УДК 681.3

Д.М. Рябко

ПОДХОД К РЕАЛИЗАЦИИ СРЕДЫ РАЗРАБОТКИ ДЛЯ DSL

Рассмотрены вопросы, касающиеся методологии разработки DSL, предложены подходы к созданию среды разработки на основе DSL, ее архитектура и способы реализации. Исследованы ключевые особенности построения отдельных инструментов, а также подсуммированы основные достоинства и недостатки MDD подхода.

Введение

Изучив опыт разработки программного обеспечения (ПО) на протяжении последних десятилетий, можно сделать вывод, что в этой области прогресс двигается в направлении повышения уровня автоматизации, что в свою очередь, предъявляет более строгие требования к качеству обслуживания. Это ведет к усложнению архитектуры. Мы можем наблюдать, как глобальные коммуникации, позволяя заказчикам и партнерам более тесно участвовать в бизнес-процессах, требуют более частого обновления приложений и их версий. В индустрии разработки ПО имеется ряд проблем, которые определяются методами программирования и могут быть представлены таким образом:

- сложность систем;
- особенность предметной области;
- формализация разработки;
- накопленный опыт;
- наглядность;
- гибкость;
- верифицируемость;
- повторное использование.

Эти проблемы являются симптомами других, более низкоуровневых проблем, отражающих недостатки в существующем способе построения ПО. Данные проблемы подробно рассмотрены и проанализированы в [1].

В данной работе рассматриваются достоинства и недостатки новой методологии в программировании, известной как Model Driven Development (MDD, разработка управляемая моделями). Цель этой методологии в создании ПО предметной области из моделей домена. Для описания моделей домена используется Domain Specific Language (DSL – язык описания специфики предметной области).

В работе излагается методология MDD. Рассматриваются вопросы реализации архитектуры среды разработки, способы построения инструментов для удобного использования языков типа DSL, варианты редакторов, валидации моделей с помощью ограничений, преобразования моделей, генерации кода и интеграции его с исходным кодом.

Особенности языков DSL в MDD

Методология MDD включает:

– модельно-управляемая архитектура (Model-Driven Architecture, MDA). Проектируемые системы представляются с использованием языка моделирования общего назначения Unified Modeling Language (UML) и его конкретных профилей. Эти модели преобразуются в артефакты, выполняемые на разнообразных платформах;

– модельно-интегрированные вычисления (Model-Integrated Computing, MIC). Для представления элементов системы и их связей используются предметно-ориентированные языки моделирования DSML, а также их преобразования в платформенно-зависимые артефакты.

В свою очередь MIC объединяет:

– предметно-ориентированные языки моделирования DSML (Domain-Specific Modeling Language), в системах типов которых формализуется структура, поведения и требования приложения внутри соответствующей предметной области. В используемых метамоделях определяются связи между понятиями предметной области, точно специфицируется основная

семантика и ограничения, ассоциируемые с этими понятиями;

– трансформационные процессоры и генераторы, которые анализируют определенные аспекты моделей и синтезируют исходный код, входные данные для имитационного моделирования, XML-описания развертывания или альтернативные представления моделей. Возможность синтеза на основе моделей помогает поддерживать согласованность между реализациями и аналитической информацией о зафиксированных в модели требованиях к функциональным возможностям системы и ее качеству.

Сами по себе модели бесполезны в конечном приложении. Модели должны быть преобразованы в исполнимый код для конкретной платформы. Такие преобразования осуществляются с помощью преобразования моделей. Модель преобразуется в другую модель с понижением абстракции, т.е. в модель более конкретную и менее абстрактную. Серия таких преобразований в конечном итоге приводит к исполнимому коду и последнее преобразование будет являться преобразованием модель-код.

Характеристика языка DSL

Как и любой другой язык, DSL состоит из трех частей:

– мета модель (также известна как абстрактный синтаксис) определяет строительные составляющие языка и правила, по которым они должны собираться, чтобы образовывать правильные модели;

– конкретный синтаксис служит для описания нотаций, используемых для описания моделей. Мета модель может иметь несколько конкретных синтаксисов. Которые могут быть текстовыми, где предложения записываются в виде *спецификаций* и графическим – в виде *моделей*;

– семантический смысл моделей должен быть хорошо определен.

Использование языка DSL позволяет решить три основные проблемы, присущие языкам GPL (General Purpose Language, язык общего назначения), используемых в объектно-ориентированом программировании (ООП):

– большой временной разрыв, между реализацией идей и их воплощением;

– понимание существующего кода;

– расширение языка.

Рассмотрим каждую из них.

Проблема 1 заключается в том, что основная временная задержка между формулировкой задачи и ее воплощением находится между составлением модели программной системы и ее реализацией на GPL. Можно рассмотреть такой пример: программисты объясняют модель программной системы друг другу в считанные часы, но на реализацию этой системы каждый из них потратит дни. Это происходит потому, что для описания модели программисты используют специальный язык, который содержит специфичные термины и формулировки. Для реализации модели на компьютере программисты используют GPL, который содержат в себе несколько десятков понятий, используя которые можно описывать модель. Естественные языки содержат десятки тысяч понятий, с помощью которой модель можно кратко описать (рис. 1). Поэтому чтобы объяснить поставленную задачу компьютеру, необходимо ее представить более детально.

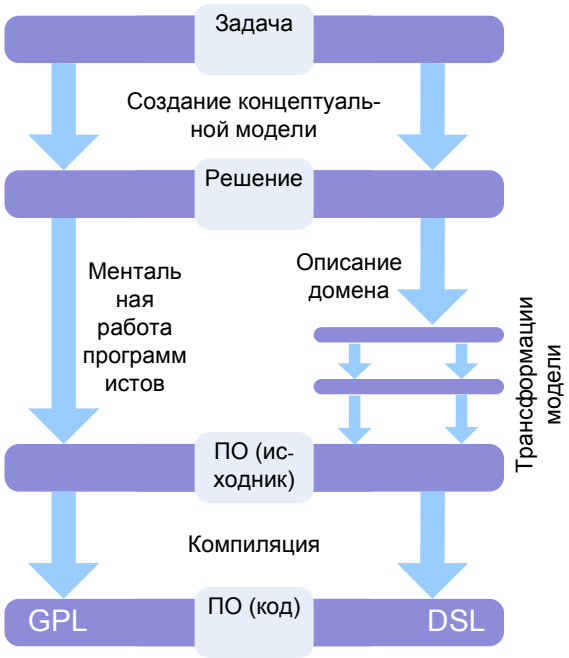

Рис. 1. Сравнение разработки приложения с использованием языка GPL и DSL

Большую часть времени программисты тратят на поиск способа выражения абстракций модели в языке GPL, используя объектно-ориентированный дизайн (OOD). Используя методологию MDD, OOD не применяется.

Проблема 2 возникает, при попытке понять, что было сделано ранее для решения определенной задачи. Процесс понимания кода заключается в обратной трансформации абстракций GPL в абстракции высокоуровневой модели, т.е. восстановление информации, которая была потеряна в процессе кодирования.

Традиционный способ решения данной проблемы это комментарии, документация, дизайн модели и т.д. Но это довольно посредственное решение поскольку оно имеет ряд слабых сторон, таких как дополнительные затраты на написание документации, необходимость постоянной синхронизации и обновления написанной документации и т.д. В идеале код сам по себе должен быть и документацией, которая позволяет понять данный код.

Проблема 3 заключается в том, что библиотеки языка не выражаются в понятиях доменной модели, а выражаются в низкоуровневых понятиях GPL, таких как классы и методы. Чтобы использовать какие-то библиотеки программист должен изучить их, изучить способ преобразования модели его задачи в решение на уровне библиотеки. Относительно простые понятия доменной модели требуют усердной предварительной работы, чтобы интерфейсы библиотеки были использованы правильно.

Подсуммируем преимущества разработки с использованием DSL.

– DSL позволяет выразить решение в терминах предметной области на соответствующем уровне абстракции. Следовательно, специалисты в данной предметной области могут разбирать, верифицировать, изменять и разрабатывать DSL-программы. Что позволит сместить акцент в программировании в сторону специалистов в конкретной области, т.е. не специалистов в программировании. Что позволит избежать потери информации или ее искажение при передачи от специалиста доменной области программисту.

– DSL-программы лаконичные, во многом самодокументирующиеся и могут повторно использоваться для различных целей. Использование в конструкциях языка понятий предметной области позволит в дальнейшем читать программу также легко, как-будто она была изложена на разговорном языке.

– DSL повышает эффективность, надёжность, переносимость и качество сопровождения. Операции, осуществляемые на уровне моделей, гораздо эффективнее, намного легче и подвержены меньшему количеству ошибок, чем те же операции, осуществляемые на уровне исходного кода.

– DSL содержит знания о предметной области, обеспечивая таким образом возможность их хранения и повторного использования.

– DSL позволяет проводить валидацию и оптимизацию на уровне абстракции, соответствующем предметной области. Валидация модели, используя предикаты на каждом уровне абстракции, позволяет избежать изнурительного тестирования на поздних этапах разработки.

– DSL улучшает тестируемость ПО и позволяет автоматизировать этот процесс. Вместо разработки сценариев тестирования и ручного набора данных, например, мы можем сгенерировать и применить их на основе декларативной информации, зафиксированной в моделях. Можно также конфигурировать и контролировать тесты, а также наблюдать за их выполнением, используя модели. Производя отладку на уровне моделей, пользователь может контролировать систему, применяя для этого выражения на языке моделирования.

– Модели, описанные с помощью одного DSL, могут быть трансформированы в модели описанные с помощью другого DSL. Это позволяет свободно интегрировать между собой различные части системы, написанные на разных DSL.

– Предметная область может быть описана на одном уровне абстракции, а

затем преобразована с дополнительно детализацией на более низкие уровни абстракции, что позволяет дополнять модель на разных этапах разработки.

– Возможность автоматизировать реструктуризацию, сборку и развертывание. Реструктуризация может быть автоматизирована, используя высокоточные доменно-специфичные модели, настроенные специально на язык программирования. Модели могут содержать информацию о сборке, включая артефакты, участвующие в ней, а также их зависимости. Они также могут содержать информацию о развертывании, включая конфигурацию развертываемых исполняемых программ, объемы и конфигурации аппаратных и программных ресурсов, необходимых исполняемым программам.

К недостаткам применения DSL можно отнести:

– высокую стоимость проектирования, реализации и сопровождения;

– необходимость начального обучения пользователей;

– отсутствие доступных реализаций различных вариантов DSL;

– сложность определения области действия DSL;

– необходимость сохранения равновесия между конструкциями, специфичными для предметной области, и конструкциями языков программирования общего назначения;

– возможное снижение эффективности по сравнению с программным обеспечением, написанном на языке программирования общего назначения.

Методология разработки DSL

Разработка языка, зависящего от предметной области, как правило, включает следующие этапы жизненного цикла (рис. 2).

Этап анализа:

– определение предметной области;

– сбор всех релевантных знаний предметной области;

– построение системы знаний в виде нескольких семантических нотаций и операций над ними;

– проектирование DSL, лаконично описывающего приложение в предметной области;

– подготовка правил, для последовательных преобразований моделей;

– определение предикатов, проверяющих семантику моделей.

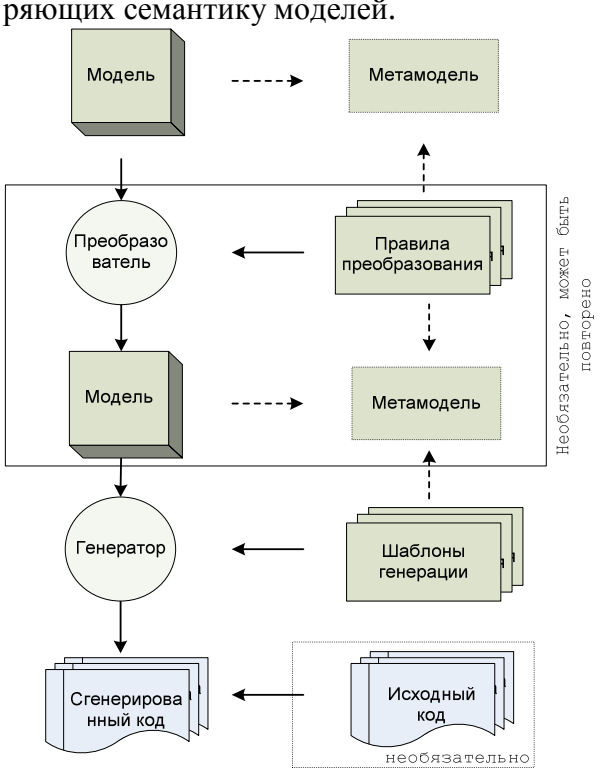

Рис. 2. Этапы методологии DSL

Этап реализация:

– изготовление индивидуального редактора, реализующего семантику языка;

– создание библиотеки, реализующей семантические верификации;

– проектирование и реализация генератора для выполнения последовательных трансформаций моделей, в конечном итоге приводящей к последовательности библиотечных вызовов.

Этап применения:

– описание модели с помощью спроектированного DSL;

– выполнение преобразований моделей и интеграция их на каждом этапе;

– генерация кода и интеграция кода с рукописным кодом.

Архитектура среды разработки MDD

В результате анализа методологии MDD предложена модель архитектуры

среды разработки для описания мета моделей и моделей, составления конкретных DSL, разработки графических и текстовых редакторов, модификации моделей и генерации кода (рис. 3). На рис. 3 используются такие цифровые обозначе¬ния:

1) проверка семантики модели, используя предикаты валидации, на соответствие метамодели предметной области. Валидация может быть реализована в реальном времени, так как она реализована в средах разработки при проверке правильности кода;

2) преобразование модель-модель позволяют свободно интегрировать между собой различные части системы, а также преобразовывать модель на разные уровни абстракции;

3) дополнение и завершение модели. Добавление новых артефактов в модель или завершение существующих;

4) загрузка и сохранение модели. Модель должна хранится в каком-то представлении в системе. Это может быть сериализация во внутренний формат или поддержка международных стандартов хранения моделей, например xmi;

5) создание индивидуального графического редактора модели на основе метамодели;

6) редактирование модели с помощью текстового редактора;

7) генерация кода на основе шаблонов;

8) интеграция сгенерированного кода с рукописным кодом.

Рассмотрим основные блоки архитектуры более детально.

Мета модель может быть спроектирована в любом графическом (ER-диаграммы) или текстовом (спецификации) редакторе. Используя редактор UML также можно определить мета модель. Далее мета модель может быть сохранена в стандартном формате XMI и загружена в любой среде разработки для дальнейшей обработки. Например, модель может быть описана с помощью инструментов UML, или если рассматривать open source IDE Eclipse, то с помощью Eclipse Modeling Framework (EMF).

Индивидуальный графический редактор – одна из самых сложных частей в архитектуре среды разработки MDD. Из наиболее известных редакторов можно выделить:

– Eclipse GMF (Graphical Modeling Framework);

– Microsoft DSL Tools;

– Meta Programming System (MPS) от JetBrains;

Рассмотрим создание редактора в каждой из сред разработки более детально.

В **GMF** предполагается, что мета модель была описана или загружена с ис-

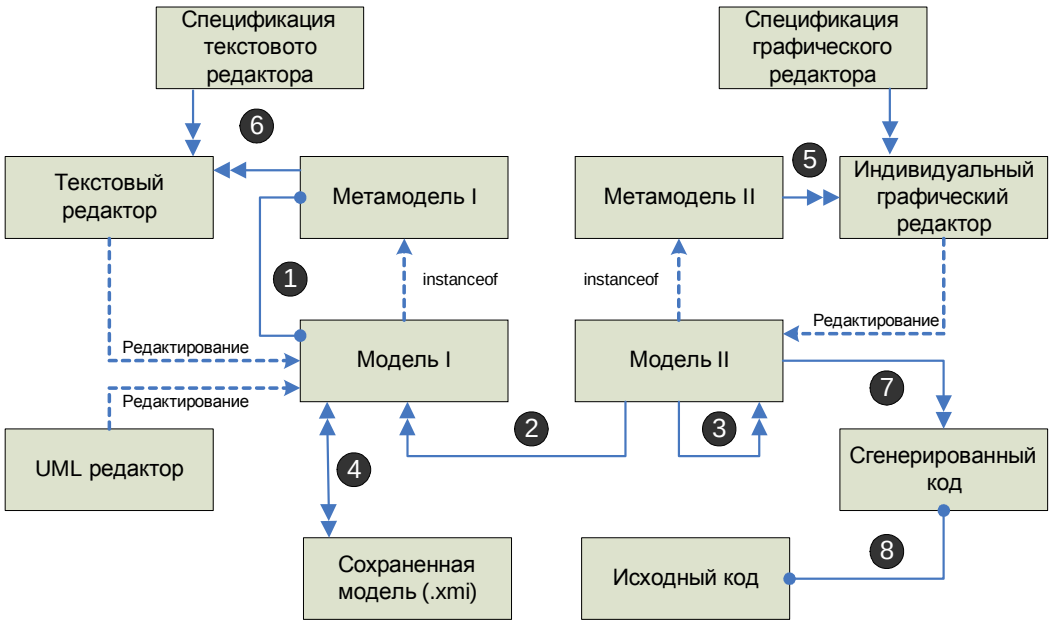

Рис. 3. Архитектура среды разработки MDD

пользованием EMF. Далее из этой метамодели генерируется набор артефактов. Во-первых, генерируется genmodel. Это в основном модель-декоратор (или модель-маркер) вокруг метамодели, которая «декорирует» набор дополнительных свойств. Далее определяются такие модели [5]:

– определяющая графическое представление, включая формы, декорации, узлы и связи, она называется gmfgraph;

– модель для палитры редактора и других инструментов, называется gmftool;

– модель отображения для связи этих двух моделей с мета моделью.

На рис. 4 показано как эти все модели связаны вместе. Для всех добавочных моделей GMF создает gmfgen модель – «низко-уровневую» модель, которую генератор кода использует как входную, окончательно производя diagram проект, который и будет содержать требуемый графический редактор.

В **DSL Tools** инструменте индивидуальный графический редактор входит в состав индивидуального дизайнера, создаваемого под конкретную доменную модель [7]. Дизайнер создания DSL требует выбора базового шаблона, состоящего из:

– минимального языка – простой шаблон, создающий очень маленький базовый язык, включающий только два объектных понятия и нотацию содержащую одно поле и одну линию;

– диаграммы активности – шаблон демонстрирующий диаграмму активности UML;

– диаграммы классов – шаблон демонстрирующий диаграммы классов UML;

– диаграммы вариантов использования – шаблон демонстрирующий диаграммы вариантов использования UML.

При построении нового дизайнера пользователь может контролировать вид будущего DSL. Формы, иконки и свойства компонентов могут быть настроены или добавлены новые. Все измененные ресурсы будут включены в дизайнер при компиляции (рис. 5). После создания доменной модели и дизайнера могут быть добавлены шаблоны кода для усовершенствования и настройки DSL. Эти шаблоны будут использоваться для генерации кода из экземпляров модели, которые пользова-

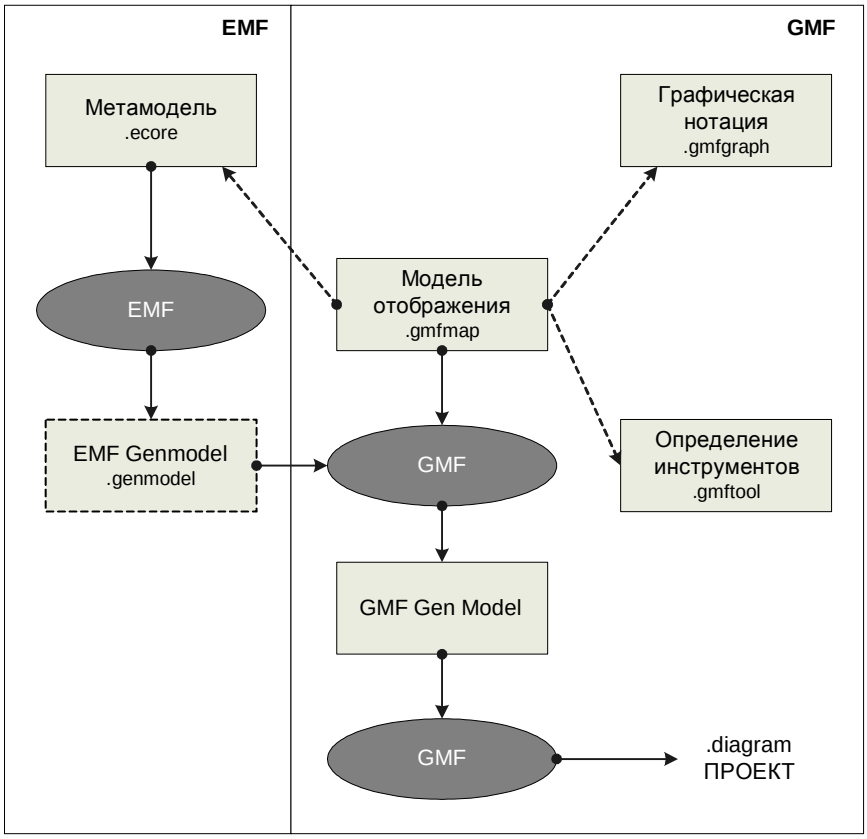

Рис. 4. Создание индивидуального графического редактора в GMF

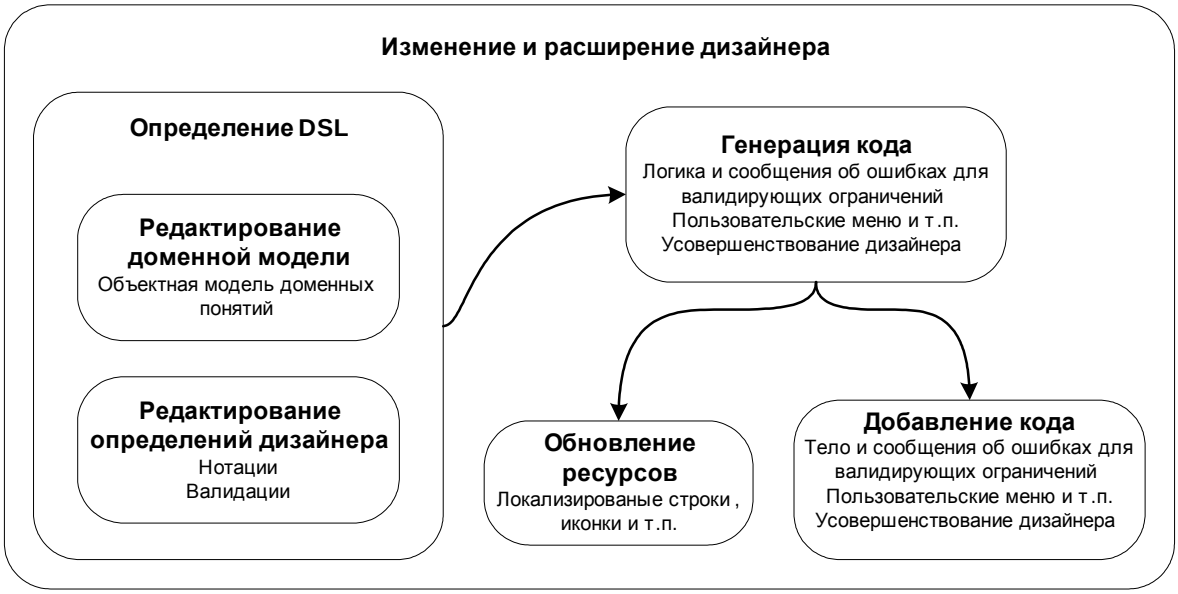

Рис. 5. Процесс создания нового дизайнера в MS DSL Tools

тель создаст. После добавления всех аспектов дизайнер компилируется и может быть установлен на Visual Studio.NET 2005, где может быть использован для генерации артефактов и шаблонов.

Разработчики **MPS** не добавили в свою среду разработки возможность создания индивидуального графического редактора, вместо которого предложили другой вид редактора. Этот редактор языков использует свой собственный язык Editor Language. Поле редактора разделено на прямоугольные ячейки, которые могут содержать ключевые слова, скобки, разделители и т.д. Каждая ячейка может содержать в себе другие ячейки. Использование ячеек имеет ряд интересных преимуществ. Во-первых, это позволяет редактору имитировать текстовый редактор, работая напрямую с графовой структурой программы. Во-вторых, в ячейки можно вводить не только текст, но и графики, таблицы и т.д. К тому же ячеечная структура это не обязательный атрибут, программист может создать что-то свое. Таким образом Editor Language позволяет создать ячеечную структуру для каждого понятия в языке. Можно определить какие понятия постоянные, например, как скобки, а какие переменные и пользователь должен определять их. Editor Language позволяет добавить такие возможности, как рефакторинг, автозаполнение, подсветка синтаксиса, подсветка ошибок и т.д.

Ограничения – это правила, которым должна удовлетворять модель, чтобы быть правильной. Формально ограничения – часть метамодели. Ограничения это булевские выражения (предикаты), которые истинны в случае, когда модель удовлетворяет метамодели.

Проверка ограничений должна быть доступна как в процессе выполнения модели, так и интерактивно, в процессе моделирования в редакторе. В основном для определения ограничений используют функциональные языки, например Object Constraint Language (OCL). Один из поддерживаемых языков в среде OMG/MOF/UML (рис. 6).

Преобразования модели, преобразования модель-модель. Каждая модель представляется с помощью какого-то конкретного синтаксиса, например xmi для моделей основанных на MOF. Однако, определять преобразования в терминах конкретного синтаксиса очень сложная, запутанная и не эффективная задача. Поэтому предлагается следующая схема (рис. 7):

– преобразовать модель в объектную структуру;

– преобразовать исходную модель в выходную (в объектной структуре);

– преобразовать целевую модель в конкретный синтаксис.

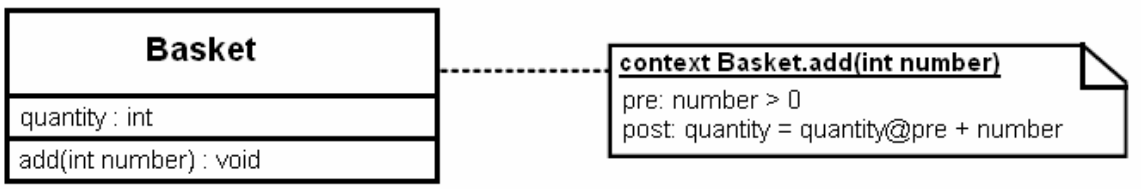

Рис. 6. Определение ограничения с помощью OCL

Этот подход к преобразованию гораздо эффективнее. В этом случае преобразователь становится намного гибче, поскольку он может преобразовывать модели с любым конкретным синтаксисом. Это очень важно в случае конкретного синтаксиса, основанного на XMI, поскольку детали XMI могут значительно отличаться между инструментами UML. И нет никакой необходимости привязывать преобразователь к конкретному синтаксису.

Генерация кода используется для генерации исполняемого кода из моделей, на основе метамодели. Существует три подхода к генерации кода:

– итеративный – где перечисляются все узлы исходной модели и используя язык преобразований генерируются узлы целевой модели;

– использование шаблона или макросов. Шаблоны работают аналогично шаблонам Velocity или XSLT. Шаблон похож на целевой язык, но позволяет вставлять макрос в любой части шаблона. Макросы это участки кода, которые будут выполняться, когда запуститься трансформация. Шаблоны также могут быть полезны в других задачах, таких как рефакторинг, оптимизация и т.д;

– использование шаблонов поиска, для поиска мест в исходной модели, где можно применить преобразование. В этом подходе можно представить шаблон как некое регулярное выражение для концептуальной модели. По аналогии с предыдущим подходом создается язык шаблона, основанный на исходном языке. Язык шаблона похож на исходный язык, но содержит гибкие критерии, позволяющие искать сложные совпадения в исходной модели. Можно представить этот подход как мощную технологию поиска и замены. Этот подход также может использоваться для проверки кода в редакторах.

Рассматривая процесс генерации кода, должна быть обеспечена возможность настройки на целевую платформу. Различные преобразования могут быть выполнены в зависимости от платформы. Очень сложно включить все возможные варианты в один набор преобразований, которые бы обслуживали все возможные случаи. Для решения этой проблемы предлагается разделить генерацию на два этапа (рис. 8). Первый этап будет читать конфигурацию и создавать генератор для текущего преобразования. Второй – использование созданного генератора для выполнения преобразований.

Данный подход широко используется в поставках *open source* дистрибутивов, где первый вызов *make install* настраивает инсталлятор (makefile) и вторым шагом собирается и устанавливает приложение.

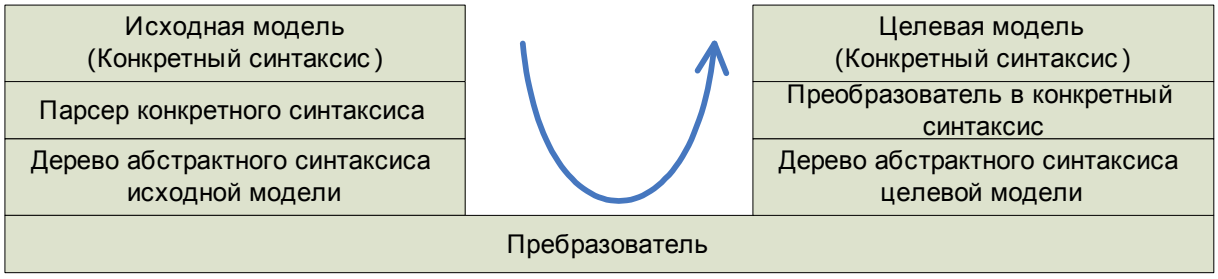

Рис. 7. Преобразование модели

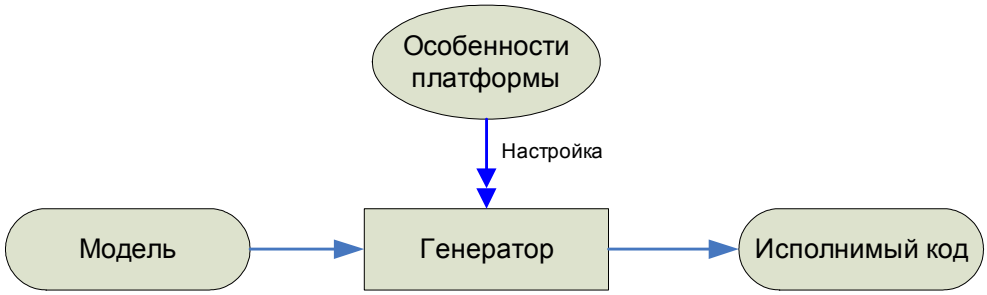

Рис. 8. Настройка на целевую платформу

Интеграция сгенерированного кода с рукописным. Если не все 100 % исполнимого кода генерируется из модели, то тогда незаполненные участки должны быть дописаны вручную. Однако, модифицирование сгенерированных файлов путем добавления рукописного кода создаст проблемы с соответствием, управлением сборки, версиями и переписыванием ручного кода при регенерации из моделей. Если сгенерированный код никогда вручную не модифицировался, то он может быть удален и перегенерирован заново. В случае, если этот код будет модифицирован, то в нем должны быть специальные защищенные области, чтобы генератор не мог удалить код при его регенерации. Для этого генератор должен перечитать ранее сгенерированный код, проанализировать его и сохранить защищенные области. Для решения вопросов использования сгенерированного и исходного кода предлагается:

– хранить сгенерированный и рукописный код в отдельных файлах;

– никогда не модифицировать сгенерированный код;

– разрабатывать дизайн и архитектуру, которые четко определяют какие артефакты сгенерированны, а которые нет;

– использовать подходящие дизайнерские решения склеивания сгенерированых и рукописных частей;

– интерфейсы также как и шаблоны проектирования такие как фабрика, стратегия, мост, шаблонный метод – хорошие исходные позиции [3].

Обобщая данный вопрос на тот случай, когда разные генераторы генерируют различные части общей системы, можно рассмотреть рукописный код как выход очень специфичного генератора – программиста.

На следующей диаграмме показано как сгенерированный и исходный код могут быть скомбинированы, используя выше описанные шаблоны проектирования [2] (рис. 9).

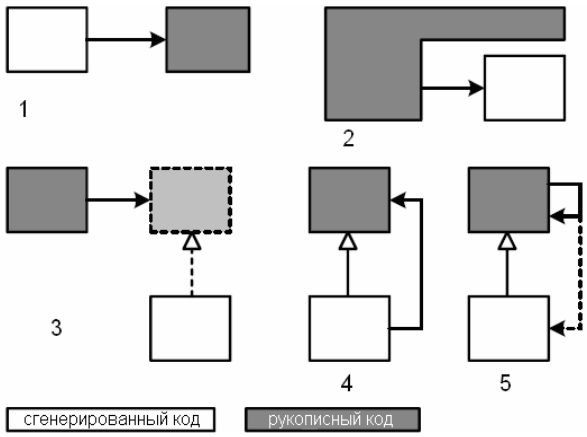

Рис. 9. Способы комбинирования сгенерированного и рукописного кода

1. Сгенерированный код может вызывать рукописный содержащийся в библиотеках. Этот способ предлагает генерировать минимальное количество кода и использовать уже реализованные компоненты.

2. Рукописный framework может вызывать сгенерированные части.

3. Фабрики могут быть использованы, чтобы «подключить» сгенерированные блоки.

4. Рукописные классы могут содержать полезные общие методы, которые могут вызываться из внутренних сгенерированных подклассов.

5. Базовый класс может содержать абстрактный метод, который он вызывает, который может быть реализован в сгенерированном подклассе.

Выводы

Данная работа – обзор подходов к организации архитектуры среды разработки на основе парадигмы MDD. Рассмотрены вопросы касающиеся особенностей реализации отдельных составляющих элементов этой среды разработки. Основа построения ее архитектуры – методология разработки DSL. Также проанализированы преимущества и недостатки парадигмы MDD. Автор собирается использовать данные разработки при проектировании и реализации инструментов для языков DSL.

1. *Greenfield J., Short K., Cook S., Kent S.* Software Factories: assembling applications with patterns, models, frameworks, and tools. Wiley Publishing Inc., September 2004. – P. 57 – 116.
2. *Stahl T., Voelter M.* Model-Driven Software Development, Wiley, 2006. – P. 184 – 185.
3. *Gamma E., Helm R., Johnson R., Vlissdes J.* Design Patterns, Addison-Wesley, 1995. – 416 p.
4. *Rudorfer, Voelter*. Domain-specific IDEs in embedded automotive software, http://www.voelter.de/data/presentations/EclipseCon.pdf
5. *GMF* Tutorial Wiki, http://wiki.eclipse.org/index.php/Graphical_Modeling_Framework
6. *Herrington J*. Code Generation in Action. Manning, 2003. – 368 p.
7. *Walkthrough.* Domain-Specific Language (DSL) Tools, 2005.

Получено 29.10.2007

***Об авторе*:**

Рябко Дмитрий Михайлович,
аспирант Института программных систем НАН Украины.

Место работы автора:

Институт программных систем НАН Украины,
03680, Киев, проспект Академика Глушкова 40.
Тел.: (097) 987 4829.
e-mail: dmytro.ryabko@gmail.com