именам, и она существенно более слабая, чем в UModel, как следует из экспериментов, представленных далее.

4.2. Поддержка простых изменений

Добавление/удаление атрибутов и методов из кода и из Rational Software Architect работает надлежащим образом. Хуже обстоит дело с классами. Новые классы добавляются из модели и из кода нормально, а вот синхронизировать удаление класса (как из кода, так и из модели) не получается.

Модификация атрибутов из кода и из модели (изменение видимости, типа, значения инициализации и т. д.) работает нормально. С методами классов дело обстоит следующим образом. Модификация возвращаемого типа, типов параметров, видимости метода синхронизируется нормально. Но если добавить новый параметр — из кода или из модели — то генерируется новый метод, а старый остается. При этом очевидно, что согласованность модели и кода нарушается.

4.3. Поддержка сложных изменений

Переименование атрибутов и методов при синхронизации приводит к добавлению новой сущности, и после этого и модель, и код перестают быть согласованными. При переименовании класса из кода синхронизатор сообщает, что не нашел никаких изменений. Если переименовать еще и файл, содержащий класс, то синхронизатор предлагает добавить новый класс в модель. При этом старый класс остается, и модель перестает быть согласованной с кодом. При переименовании класса из модели при синхронизации в код добавляется новый класс, и модель снова перестает быть согласованной с кодом.

Есть еще возможность при синхронизации разноименных классов воспользоваться специальной идентификацией, принудительно прописав, что пара классов в коде и в модели с разными именами соответствуют друг другу. При этом все изменения в одном классе при синхронизации будут попадать в другой и наоборот.

Копирование, перетаскивание атрибутов, методов, классов работает так же, как в UModel.

4.4. Удобство использования

Rational Software Architect обладает дружественным и красивым пользовательским интерфейсом. Однако качество механизма циклической разработки, существенно ниже, чем, например, в UModel.

При синхронизации пользователю задается много вопросов, и если ошибиться с ответами — не поставить нужные галочки, то модель и код быстро приходят в противоречивое состояние, а синхронизатор начинает выполнять непонятные вещи, например, все время добавлять один и тот же конструктор в один класс (и там становится много-много одинаковых конструкторов!). Кроме того, после многих действий синхронизатора модель и код оказываются в рассогласованном состоянии. Использовать механизм синхронизации в реальном индустриальном проекте, по всей видимости, пока еще нельзя — как из-за качества (слишком много ошибок), так и из-за сложной логики синхронизации, отягощенной еще возможностью сложных проекций: например, атрибут в модели может переводиться в пару процедур `Get_имя_атрибута/Set_имя_атрибута`, и такие случаи порождают огромное количество возможных, в том числе и непредсказуемых вариантов при синхронизации, в особенности, при различных некорректных действиях пользователя (например, при изменении имени операции `Get`!).

5. Анализ и выводы

Для того чтобы систематизировать и сравнить полученные результаты анализа, они были собраны воедино в общую таблицу (табл. 1).

На основании данных таблицы можно сказать, что из рассмотренных средств UModel наиболее подходит для промышленной разработки. Его реализация механизма циклической разработки надежна, удовлетворяет интуитивным представлениям о том, как должен работать данный инструментарий, и данный пакет поддерживает наибольшее количество операций синхронизации. В остальных средствах механизм циклической разработки является скорее «демо-функциональностью», быстро приводит модель и код в рас-

Сравнение реализации механизма циклической разработки в разных UML-средствах

UML-пакеты / Критерии	Уточнения критериев	IBM Rational Software Architect	Enterprise Architect	UModel
Идентификация	Качество	3	3	5
Простые изменения	Качество	3	2	5
Сложные изменения	Качество	3	3	4
Удобство использования	Удобство интерфейса	3	3	4
	«Богатство» настроек	5	2	2
	Интуитивная понятность функциональности	1	2	5
Итого		17	15	25

согласованное состояние, сложен и неочевиден в использовании. Наличие сложных проекций модели в код, забота о безопасности кода, плохо реализованная идентификация — вот основные причины проблем при реализации механизма циклической разработки.

Список литературы

[1] UML 2.0 Infrastructure Specification, September, 2004, http://www.omg.org/

[2] http://uml-directory.omg.org/vendor/list.htm

[3] http://www.objectsbydesign.com/tools/umltools_byCompany.html

[4] http://www.software-pointers.com/en-tools-uml.html

[5] http://en.wikipedia.org/wiki/Comparison_of_Unified_Modeling_Language_tools

[6] http://www.uml-forum.com/tools.htm

[7] http://www.uml2.ru/forum/index.php?topic=430.0

[8] *Boklund A., Mankefors-Christiernin S., Johansson C., Lindell H.* A Comarative Study of Forward and Reverse Engineering in UML Tools In. Proceedings APPLIED COMPUTING 2007, Salamanca, Spain, 18–20 February 2007.

[9] *Sendall S., Kaster J.* Taming Model Round-Trip Engineering // Proceedings of Workshop on Best Practices for Model-Driven Software Development, 2004.

[10] *Liu C.* Round Trip Engineering Scenario Using Rational Software Architect and ClearCase Remote Client.
http://www.ibm.com/developerworks/rational/library/10/

[11] *Shvaiko P., Euzenat J.* A Survey of Schema-based Matching Approaches. Journal on Data Semantics IV, 2005. P. 146–171.

[12] *Sudarshan S. Chawathe, Anand Rajaraman, Garcia-Molina H., Widom J.* Change Detection in Hierarchically Structured Information // ACM SIGMOD International Conference on Management of Data (SIGMOD). 1996. P. 493–504.

[13] *Sudarsah S. Chawathe, Garcia-Molina H.* Meaningful Change Detection in Structured Data // Proceedings of the ACM SIGMOD International Conference on Management of Data. ACM Press. 1997. P. 26–37.

[14] *Кролл П., Крачтен Ф.* Rational Unified Process — это легко. Руководство по RUP для практиков. Изд-во КУДИЦ-Образ, 2004. 432 с.