Московский Энергетический Институт
(Технический Университет)

Курсовой проект
по курсу: «Распределённые информационные системы и базы данных»

На тему: «Архитектура, управляемая моделью (Model-driven architecture, MDA)»

Студент Макашова М.Б.

Группа А-13-07

Преподаватель Куриленко И.Е.

Москва 2011

Оглавление

Введение

Последние годы наиболее часто используемым подходом к проектированию программного обеспечения (ПО) является объектно-ориентированный подход (ООП) . Разработанный в 80-х годах XX века он активно развивается. В основе подхода лежат понятия класса (типа), объекта (экземпляра класса), метода (некоторой реализуемой функциональности) и атрибута (характеристики). Система, построенная согласно данному подходу, представляет собой совокупность взаимодействующих между собой объектов. Задачей разработчика является разработка эффективной иерархии классов для реализации приложения, основываясь на трех основных принципах ООП: полиморфизме, инкапсуляции и наследовании. Первые среды, поддерживающие ООП, реализовывали функции упрощенного создания синтаксических конструкций (классов, методов, атрибутов). Кроме того, поставщиками сред стали разрабатываться библиотеки классов, реализующих сервисные функции в объектно-ориентированном стиле (например, скрывающие вызовы операционной системы (ОС), предоставляющие возможность упрощенной работы с пользовательским интерфейсом и др.). Со временем сложность поставляемых библиотек росла в интересах поддержки новых технологий разработки, создания распределенных приложений, работы с Internet. Кроме того стала очевидна низкая степень совместимости созданных на их базе сред и приложений. В результате по мере увеличения степени сложности систем все более явными становились сложности создания объектно-ориентированных приложений:

- высокий риск ошибок, заложенных при дизайне системы из-за недостаточного опыта исполнителя или высокой степени сложности ПО;
- сложности при разработке распределенных приложений и интеграции созданных ранее систем из-за отсутствия унифицированных способов сетевого взаимодействия;

- низкая степень совместимости сред разработки и приложений, построенных на базе платформ от разных поставщиков;
- сложности при переделке систем как результат исторического развития архитектуры и низкого качества или отсутствия документации.

Перечисленные проблемы привели к удорожанию разработки и осознанию необходимости поиска решения озвученных проблем.

Одним из способов решения первой проблемы является использование шаблонов проектирования и описанных архитектурных подходов. Шаблоны проектирования являются важным этапом эволюции подходов к разработке, расширяя концепцию повторного использования результатов разработки на архитектурный уровень. Они активно используются на практике, а знание шаблонов проектирования стало одним из критериев для определения профессионального уровня разработчика. В тоже время, использование шаблонов привело к появлению новой проблемы. Недостаточно обдуманное применение шаблонов разработчиками низкого уровня приводит к перегруженности архитектуры, даже если она построена как совокупность известных шаблонов проектирования.

Другим способом снижения риска ошибок на стадии разработки архитектуры является ее моделирование. В этом случае можно говорить о визуализации разрабатываемой архитектуры посредством некоторого программного средства и методологии моделирования. Такие средства и модели имели независимое развитие и получили широкое распространение не только в сфере разработки ПО, но и за ее рамками. В частности широкое распространения получило моделирование бизнес процессов (Business Process Modeling, BPM). Моделирование способствует решению третьей и четвертой проблемы из списка выше. В области разработки ПО средства моделирования постепенно эволюционировали от средств визуального

представления к инструментам с поддержкой автоматической генерации кода и далее к комплексным интегрированным средам.

Концепция ООП к архитектуре приложений нашла свое развитие в компонентно-ориентированном подходе. Главным отличием последнего является использование понятия компонента, а не класса или объекта в качестве базовой конструкции. Компонент – это составная единица программной системы, четко заданная на уровне интерфейса и связей с другими компонентами. Компонентно-ориентированный подход можно рассматривать как основу для сервис-ориентированного подхода, являющегося в последние несколько лет одним из наиболее популярных.

Как результат развития перечисленных направлений выделился ряд архитектурных подходов, отражающих концепцию построения сложного ПО и соответствующие практические рекомендации. Часто эти подходы не являются взаимоисключающими и дополняют друг друга.

Model Driven Architecture (MDA, архитектура управляемая моделью) – это архитектурный подход к построению многокомпонентного ПО, основанный на разработке независимого от платформы и языка программирования представления системы (модели) с последующим переходим к исходному коду системы через автоматизированную генерацию кода. Разработка MDA началась в 2000 году. Предпосылкой к разработке являлся тот факт, что уже к концу 90-х годов прошлого столетия было создано большое количество технологий и протоколов для создания распределенных приложений. В качестве примеров можно привести COM/DCOM, CORBA, Java/RMI, XML/SOAP/RPC и др. Большинство таких технологий созданы зависимыми от платформы, не все из них были описаны в виде стандарта. Это создавало большие проблемы при разработке ПО и интеграции компонент в единую инфраструктуру.

Основной целью разработчиков MDA являлось создание архитектурного подхода, позволяющего снизить риск, вызванный различиями между и технологиями разработки программных систем и платформами. Для решения

этой задачи в MDA вводятся понятия модели, зависимой от платформы (Platform Specific Model (PSM)), и модели, независимой от платформы (Platform Independent Model (PIM)). MDA предполагает создание PIM и последующий переход к PSM с использованием специализированных средств. Кроме того возможен переход от одной PSM модели к другой.

Консорциум OMG как разработчик MDA не предоставляет никаких программных средств для обеспечения перехода от модели к модели. Предполагается, что они будут реализоваными силами сторонних разработчиков. Фактически, MDA – это концепция разработки, поддержанная группой стандартов (UML, MOF, CWM и XMI), разработанных OMG. Эти стандарты накладывают требования, которым должны удовлетворять модели, и предоставляют рекомендации для разработчиков сред создания ПО по MDA [1].

Основные понятия

Модель - описание или спецификация системы и ее окружения, созданная для определенных целей. Часто является комбинацией текстовой и графической информации. Текст может быть описан специализированным или естественным языком.

Управление на основе модели - процесс разработки системы, использующий модель для понимания, конструирования, распространения и других операций.

Платформа - набор подсистем и технологий, которые представляют собой единый набор функциональности, используемой любым приложением без уточнения деталей реализации.

Вычислительная независимость - качество модели, обозначающее отсутствие любых деталей структуры и процессов.

Платформенная независимость - качество модели, обозначающее ее независимость от свойств любой платформы.

Вычислительно-независимая модель - модель, скрывающая любые детали реализации и процессов системы; описывает только требования к системе и ее окружению.

Платформенно-независимая модель - модель, скрывающая детали реализации системы, зависимые от платформы, и содержащая элементы, не изменяющиеся при взаимодействии системы с любой платформой.

Платформенно-зависимая модель - модель системы с учетом деталей реализации и процессов, зависимых от конкретной платформы.

Модель платформы - набор технических характеристик и описаний технологий и интерфейсов, составляющих платформу.

Преобразование модели - процесс преобразования одной модели системы в другую модель той же системы [2].

Цель MDA

В современных программных системах активно используются много различных стандартов и технологий промежуточного слоя — CORBA, DCOM, .Net, Web-службы, технологии, основанные на Java и т.д. Все чаще возникает потребность в интеграции и обеспечении взаимодействия систем, основанных на разных технологиях, а также в модернизации существующих программ и их переработке в соответствии с новой технологической основой. Новая архитектурная концепция, предложенная консорциумом OMG, основанная на модельно-ориентированном подходе к разработке программного обеспечения, позволяет существенно упростить и частично автоматизировать разработку, интеграцию и модернизацию систем.

В связи с большим количеством используемых и разрабатываемых стандартов и технологий стало очевидно, что попытка создать единый универсальный стандарт построения и взаимодействия программных систем обречена на неудачу. Примером тому стандарт CORBA, который, хотя и приобрел определенную известность, так и не занял предполагаемого для него места «универсального стандарта». Поэтому консорциум OMG принял решение перейти от «стандарта интеграции» к интеграции стандартов. Концепция MDA (Model Driven Architecture) призвана обеспечить общую основу для описания и использования большинства существующих стандартов, не ограничивая разработчиков в выборе конкретных технологий.

Интеграция стандартов достигается за счет:

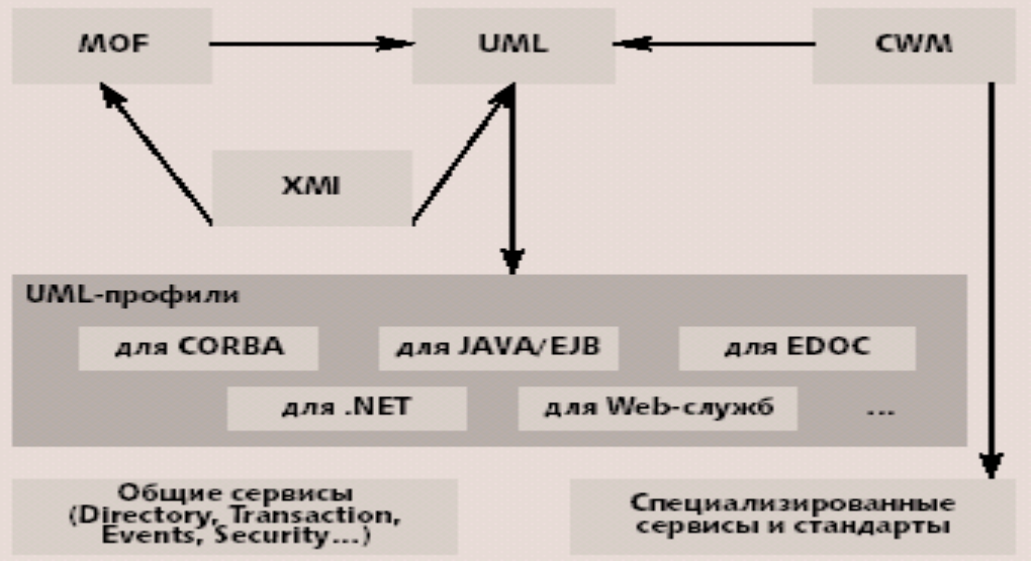
- введения концепции платформно-независимой модели приложения;
- использования унифицированного инструмента (UML) для описания таких моделей;
- наличия разработанных OMG стандартных отображений моделей в среду большинства технологических платформ и программных инструментов промежуточного слоя.

Использование MDA для разработки и интеграции программного обеспечения позволяет сохранить инвестиции, сделанные в разработку бизнес-логики даже при смене технологических платформ.

Модельно-ориентированный подход облегчает интеграцию как разнородных, основанных на различных технологиях распределенных систем, так и организацию взаимодействия между системами, основанными на одной технологии, но использующими разные интерфейсы, сервисы и стандарты. Кроме того, MDA позволяет разрабатывать стандартные сервисы (репозитарии, сервисы событий и сообщений и т.д.), которые поддерживают сразу несколько технологий промежуточного слоя. Например, можно создать репозитарий объектов, который предоставляет соответствующий сервис системам как на базе CORBA, так и на базе Web-сервисов. Это не только упрощает разработку программного обеспечения, но и позволяет организовать взаимодействие различных систем [3].

Ядро MDA

Ядром MDA являются несколько стандартов — UML, MOF, CWM и XMI. Язык UML (Universal Modeling Language) используется для описания всех моделей. Совокупность метамоделей CWM (Common Warehouse Model) представляет наиболее часто используемые в базах данных и инструментах бизнес-анализа метаданные. CWM содержит большое количество различных метамоделей, описывающих функционирование бизнеса. MOF (Meta Object Facility) — общий абстрактный язык для описания метамоделей; на его основе построены формальные описания метамодели для CWM и UML. Последний стандарт, XMI (XML Metadata Interchange), играет служебную роль, описывая отображение моделей MOF и UML на стандарт XML. При этом метамодели преобразуются в DTD-структуру документа, а модели — в тело XML-документа. Это позволяет объединить модель и ее метамодель в одном документе и получить так называемый «самоописываемый» (self-describing) документ, содержащий не только данные, но и информацию, необходимую для их интерпретации.



В основе MDA лежит понятие *платформно-независимой модели* (platform independent model, PIM). Речь идет о детальной исполняемой модели на языке действий UML (action semantics) с пред- и постусловиями, сформулированными на OCL.

Вполне логично, что для описания PIM выбран язык UML. Этот язык принципиально позиционировался как независящий от платформ и технологий. Однако UML в своих ранних версиях не являлся «точным» языком: он только предоставлял дизайнеру возможность описать структуру и поведение системы, практически не определяя ее функционирование и используемые алгоритмы. В новый стандарт UML 2.0, который был одобрен OMG в июне 2003 года, включено большое количество средств, позволяющих описывать внутреннюю организацию и функционирование системы. Одна из основных задач, которые были решены при создании этого стандарта, — превратить UML в алгоритмически полный исполнимый язык, в то же время, по возможности, не повышая уровня детализации UML-моделей.

Несмотря на то, что PIM — это детальная исполняемая модель, ее вряд ли можно использовать на практике как финальный программный продукт. UML-инструментарий нового поколения предоставляет унифицированную среду, в которой можно интерпретировать эту модель и получать исполняемый код прототипного качества. Такой код может быть крайне неэффективным, не удовлетворять некоторым функциональным требованиям, не полностью реализовывать функциональность системы и даже требовать участия человека в процессе исполнения. Инструментарий, используемый для интерпретации UML-модели, может быть полуавтоматическим и допускать вмешательство оператора. Только после привязывания к конкретной платформе можно получить код промышленного качества. Но, хотя исполнение PIM-модели и нельзя применять для решения практических задач, оно необычайно важно для целей тестирования и отладки. Фактически, у разработчиков появляется возможность получить

первый прототип системы еще до начала стадии кодирования, когда сравнительно легко вносить даже существенные изменения в систему (в том числе, изменения требований и технического задания).

Многие крупные производители объявили о поддержке MDA и начале разработки соответствующих инструментов, и в ближайшее время можно ожидать выхода первых инструментариев разработки с использованием MDA. Но, вероятно, потребуется значительный период времени, чтобы инструменты развились и смогли максимально использовать возможности технологии. Пока это не сделано, значительную часть рутинной работы придется выполнять вручную [3].

Архитектура CWM

CWM

Спецификация Common Warehouse Metamodel (Общая метамодель Хранилища данных, далее CWM) определяет метамодель (модель модели данных), представляющую как бизнес, так и технические метаданные, которые в большинстве случаев присутствуют в области технологии Хранилищ данных и бизнес аналитики. Она используется как основа для обмена экземплярами метаданных между гетерогенным программным обеспечением, поставляемым различными производителями. Системы, которые "понимают" метамодель CWM, обмениваются данными в форматах, которые согласуются с этой моделью.

CWM выражен на языке UML (Unified Modeling Language, Унифицированный язык моделирования). Но, хотя UML является нотационным основанием для определения CWM, CWM расширяет базовую метамодель UML с помощью концепций технологий Хранилищ данных и бизнес-анализа.

Можно сказать, что CWM расширяет язык UML в том смысле, что каждый метакласс (metaclass) CWM наследуется напрямую, либо ненапрямую из метаклассов UML. Таким образом, CWM можно характеризовать как язык определенной области применения, предназначенный для определения моделей Хранилищ данных.

CWM определяющая метамодель для обмена экземплярами метаданных между различными средствами моделирования и информационными системами [4].

MOF

Другой стандарт OMG - Meta Object Facility (Средство метаобъекта, MOF) - определяет общие интерфейсы и семантику для взаимодействующих метамоделей. MOF - это пример мета-метамодели, или модели метамодели

(подмножество UML). Он также определяет набор IDL-преобразований (Interface Definition Language, язык описания интерфейса, который устанавливает спецификацию интерфейса для обнаружения и управления моделей с помощью программных APIs).

Помимо определения общей семантики для метамоделей MOF также служит в качестве модели для UML (то есть в конечном итоге MOF определяет язык, на котором выражается метамодель UML). Поскольку CWM наследуется из UML, MOF также является моделью и для CWM. Все модели CWM выражаются на UML и реализуют семантику MOF.

MOF состоит из четырех уровней, верхний из которых соответствует мета-метамодели. Третий уровень соответствует метамодели (например, UML). Второй уровень соответствует экземпляру UML модели, описывающее какую-либо систему. Самый нижний уровень отражает экземпляры моделируемых объектов [4].

XMI

Наконец, третий стандарт, который непосредственно задействован в обмене метамоделями - это XMI. XMI (XML Metadata Interchange, Обмен метаданными XML) - это стандарт OMG, который устанавливает правила преобразования метамоделей MOF в XML. XMI определяет, как использовать XML-теги для представления сериализованных моделей, совместимых с MOF. Метамодели MOF транслируются в XML DTD, а модели - в XML-документы, которые согласуются со своими DTD.

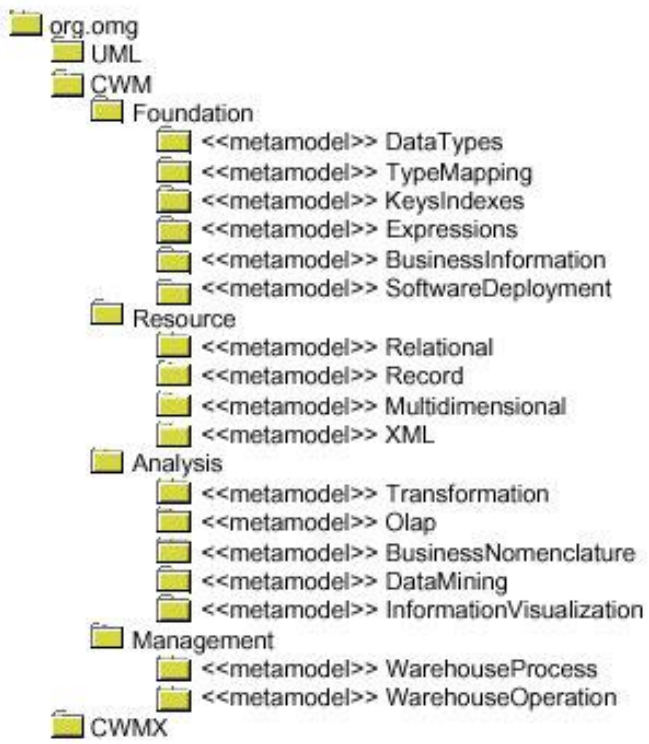
XMI описывает отображение метамоделей, построенных согласно MOF в XML, который в свою очередь может являться основой для обмена метаданными [4].

Представление метамодели

Каждая метамодель CWM представляется как XML DTD (в соответствие с правилами XMI), так и определение IDL. В первом случае

модели CWM преобразуются в поток (serialize), после чего ими обмениваются, как документами XMI. При экспорте метаданные посредством XMI-документа, необходимо выполнить XMI-преобразование (MXI-rendering) в форме, которая легальна по отношению к DTD. При импорте данных с помощью XMI-документа, следует проверять модель на допустимость по этим DTD.

Во втором случае моделей объектов CWM создаются в памяти или хранятся в репозитории - в этой ситуации IDL предпочтительней, поскольку он определяет необходимые интерфейсы, подписи методов и структуру совокупности , которые эта модель должна поддерживать.



Итак, CWM фактически состоит из ряда составных метамоделей (или суб-метамоделей), которые организованы в виде следующих 4 слоев: базовый слой (Foundation), источники данных (Resources), анализ (Analysis) и управление Хранилищем (Management) .

Базовый слой состоит из метамоделей, которые поддерживают моделирование таких различных элементов и сервисов, как типы данных,

системное преобразование типов, абстрактные ключи и индексы, выражения, бизнес-информация и включения программного обеспечения, основанного на использовании компонентных объектов.

Слой источников данных предоставляет возможность моделировать существующие и новые источники данных, в том числе реляционные базы данных, ориентированные на запись базы данных (record oriented databases), а также XML- и основанные на объектах (object-based) источники данных.

Слой анализа предоставляет средства для моделирования сервисов информационного анализа, которые обычно используются в Хранилище данных. Он определяет метамодель для преобразования данных, OLAP, визуализации информации/репортинга (business nomenclature) и data mining.

Слой управления состоит из метамоделей, представляющих стандартные процессы и операции Хранилища данных, журнализации (activity tracking) и планирования работ [scheduling] (например, ежедневной загрузки и выгрузки).

Этот набор метамоделей, предоставляемых CWM, достаточен для моделирования всего Хранилища данных. Используя инструмент, поддерживающий CWM, можно было бы сгенерировать экземпляр Хранилища данных прямо из модели Хранилища данных. Каждый из этих различных инструментов использует те части модели, которыми можно воспользоваться. Например, сервер реляционной базы данных задействует реляционный блок этой модели и будет использовать его для построения его каталога. Аналогично OLAP-сервер будет отыскивать в модели метаданные OLAP и использовать их для определения многомерной схемы. А инструмент извлечения, преобразования и загрузки данных (ETL) скорее всего обработал бы срез модели Хранилища данных, которая охватывает несколько метамоделей CWM, в том числе метамодели OLAP, преобразования, типа данных, преобразования типов, выражений и реляционную метамодель [4].

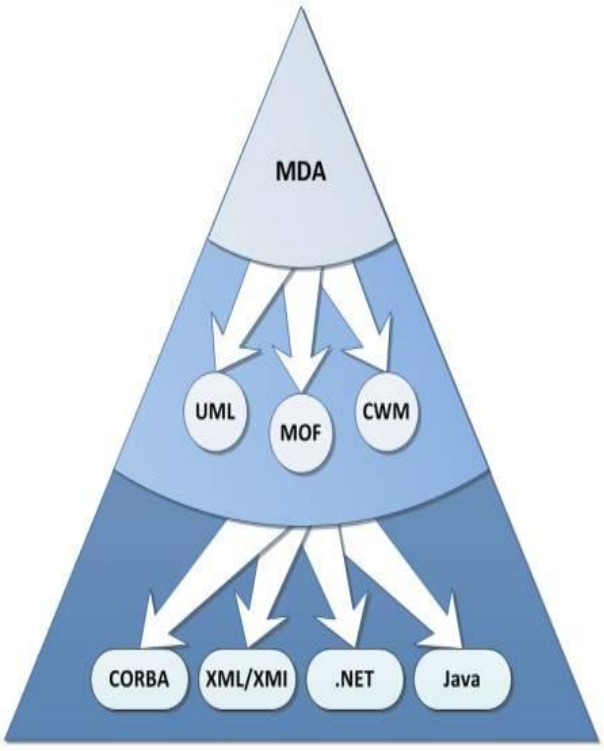
Технология Model Driven Architecture (MDA), суть, перспективы использования.

Model Driven Architecture — модельно-ориентированный подход к разработке программного обеспечения.

Суть этой технологии состоит в построении абстрактной метамодели управления и обмена метаданными (моделями) и задании способов ее трансформации в поддерживаемые технологии программирования (Java, CORBA, XML и др.).

Архитектура MDA предлагает новый интегральный подход к созданию многоплатформенных приложений и базируется на трех основных элементах:

- UML(Unified Modelling Language) унифицированный язык моделирования;
- MOF (MetaObject Facility) абстрактный язык для описания информации о моделях (язык описания метамоделей);
- CWM (Common Warehouse Metamodel) общий стандарт описания информационных взаимодействий между хранилищами данных.



На центральном уровне структуры находится собственно MDA, которая «разворачивается» наружу посредством второго уровня, содержащего вышеперечисленные базовые составляющие UML, MOF и CWM, и на третьем, внешнем уровне представлены некоторые из широко известных в настоящее время программных платформ разработки: CORBA,

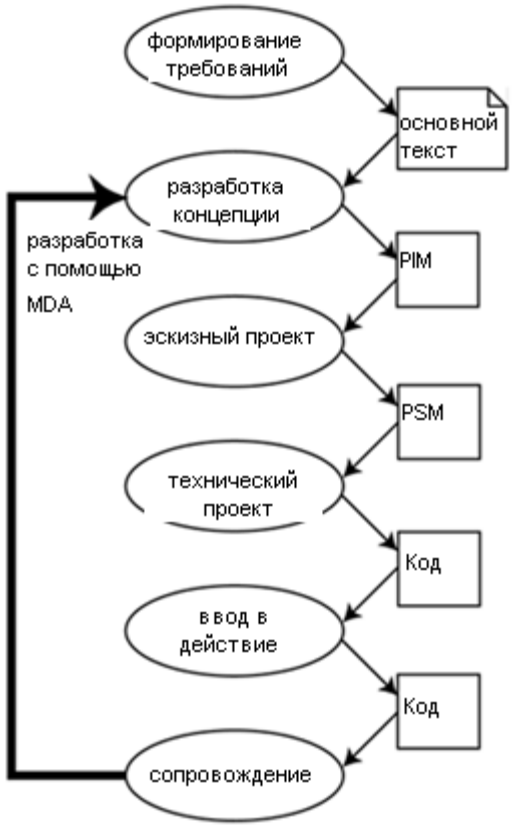
XML, .NET, JAVA. На этом внешнем уровне, по замыслу OMG, могут и должны быть размещены и все возможные будущие технологии разработки. Этим подчеркивается тот факт, что OMG считает архитектуру MDA не просто новой технологией, а скорее «метатехнологией» создания приложений. Последняя отныне будет и единственно актуальной — вне зависимости от развития и появления новых средств разработки, которые MDA уже «заранее интегрировала» в себя.

Само понятие «разработчик программного обеспечения» может при внедрении MDA довольно сильно видоизмениться. Со смещением акцента на создание модели разработкой приложений будут заниматься не столько программисты, сколько специалисты, владеющие описываемой предметной областью. Возможно, что также в какой-то степени «пострадает» традиционное деление специалистов на разработчиков баз данных и разработчиков приложений баз данных. Уже сейчас возможно при разработке MDA-приложений практически полностью абстрагироваться от знания конкретной СУБД; более того, во многих случаях нет необходимости и использовать язык SQL, поскольку многие инструменты MDA предоставляют возможность работать на более «высоком» уровне (бизнес-уровне), где становится абсолютно не важным знание конкретной структурной схемы базы данных или состава полей ее таблиц.

Однако программисты-разработчики вряд ли останутся без работы, так как, с одной стороны, создание MDA-инструментария само по себе является чрезвычайно интересной, сложной и объемной задачей для них. А с другой стороны, внедрение MDA уже сейчас избавляет и самих программистов от рутинной работы, передавая большую ее часть искусственному программному интеллекту — инструментам реализации MDA [5].

Жизненный цикл разработки с помощью MDA

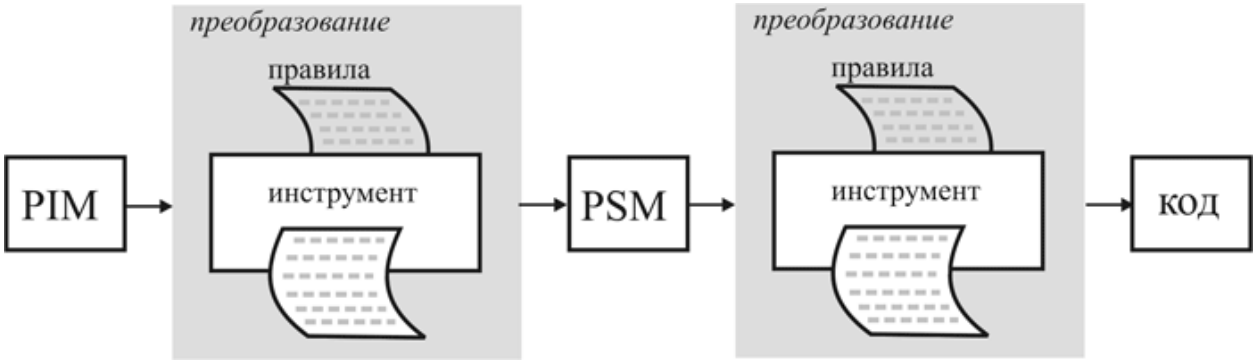
Жизненный цикл разработки с помощью MDA не сильно отличается от традиционного. Он разделен на такие же этапы.



Одно из существенных различий состоит в природе артефактов, которые создаются во время процесса разработки. Здесь артефакты являются формальными моделями, то есть (говоря примитивным языком), моделями, понятными компьютерам. Далее рассмотрим три вида моделей, из которых состоит ядро технологии MDA [6].

Типы моделей

Архитектура MDA описывает и структурирует поэтапный процесс разработки любых программных систем на основе создания и использования моделей. При этом используется несколько типов моделей, создаваемых и преобразуемых на различных этапах разработки. Процесс разработки по MDA это последовательное (поэтапное) продвижение от одной модели системы к другой. При этом каждая последующая модель преобразуется из предыдущей и дополняется новыми деталями. Модели, общая схема разработки и процесс преобразования моделей - ключевые составные части архитектуры.



Основная идея MDA в том, что преобразование из PIM в PSM, а также же генерация кода из PSM, может производиться автоматически. Преобразования проводятся при помощи *инструментов преобразования* (transformation tools), которые в свою очередь используют *правила преобразования*. Эти правила будут написаны на языке, который будет описан стандартом QVT (Queries, Views, Transformations). Преобразования могут быть параметризованы, что позволит их подстраивать под нужды конкретных проектов.

Рассмотрим типы моделей, используемых в архитектуре MDA [2].

CIM

Common Information Model (типовая информационная модель, CIM) — открытый стандарт, определяющий представление управляемых элементов IT среды в виде совокупности объектов и их отношений, предназначенный обеспечить унифицированный способ управления такими объектами, вне зависимости от их поставщика или производителя.

В упрощенном виде CIM можно представить как способ, позволяющий нескольким участникам обмениваться информацией, необходимой для управления их элементами. Упрощение заключается в том, что CIM не только определяет представление управляемых элементов и управляющей информации, но и предоставляет возможность управлять ими и контролировать их работу. Управляющее программное обеспечение, созданное с использованием CIM, может работать с множеством реализаций этого стандарта без потери данных или сложных перекодировок.

CIM разработан и опубликован Distributed Management Task Force. Связанный с ним стандарт Web-Based Enterprise Management (также разработанный DMTF), определяет реализацию CIM, включая протокол обнаружения и доступа.

Стандарт CIM включает спецификацию инфраструктуры и схему.

Спецификация инфраструктуры определяет архитектуру и понятия CIM, включая язык определения CIM Schema (и любых её расширений), и способ отображения CIM на другие информационные модели, например SNMP. Архитектура CIM объектно-ориентированная, поскольку основывается на UML: управляемые элементы представляются классами CIM, любые отношения между ними представляются ассоциациями CIM, а наследование позволяет создавать специализированные элементы из более простых базовых.

Схема SIM - концептуальная схема, определяющая набор объектов и отношений между ними, представляющих общую основу управляемых

элементов в IT среде. Схема охватывает большую часть современных элементов IT среды, например компьютеры, операционные системы, сети, подпрограммное обеспечение, сервисы и хранилища. Схема определяет общий базис представления таких элементов. Поскольку большинство управляемых элементов для каждого типа элемента и его производителя имеют своё поведение, схема является расширяемой и даёт возможность производителям представлять специфический функционал сходным образом с базовым функционалом, определенном в схеме.

На CIM основаны либо используют большинство остальных стандартов DMTF (так как WBEM или SMASH). Также он является основой стандарта SMI-S, предназначенного для управления хранилищами.

Множество производителей предоставляют различные реализации CIM:

- большинство операционных систем предоставляют реализацию CIM. Например, CIM реализован в семействе Microsoft Windows (WMI) и некоторых дистрибутивах GNU/Linux
- CIM и WBEM активно применяется в области сетей хранения данных в виде основанного на CIM стандарта SMI-S, определенного ассоциацией SNIA
- большинство производителей серверов сотрудничают с DMFT в рамках стандарта SMASH, основанного на CIM
- DMTF разрабатывает стандарт DASH управления настольными компьютерами [7]

CIM описывает общие требования к системе, словарь используемых понятий и условия функционирования (окружение). Модель не должна содержать никаких сведений технического характера, описаний структуры и функционала системы. CIM максимально общая и независимая от реализации системы модель. Спецификация MDA подчеркивает, что CIM должна быть построена так, чтобы ее можно было преобразовать в платформенно-

независимую модель. Поэтому CIM рекомендуется выполнять с использованием унифицированного языка моделирования UML [2].

PIM

На этапе анализа на основании требований вырабатывается платформно независимая модель системы (PIM). Она привязана к постановке задачи и предметной области и не зависит от таких деталей реализации, как, например, язык программирования или тип базы данных (реляционная, объектная, иерархическая и т.д.) [8].

Платформенно-независимая модель (Platform Independent Model, PIM) описывает состав, структуру, функционал системы. Модель может содержать сколь угодно подробные сведения, но они не должны касаться вопросов реализации системы на конкретных платформах. Модель PIM создается на основе CIM. Для создания модели используется унифицированный язык моделирования UML [2].

PSM

Далее, на этапе дизайна будет осуществлен выбор деталей реализации: платформ, языков, распределенной или централизованной архитектуры. На основании эти решений PIM будет преобразована в соответствующие платформно зависимые модели (PSM). Для этого преобразования скорее всего будут использоваться готовые инструменты преобразования и библиотеки правил преобразования. Из одной PIM может быть сгенерировано несколько PSM. Например, одна из PSM может основываться на CWM метамодели для реляционых баз данных и описывать модель данных. В тоже время другая модель может, используя UML/EJB Mapping, представить PSI в терминах Enterprise Java Beans. Для стыковки разных PSM моделей в процессе PIM⇒PSM трансформации могут также быть сгенерированы так называемые «*bridges*» — связки между разными PSM моделями, сгенерированными из общей PIM модели [8].

Платформенно-зависимая модель (Platform Specific Model, PSM) описывает состав, структуру, функционал системы применительно к вопросам ее реализации на конкретной платформе. В зависимости от назначения модель может быть более или менее детализированной. Модель создается на основе двух моделей. Модель PIM служит основой модели PSM. Модель платформы используется для доработки PSM в соответствии с требованиями платформы [2].

Модель платформы

Ну и наконец, из PSM при помощи других инструментов преобразований и других наборов правил будет сгенерирован код. Например, если Oracle был выбран как реляционная база данных, то будет использован набор правил, который из соотвествующей PSM построит схему базы данных. Другой набор правил может сгенеририровать Java код для EJB контейнера. Опять же, при использовании нескольких PSM моделей будут сгенерированы связки на уровне кода между ними, которые, например, позволят сгенерированным Java Beans работать с сгенерированной схемой базы данных Oracle. Хотя преобразования будут делаться автоматически, выбор их параметров останется за дизайнером. Например, он может указать использовать ли определенные возможности платформы или нет, подсказать системе примерные объемы ожидаемых данных, и т.п [2].

Модель платформы описывает технические характеристики, интерфейсы, функции платформы. Зачастую модель платформы представлена в виде технических описаний и руководств. Модель платформы используется при преобразовании модели PIM в модель PSM. Для целей MDA описание модели платформы должно быть представлено на унифицированном языке моделирования UML [8].

Уровни модели

Уровень бизнес-логики

Уровень бизнес-логики содержит описание основного функционала приложения, обеспечивающего исполнение его назначения. Как правило, уровень бизнес-логики хуже всего поддается автоматизации. Он составляет львиную долю кода приложения, который приходится писать вручную [2].

Уровень данных

Уровень данных описывает структуру данных приложения, используемые источники, форматы данных, технологии и механизмы доступа к данным. Для приложений .NET чаще всего используются возможности ADO.NET [2].

Уровень пользовательского интерфейса

Уровень пользовательского интерфейса описывает возможности приложения по взаимодействию с пользователями, а также состав форм приложения, функционал элементов управления (например, контроль ввода данных). Легкость автоматизации этого уровня зависит от того, насколько унифицированы пользовательские операции. Если удается создать типовые шаблоны элементов управления для основных операций, появляется возможность автоматической генерации форм и их содержимого при создании приложения из модели [2].

Типы моделей

Бизнес-модели и модели программного обеспечения

Системы, которые описывают бизнес-модели, это предприятия или компании. Языки, использующиеся для бизнес-моделирования, содержат словарь, который позволяет разработчику модели определять бизнес-процессы, заинтересованных лиц, отделы, зависимости между процессами, и так далее.

Бизнес-модель не обязательно должна отражать какую-либо информацию о программных системах, используемых компанией. Поэтому она также называется вычислительно-независимой моделью (CIM – Computational Independent Model). Всякий раз, когда часть бизнеса-процесса поддерживается программной системой, пишется программная модель этой системы. Эта модель - описание системы программного обеспечения. Можно сделать вывод о том, что бизнес-модели и модели программных систем описывают совершенно разные в реальной жизни системы.

Однако, требования к программной системе часто вытекают из бизнес-модели, которую должно поддерживать разрабатываемое программное обеспечение. Для большинства бизнес-моделей существует множество

программных систем. Каждая система используется для поддержания какой-либо части главной бизнес-модели. Таким образом, есть отношение между бизнес-моделью и моделями программных систем, поддерживающих бизнес-процессы.

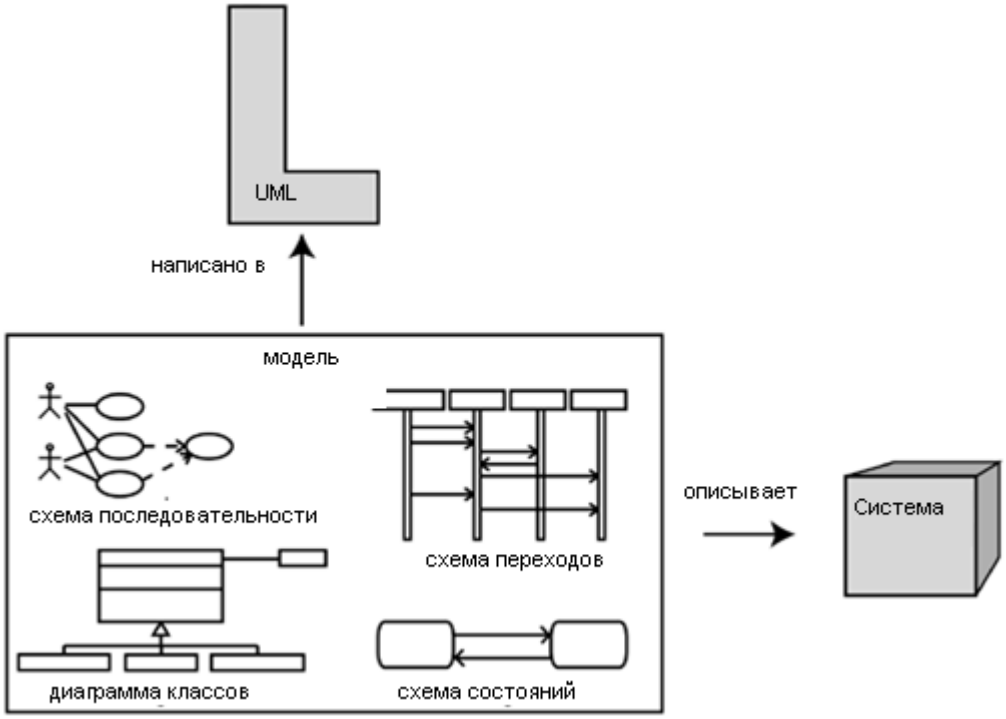
Тип системы, описанной моделью, очень важен для процесса преобразования. CIM – это модель, совершенно не зависящая от программного обеспечения. Она описывает только бизнес-процесс. Отдельные части CIM могут поддерживаться системами программного обеспечения, но сама CIM остается независимой. Автоматическая получение PIM из CIM не возможно, потому что составление списка частей CIM, которые должны отражаться в программной системе, всегда остается за человеком. Для каждой части CIM, поддерживаемой системой, сначала должна быть разработана PIM [9].

Структурные и динамические модели

В UML, например, диаграмму классов считают структурной моделью, а диаграмму состояний - динамической моделью, в то время как в действительности диаграмма классов и диаграмма состояний настолько зависят друг от друга, что они должны быть расценены как часть одной и той же модели.

Факт, что моделирование программного обеспечения начинается с создания диаграммы классов, не означает, что разрабатывается лишь модель класса. В этом случае при разработке программной модели определяются статические аспекты посредством статичного представления. Иногда разработка начинается с создания динамической схемы, такой как схема состояний или переходов. В этом случае определяются динамические аспекты посредством динамического представления. Позже, например, когда диаграмма состояний добавляется диаграмме классов, мы просто добавляем динамические аспекты к той же самой модели, или наоборот. Поэтому, общая терминология немного неверна. Диаграммы классов и диаграммы состояний

лучше называть структурными и динамическими представлениями. На рисунке показано, как различные схемы в UML являются представлениями одной и той же модели.

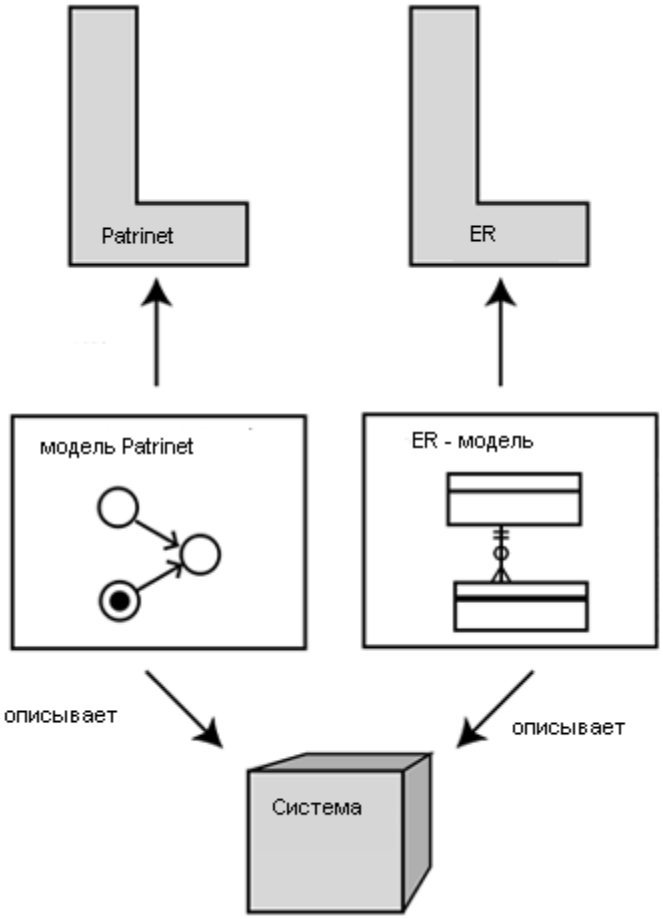


В UML существует прямая связь между статическими и динамическими представлениями, потому что они отражают один и тот же объект на разных уровнях визуализации. Например, класс в UML-модели выглядит как прямоугольник с именем класса в режиме «Представление класса», в то время как тот же класс выглядит как тип экземпляра в схеме последовательности. Схемы всех видов описывают один и тот же объект.

Система может иметь и структурный, и динамический аспекты. Если используемый язык способен отразить оба аспекта, есть возможность спроектировать полную модель системы. Примером такого языка может служить UML.

Если структурные и динамические аспекты не могут быть описаны в одной модели, потому что используемый язык не в состоянии отразить оба аспекты, естественно возникает надобность в двух разных моделях. Отметим, что обе модели связаны; они описывают одну и ту же систему. Тип модели в

таком случае может более понятно определен исследованием языка. Рисунок показывает ситуацию, где две различных модели, описывающие одну и ту же систему, записаны на двух различных языках.



Можно сделать вывод о том, что аспект, описанный в диаграмме или модели, не обязательно связан с типом модели. Существенная характеристика модели - язык, на котором записана модель. Некоторые языки являются более выразительными, чем другие и более подходящими для того, чтобы представить определенные аспекты системы [10].

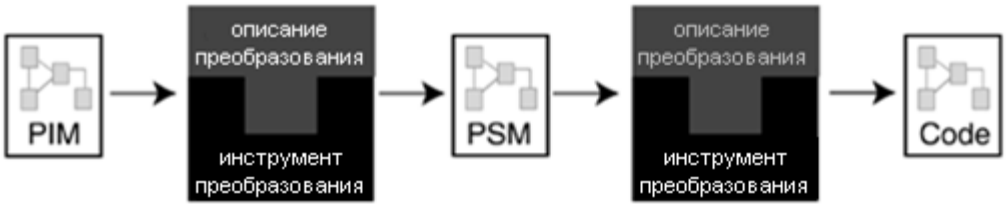
Платформо-независимые и зависимые модели

Существует несколько определений PIM и PSM моделей. В документации OMG эти модели кардинально отличаются. Модель может быть либо PIM, либо PSM. В реальности очень сложно провести черту,

разделяющую эти два типа моделей. Единственное, что можно сказать, это то, какая из них более зависима от платформы. В рамках MDA модели менее зависимые от платформы трансформируются в более зависимые от платформы модели. Поэтому понятия PIM и PSM сильно связаны [9].

Преобразование

Выше было рассказано, какие роли в концепции MDA играют PIM, PSM и код. Инструменты преобразования переводят PIM в одну или несколько PSM. Следующий инструмент преобразовывает PSM в конечный код. Эти преобразования естественны для процесса MDA. На рисунке ранее инструмент трансформации был показан как «черный ящик». Входными данными является модель одного типа, выходными – модель другого типа или код. При более пристальном рассмотрении инструмента преобразования можно увидеть элементы, которые принимают участие в самом процессе преобразования. Где-то внутри записаны правила, по которым происходит трансформация.



Важно понимать различие между непосредственно процессом преобразования, который состоит в генерации одной модели из другой, и правилами преобразования. Инструмент трансформации использует одни и те же правила для любой входной модели.

Правила задают связи между конструкциями исходного языка и языка, на котором описана выходная модель. Например, можно задать правила

преобразования из UML в C#. Они будут описывать, какие конструкции C# будут сгенерированы из разных UML-моделей.

В общем случае, правила преобразования – это однозначное описание способа, которым одна модель может быть использована для создания из нее другой. Основываясь на этих наблюдениях, можно дать определения преобразованию, описанию преобразования и правилам преобразования.

- Преобразование - автоматическая генерация целевой модели из исходной модели согласно описанию преобразования.
- Описание преобразования - ряд правил преобразования, которые описывают, как модель на исходном языке может быть преобразована в модель на выходном языке.
- Правило преобразования - описание того, как одна или более конструкций на исходном языке могут быть преобразованы в одну или более конструкций на выходном языке.

Чтобы быть работоспособной, преобразование должно обладать определенными характеристиками. Самая важная характеристика - преобразование должно сохранить соответствие между источником и целевой моделью. Значение отдельного элемента модели может быть сохранено, только если оно выражено и в источнике, и в целевой модели. Например, спецификация поведения может быть частью модели UML, но не ER-модели. Даже в этом случае должна быть возможность преобразовать UML-модель в ER-модель, сохраняя структурные характеристики системы [9].

Преобразования между идентичными языками

Сказанное выше не помещает ограничений на входные и выходные языки. Это означает, что исходная и целевая модели могут быть записаны либо на одном языке, либо на разных. Мы можем определить преобразования от UML-модели до UML- модели или от Java до Java.

Есть несколько примеров таких ситуаций. Метод рефакторинга модели или части кода (помним, что код также являются моделью) может быть описан определением преобразования между моделями, описанными на одном и том же языке. Другой известный пример - нормализация ER-модели. Есть четко определенные правила нормализации, которые могут быть применены много раз на различных ER-моделях с ожидаемым корректным результатом. Например, правило нормализации, которое приводит модель ко второй нормальной форме:

- переместить все атрибуты объекта, которые не зависят от полного ключа этого объекта, к отдельному объекту, сохраняя отношения между исходным объектом и заново создаваемым.

Это правило может быть применено к любой ER-модели. Оно разбивает одну сущность и ее атрибуты в исходной модели на две сущности, их атрибуты и отношения между ними в результирующей модели, где обе модели написаны на ER-языке.

В случае преобразований между UML-моделями нужно быть особенно аккуратным. Очень часто цели, которые выполняют исходная и целевая модели кардинально отличаются [9].

Основы фреймворка MDA

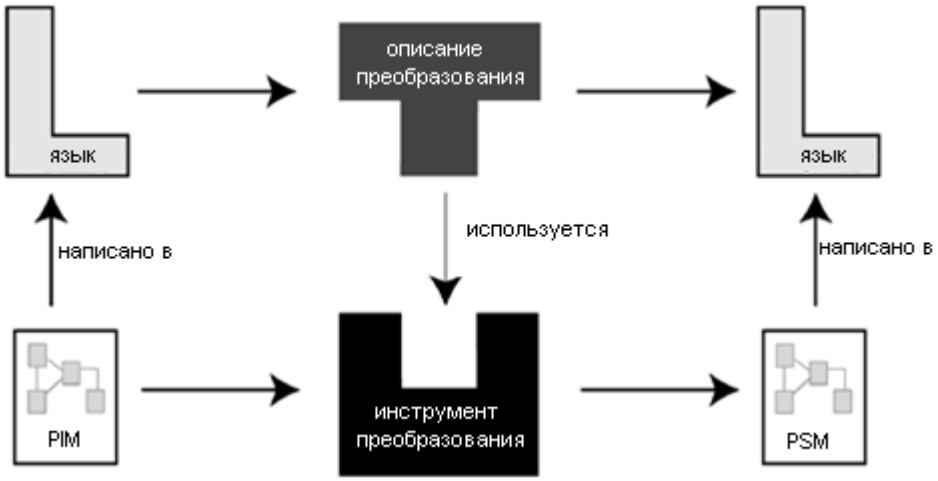
Как было сказано ранее, модель – это описание системы.

- PIM-модель – это платформо-независимая модель, описывающая систему без каких-либо знаний о конечной платформе, на которой будет размещена созданная система.
- PSM-модель – это платформо-зависимая модель, которая описывает систему с точки зрения знаний о том, на какой платформе будет размещена система.

Модель должна быть написана на хорошо определенном языке.

Описание преобразования показывает, как модель на исходном языке может быть преобразована в модель на результирующем языке.

Инструмент преобразования выполняет преобразование исходной модели, основываясь на описании преобразования.



С точки зрения разработчика самыми важными элементами являются PIM и PSM. Разработчик фокусируется на создании PIM, описывая систему на высоком уровне абстракции. На следующем этапе выбирается инструмент преобразования, способный провести трансформацию разработанной PIM

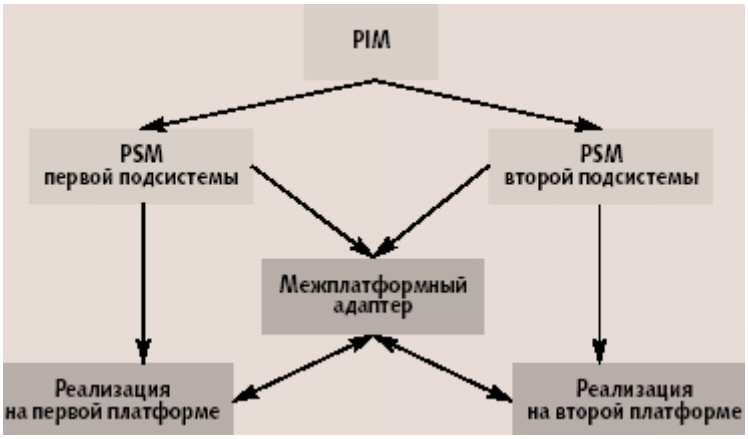
согласно какому-либо описанию преобразования. В результате получается PSM, которая в дальнейшем может быть преобразована в код.

На рисунке выше показана только одна PSM, но в то же время из одной PIM можно получить несколько PSM с готовыми мостами между ними.

В зависимости от уровня детализации платформы, модели (кроме модели платформы) могут содержать сведения о различных функциональных частях системы. В этом случае говорят об уровнях модели. Обычно различают следующие основные уровни модели [11].

Этапы разработки

Процесс разработки разбивается на три этапа.



Первый этап

На первом этапе разрабатывается вычислительно-независимая модель (CIM). Часто модель, создаваемую на этом этапе, также называют доменной или бизнес-моделью. Цель данного этапа разработка общих требований к системе, создание общего словаря понятий, описание окружения, в котором система будет функционировать. Сущности, описываемые в модели CIM этого этапа, должны тщательно анализироваться и отрабатываться. Право на включение в модель должны иметь только те элементы, которые будут использованы и развиты на последующих этапах разработки. Для создания модели CIM на данном этапе можно использовать любые средства. Однако для совместимости с последующими этапами весьма желательно иметь описание модели на языке UML. Следует учитывать, что модель CIM первого этапа представляет собой скорее общую концепцию системы и не является насущно необходимой для процесса разработки приложения. При создании небольших программных систем этот этап можно опустить, однако при работе со сложными проектами он становится почти обязательным. К примеру, разработка текстового технического задания, казалось бы, никак не

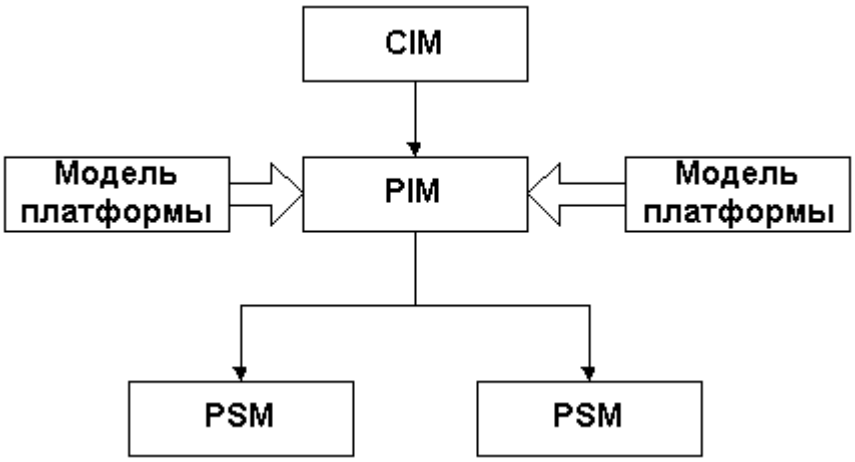
помогает собственно процессу программирования, зато, существенно способствует пониманию задачи в целом и позволяет избежать грубых ошибок проектирования в дальнейшем [2].

Результатом первого этапа разработки в соответствии с подходом MDA является описание PIM на языке UML. Завершенная платформно-независимая модель содержит полное описание системы, однако свободна от деталей, относящихся к реализации и используемым технологиям. После того, как модель построена, ее необходимо преобразовать в *платформно-зависимую модель* (platform-specific model, PSM). Это преобразование производится с помощью разработанных OMG *стандартных отображений*, разных для каждой платформы промежуточного слоя; при этом в модель вносится информация, относящаяся к деталям практической реализации и выбранной платформе. Благодаря тому, что преобразование от PIM к PSM стандартизовано, могут быть созданы инструменты — анализаторы и генераторы описания моделей, существенно упрощающие и автоматизирующие это преобразование. На данном этапе по-прежнему существенна работа дизайнеров и архитекторов системы: во многих случаях PIM предоставляет недостаточно данных для полностью автоматического переноса на конкретную технологию, и требуется предварительно разметить и конкретизировать PIM в терминах целевой системы. Например, UML-класс из платформно-независимой модели можно отобразить на интерфейс, объект или «тип-значение» (valuetype) при переходе к платформе CORBA. В таких случаях необходимо разметить модель вручную [3].

Второй этап

На втором этапе разрабатывается платформенно-независимая модель (PIM). Она может разрабатываться с нуля в случае отсутствия модели первого этапа или основываться на CIM. Преобразование CIM в PIM осуществляется на основе описания на зыке UML, созданного на первом этапе. Здесь в него добавляются элементы, описывающие бизнес-логику, общую структуру

системы, состав и взаимодействие подсистем, распределение функционала по элементам, общее описание и требования к пользовательскому интерфейсу. Модель PIM этого этапа обязательно включается во все автоматизированные среды разработки приложений на основе MDA [2].



Третий этап

На третьем этапе создаются платформенно-зависимые модели (PSM). Их число соответствует числу программных платформ, на которых будет функционировать приложение. Кроме этого, возможны случаи, когда приложение (или его составные части) должны работать на нескольких платформах одновременно. Модель PSM создается путем преобразования модели PIM с учетом требований модели платформы. Процесс преобразования описывается ниже. На этапе создания модели PSM разработка приложения согласно архитектуре MDA заканчивается. Считается, что правильно построенная PSM содержит техническую информацию, достаточную для генерации исходного кода (там, где это возможно) и необходимых ресурсов приложения. Здесь эстафетную палочку должна подхватить среда разработки, реализующая MDA. С формальной точки зрения это уже не относится к компетенции MDA, но для разработчика преобразование PSM в исполняемый код приложения непосредственное продолжение процесса разработки [2].

Завершенная платформно-зависимая модель содержит всю необходимую информацию для генерации кода системы, а также для генерации вспомогательного кода и описаний, необходимых для использования выбранной платформы и технологий (в частности, могут быть созданы IDL-описания интерфейсов для технологии CORBA). Фактически платформно-зависимая модель близка к UML-модели системы, полученной при классическом подходе к разработке, но более детальна и может содержать больше информации, относящейся к используемым технологиям [3].

Однако, архитектура MDA описывает еще один вариант прохождения третьего этапа, который называется ***прямым преобразованием в код***. Спецификация MDA для этого случая сообщает, что могут существовать инструментарии, напрямую преобразующие модель PIM в исполняемый код приложения. Модель PSM при этом может создаваться как контрольное описание, позволяющее проверить результат прямого преобразования [2].

Тогда этот этап проходит так же, как и при классическом подходе к разработке программного обеспечения с помощью UML-модели. Инструменты, предназначенные для генерации кода по UML-модели на языке программирования, существуют давно; эти наработки можно использовать при анализе и кодогенерации по PSM. После генерации кода производится его доработка, задается низкоуровневая функциональность системы и производится необходимая оптимизация. По завершении стадии кодирования можно произвести компиляцию, сборку и настройку системы [3].

Преобразование моделей PIM PSM

Наиболее сложным и ответственным этапом при разработке приложений в рамках архитектуры MDA является преобразование модели PIM в модель PSM. Именно на этом этапе общее описание системы на языке UML приобретает вид, пригодный для воплощения приложения на конкретной платформе. Как уже говорилось, в процессе принимает участие модель платформы. Преобразование моделей проходит три последовательные стадии:

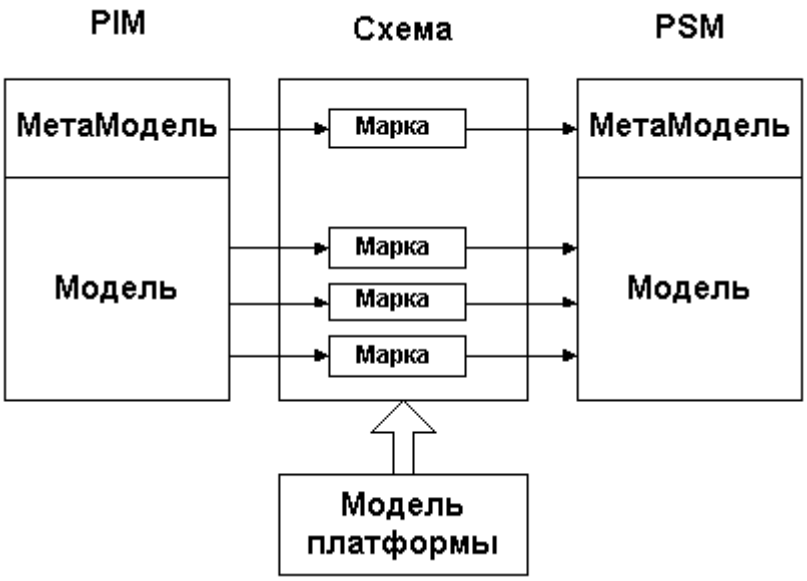
- Разработка схемы преобразования (mapping)
- Маркирование (marking)
- Собственно преобразование (transformation)

Рассмотрим их подробнее.

Первоначально необходимо разработать схему преобразования элементов модели PIM в элементы модели PSM. Для каждой платформы создается собственная схема преобразования, которая напрямую зависит от возможностей платформы. Схема преобразования затрагивает как содержание модели (совокупность элементов и их свойства), так и саму модель (метамодель, используемые типы). В схеме преобразования нужным типам модели, свойствам метамодели, элементам модели PIM ставятся в соответствие типы модели, свойства метамодели, элементы модели PSM. При преобразовании моделей может использоваться несколько схем преобразования.

Для связывания используются марки (mark) самостоятельные структуры данных, принадлежащие не моделям, а схемам преобразования и содержащие информацию о созданных связях. Наборы марок могут быть объединены в тематические шаблоны, которые возможно использовать в различных схемах преобразования. Процесс задания марок называется маркированием. В

простейшем случае один элемент модели PIM соединяется маркой с одним элементом модели PSM. В более сложных случаях один элемент модели PIM может иметь несколько марок из разных схем преобразования. Что касается преобразования метамодели, то в большинстве случаев марки могут расставляться автоматически. А вот для элементов модели часто требуется вмешательство разработчика. В процессе маркирования необходимо использовать сведения о платформе. Эти сведения содержатся в модели платформы.



Процесс преобразования моделей заключается в переносе маркированных элементов модели и метамодели PIM в модель и метамодель PSM. Процесс преобразования должен документироваться в виде карты переноса элементов модели и метамодели. Способ преобразования моделей может быть:

- Ручной
- С использованием профилей
- С настроенной схемой преобразования
- Автоматический [2]

Многоплатформенные модели

Архитектура MDA учитывает возможность разработки приложений, одновременно функционирующих на нескольких платформах. Для этого марки схемы преобразования моделей PIM PSM устанавливаются в соответствии с распределением функционала приложения по платформам. Затем генерируется несколько платформенно-зависимых частей приложения. Проблема взаимодействия частей такого гетерогенного приложения решается на уровне бизнес-логики приложения на этапе разработки. Для обмена данными могут использоваться специально разработанные подсистемы, использующие для организации обмена заранее согласованные механизмы, форматы данных, интерфейсы. Более того, разработка механизмов межплатформенного взаимодействия хорошо поддается автоматизации. Инструментарии MDA могут содержать функционал для создания таких механизмов [2].

Инструменты

После популяризации MDA многие производители стали утверждать, что их инструменты поддерживают MDA. На самом деле, поддерживаются лишт некоторые аспекты концепции. Проанализируем возможности существующих инструментов.

Поддержка преобразования

Инструменты для простой генерации кода из существующей модели существуют уже более десяти лет и неплохо вписываются в конецепцию MDA. Рассмотрим эти инструменты.

Инструменты преобразования PIM в PSM

Инструменты этой категории призваны трансформировать PIM в одну или более PSM. На данный момент таких инструментов не очень много, хотя некоторые из них все-таки обеспечивают минимальную функциональность.

Инструменты преобразования PSM в код

Большинство инструментов работают как «черный ящик». Они имеют встроенные правила преобразования и берут один тип модели как исходные данные и генерируют модуль другого типа как результат. Исходной является PSM, результирующей – код.

Несколько инструментов отражают связи между PSM и кодом. Они дают возможность увидеть изменения в обеих моделях сразу после изменения какой-либо из них. Это возможно, потому что PSM и код очень тесно связаны между собой. Они находятся примерно на одном уровне абстракции.

Инструменты преобразования PIM в код

Другие виды инструментов поддерживают два вида трансформации: из PIM в PSM, и из PSM в конечный код. Иногда пользователю показывается

только прямая трансформация из PIM в код. Трансформация из PIM в PSM опускается. В этом случае исходный и результирующий язык, а также определения трансформации встроены в инструмент, который опять же работает по принципу «черного ящика».

В качестве языка определения PIM обычно используется UML. Динамическая часть, не отражаемая средствами UML, обычно добавляется в сгенерированный код вручную.

Настраиваемые инструменты

Инструменты должны позволять параметризировать процесс преобразования. Обычно невозможно подстроить определения трансформации под свои требования, потому что доступ к ним закрыт. Лучшее на что можно рассчитывать – это правила преобразования, написанные на каком-либо внешнем скриптовом языке. Чтобы внести изменения в такой скрипт требуется много времени, потому что пока еще не создано специализированного для написания таких определений языка.

Большинство инструментов работают только с языком описания PIM, которым является UML. Хотя диаграммы UML используются для создания PIM, на практике инструменты не используют определения языка UML, заменяя их своими интерпретациями. Из-за этого даже специалист в области UML потратит много времени на изучение определений, вшитых в инструменты, для написания своих определений трансформации.

Инструменты задания правил преобразования

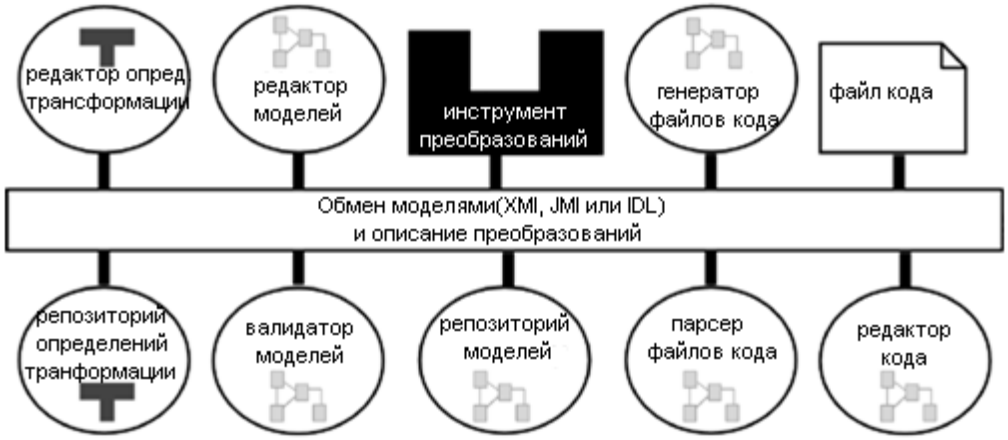
Эти инструменты обеспечивают создание и модификацию правил преобразования. Они нужны в случае, если разработчика не устраивают имеющиеся правила преобразования, и ему требуется написать свои собственные. Как было сказано ранее, такими средствами могут являться инструментально-специализированные скриптовые языки. Жесткая зависимость всей концепции MDA от определений трансформации делает

необходимым создание языка, способного четко описывать правила преобразования, и инструментов, лучше справляющихся с этой задачей. Такие инструменты еще не созданы.

Исходя из написанного, можно сделать вывод о том, что поскольку на данный момент не существует реальных инструментов моделирования правил преобразования, то и полезность всей концепции MDA оказывается под вопросом. На самом деле все не так плохо. Несмотря на то, что полный потенциал концепции еще не используется, даже существующие инструменты дают значительные приемущетсва.

Другие инструменты

Хотя ядром MDA являются инструменты преобразования, нужны еще и другие инструменты. Кроме функциональности, приносимой инструментами преобразования, требуется и другая.



- Редактор кода: набор функций, представляемых Interactive Development Environment(IDE), например, отладка, компиляция, форматирование кода.
- Файлы кода: хотя мы рассматривали код в качестве модели, обычно он хранится в текстовых файлах. Это не тот формат, который может быть

понят остальными инструментами. Поэтому, требуются следующие две вещи:

- Парсер файла кода: парсер, который преобразует текстовый файл с кодом в форму модели, которую могут использовать другие инстументы.
- Генератор файлов кода: отражение парсера. Работает наоборот. Преобразует модель кода в текстовые файлы.

- Репозиторий моделей: база данных моделей, где хранятся модели и могут быть получены, используя XMI, JMI или IDL.
- Редактор моделей: инструмент для создания и редактирования моделей.
- Валидатор моделей: модели, на основе которых генерируются другие модели, должны быть очень четко-определенными. Валидатор проверяет модели на соответствие некоторым правилам построения моделей, для того, чтобы она была применима для трансформации.
- Редактор определений трансформации: редактор для создания и редактирования опредеоений трансформации.
- Репозиторий определений трансформации: хранилище созданных определений трансформации.

Множесто инструментов на сегодняшний день в некоторой степени совмещают несколько вышеописанных функций. Традиционные CASE-инструменты предоставляют средства для редактирования и хранения моделей (редактор и репозиторий). Генератор кода, основанный на скриптовом языке и встроенный в CASE-инструмент, является инструментов трансформации и редактором определений трансформации. В таком случае, репозиторием определений трансформации является система текстовых файлов.

Все функций могут идти в двух формах: применительно к языку или в общем. Инструмент, рассчитанный на конкретный язык, может включать в себя редактор моделей для UML и генератор кода из UML модели в код C#. Общий редактор моделей позволяет пользователю редактировать любую модель.

Выбор инструментов

Изложенная выше информация помогает в выборе нужных инструментов для разработки по концепции MDA. Для начала нужно определиться с требованиями. Нужно ли привязываться к конкретному языку или требуется универсальный механизм? Нужно ли комбинировать различные инструменты, используя различные стандартные интерфейсы, или ограничиться одним, который предоставляет весь нужный функционал?

При выборе инструмента нужно выяснить, какие функции он поддерживает и на какой язык ориентирован. Так же необходимо проверить, работают ли функции на моделях, описанных с использованием стандартных механизмов (XMI, JMI, IDL). После этих манипуляций можно составить довольно четкое впечатление о выбранном инструменте. Не существует средств, предоставляющих полную функциональность в этой области. Поэтому следует быть готовыми комбинировать несколько различных инструментов.

Хотя процесс трансформации моделей является ключевым в MDA, нужно также озаботиться инструментами, выполняющими другие важные функции. При комбинировании инструментов главное – это разобраться, какие функции поддерживает каждый из них [12].

Достоинства MDA

MDA предоставляет архитектуру, которая обладает рядом ключевых достоинств:

- Переносимость, нарастающее повторное использование приложения, уменьшение стоимости и сложности разработки и управления приложением в настоящее время и в будущем.
- Строгие методы гарантии того, что системы, базируемые на различных технологиях реализации, соответствуют общей бизнес-логике и требованиям.
- Независимость от платформы, значительное сокращение времени, стоимости и сложности, связанной с переработкой приложений для различных платформ и сменой платформ.
- Настройка на предметную область посредством специфических моделей, которые позволяют быстро реализовывать новые приложения, используя стандартные для данной области компоненты.
- Возможность для разработчиков, дизайнеров и системных администраторов использовать удобные им языки и концепции; бесшовное связывание и интегрирование фрагментов, разрабатываемых разными командами.

Пользуясь MDA, можно организовать не только переход от абстрактной модели к детальной (от PIM к PSM, от PSM к коду системы), но и обратное преобразование — повышение уровня абстракции. Возможно создание инструментов, позволяющих осуществлять не только прямое, но и обратное преобразование моделей на основе стандартных отображений. Благодаря этому открывается возможность вести разработку, тестирование и модификацию одновременно платформно-независимой и платформно-зависимой моделей; если возникает необходимость изменить логику работы

программы на абстрактном уровне (т.е. изменить PIM), эти изменения могут быть отображены в изменения PSM [3].

Преимущества, которые дает архитектура MDA:

- Кардинальное повышение производительности разработки (устраняется этап «ручного» программирования).
- Документированность и легкость сопровождения.
- Централизация логики функционирования.
- Облегчение доступности и управляемости разработки (UML-диаграммы, представленные в графическом виде, являются достаточно наглядными и по существу не требуют знания программирования или теоретических основ разработки реляционных баз данных) [5].

Рассмотрим более подробно преимущества, предоставляемые концепцией MDA [12].

Производительность

В MDA разработчик делает акцент на разработку PIM, поскольку генерация PSM происходит автоматически. Конечно, все еще необходимо определить точное преобразование, что является трудной и узкоспециализированной задачей. Однако такое преобразование определяется только однажды и многократно может быть применено в дальнейшем. С точки зрения затраченных усилий и времени окупаемость определения преобразования является хорошей, но решение этой задачи под силу только высококвалифицированному специалисту.

Специалист, занимающийся проектированием PIM, может работать, не задумываясь о деталях и специфических особенностях целевых платформ и множестве других технических деталях. Они будут автоматически добавлены во время преобразования PIM в PSM. Это способно улучшить производительность по двум причинам:

- во-первых, объем работ проектировщиков PIM сокращается, потому что отпадает надобность описывать специфичные для платформы детали. Объем кода, требующего ручное написание, на уровне PSM так же сокращается, потому что большая часть кода генерируется автоматически;
- во-вторых, разработчики могут сконцентрироваться на создании PIM, таким образом, уделяя больше внимания решению именно бизнес-проблемы, а не написанию кода. Это позволяет быстрее разрабатывать системы, намного больше соответствующие требованиям конечных пользователей. Кроме того, такие системы более устойчивы к ошибкам, так как они более детально промоделированы.

Такое повышение эффективности может быть достигнуто только с помощью инструментов, которые полностью автоматизируют процесс генерации PSM из PIM. Подразумевается, что большая часть информации о системе должна быть включена в PIM или инструмент генерации. Высокоуровневая модель перестает быть «просто бумагой», как это было раньше. Теперь она непосредственно связана с кодом. Из-за этой связи к PIM предъявляются более высокие требования. Она должна быть законченной, непротиворечивой, полной и абсолютно точной. Человек, анализирующий неточную высокоуровневую модель, все-таки может правильно понять неполностью или не совсем верно отраженные в ней аспекты. Инструмент автоматической генерации – нет.

Мобильность

Как было сказано ранее, основное внимание уделяется разработке PIM. По определению, PIM – платформо-независимая модель. Из-за непривязанности основной модели к какой-либо платформе достигается высокая мобильность. Одна и та же PIM может быть автоматически преобразована во множество PSM, каждая из которых генерируется для

различных платформ. Поэтому, все что определяется на уровне PIM обладает абсолютной мобильностью.

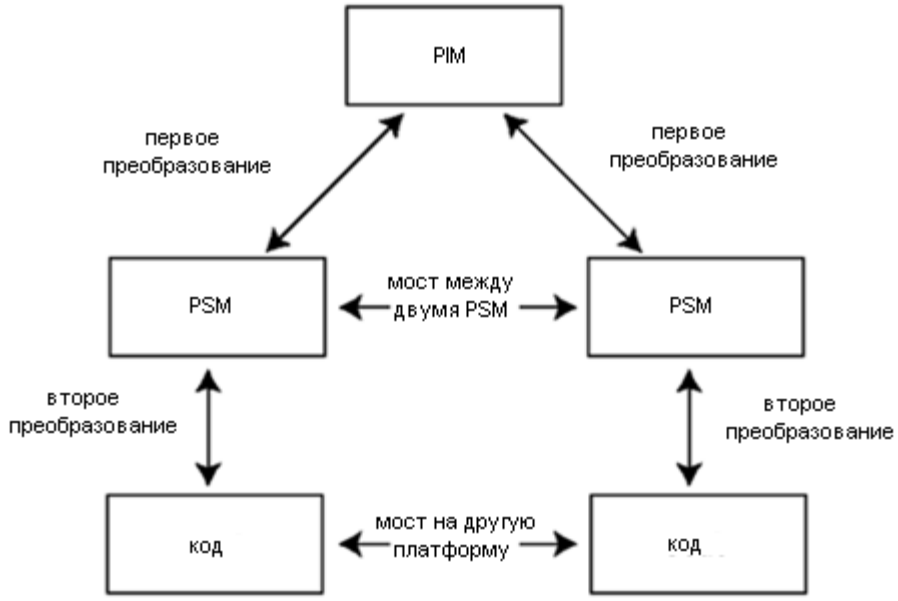
Степень достигаемой мобильности зависит от имеющихся инструментов автоматического преобразования. Для популярных платформ уже доступно (или станет доступно в ближайшем будущем) множество таких инструментов. Для менее популярных платформ, вероятно, придется использовать инструмент, который работает по система Plug-in, то имеет возможность работать по изменяемым правилам преобразования. В этом случае сами правила, естественно, должны быть написаны вручную.

Вероятно, в будущем, в случае успеха концепции MDA, при разработке новых платформ для них будут параллельно разрабатываться и правила преобразования. Это позволит быстро развернуть новые системы с новой технологией, базируясь на ранее спроектированных PIM.

Функциональная совместимость

Между различными PSM, сгенерированными от одного PIM, могут существовать отношения. Такие отношения называются мостами. Если PSM предназначены для разных платформ, они не имеют возможности непосредственно взаимодействовать друг с другом. Так или иначе, необходимо трансформировать понятия с одной платформы в понятия, используемые в другой платформе. Под этим и понимают функциональную совместимость. MDA решает эту проблему, генерируя не только PSM для различных платформ, но и необходимые мосты между ними. Если имеется возможность получения двух различных PSM из одной PIM, то, по идее, вся информация, нужная для обеспечения взаимодействия между ними, доступна. Для каждого элемента в одном PSM мы знаем, от которого элемента в PIM он был сгенерирован. Также мы знаем, что от этого же элемента PIM был сгенерирован соответствующий элемент во второй PSM. Исходя из этого, можно сделать вывод о соответствии друг другу элементов различных PSM. Кроме того, так как нам известны все технические детали

платформ обеих PSM (иначе, мы, возможно, не выполнили преобразования PIM к PSM), у нас есть вся информация для генерации моста между этими двумя моделями.



Возьмем, например, две PSM: первую, ориентированную на Java и вторую - на модель реляционной базы данных. В PIM определен элемент «Клиент». Разработчик знает, к какому Java-классу должен быть преобразован этот элемент. Он также имеет представление о таблице в базе данных, которая должна соответствовать этому элементу. Создание моста между Java-классом и таблицей в базе данных не составит особого труда. Для получения объекта из базы данных, мы запрашиваем таблицу, полученную от элемента «Клиент», и инициализируем Java-класс. Чтобы сохранить объект, мы берем информацию из Java-класса и записываем ее в таблицу «Клиент».

Межплатформенная функциональная совместимость так же может быть реализована с помощью инструментов, которые генерируют не только сами PSM, но и мосты между ними. Так же могут быть сгенерированы мосты на другие платформы. Таким образом, внезапная смена платформы не приведет

к катастрофическим последствиям и не сделает необходимым полное переписывание системы.

Документация

PIM является абстракцией гораздо более высокого уровня, чем код. PIM используется, чтобы генерировать PSM, которые поочередно используются для генерации код. Модель - точное представление кода. Таким образом, PIM выполняет функцию высокоуровневой документации, которая необходима для любой программной системы.

Разница в том, что PIM не «забрасывают» после первого написания. Чтобы внести какое-либо изменение в систему, разработчик должен изменить PIM. Далее по цепочке изменяются PSM и код. На все эти изменения уходит очень мало времени. В идеале должны быть разработаны инструменты, способные поддерживать обратную связь между PSM и PIM. При изменении PSM они должны автоматически отражать эти изменения на PIM.

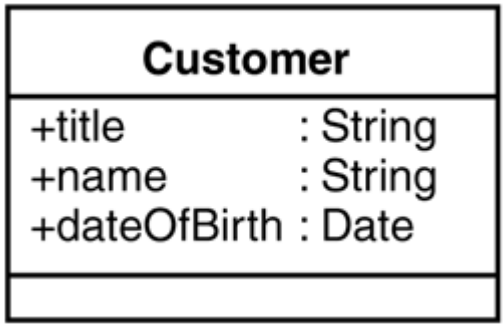
Конечно, кроме PIM остается доступной и обычная, традиционная документация. Разработчикам всегда может понадобиться задокументировать какую-либо дополнительную информацию о системе, которая была бы лишней в PIM. Примером такого дополнения может служить аргументирование корректности разработанной PIM [13].

Примеры

Рассмотрим два маленьких примера применения концепции MDA. Сами примеры не сложны, даже в каком-то смысле тривиальны. Они не показывают преимущества MDA в полной мере. Их задача – показать, как применять концепцию MDA на конкретных примерах. В обоих примерах акцент делается на высоко-уровневую PIM, созданную на UML, и на PSM более низкого уровня.

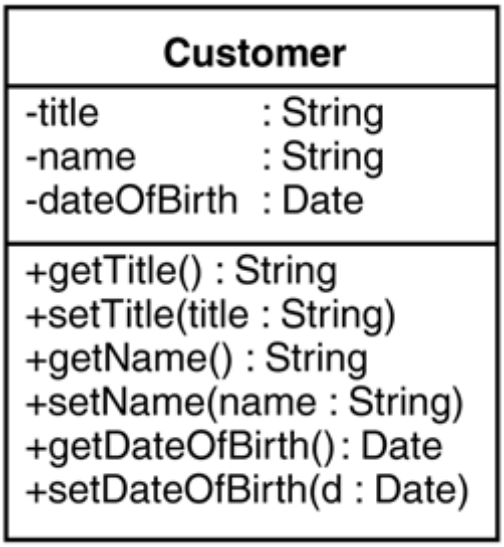
Публичные и частные атрибуты

В первом примере рассматривается преобразование между двумя UML моделями. PIM трансформируется в PSM, рассчитанную на Java. Главное в этом примере - преобразование публичных атрибутов в соответствующие им get- и set- методы. Класс «Клиент» содержит три атрибута: название, имя и дата рождения. Все атрибуты публичные. В высоко-уровневой PIM можно использовать публичные атрибуты. Они лишь означают, что элемент имеет какие-то атрибуты и они могут быть изменены впоследствии.



В PSM мы моделируем исходный код вместо бизнес-концепции. Использование публичных атрибутов считается плохим тоном. инкапсулировать публичные атрибуты, а изменять и получать их через специальные методы.

Все атрибуты частные и доступ к «Клиенту» осуществляется через четко-определенные операции. Это позволяет только объекту «Клиент» управлять своим использованием и изменением атрибутов.



Обе модели полезны, так как представляют разные уровни информации для разных разработчиков. Однако между этими моделями есть явная связь. Описание трансформации из PIM в PSM включает несколько правил:

- Для каждого класса с именем className в PIM должен существовать порожденный класс с таким же именем в PSM
- Для каждого атрибута с именем attributeName из класса в PIM порожденный класс в конечной модели должен включать:
  - частные атрибуты с такими же именами;
  - публичный метод с именем атрибута и приставкой get. Тип возвращаемого значения соответствует типу атрибута;
  - публичная операция с именем атрибута и приставкой set. Возвращаемое значение отсутствует. Тип параметра соответствует типу атрибута.

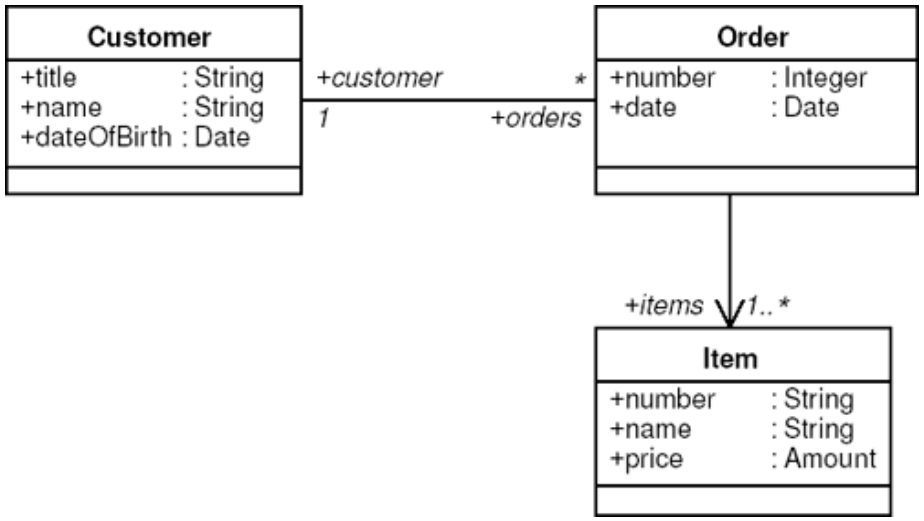
У этого правила преобразования есть отражение - правило, которое преобразовывает аспект PSM в PIM:

- Для каждого класса с именем className в PSM есть класс-пророди тель с таким же именем в PIM

Так как в этом примере мы знаем, что конечной платформой является Java, мы сразу можем описать правила преобразования, по которым PSM будет преобразована в код. В конечном итоге, мы получим исходный код из PIM, применив две трансформации: из PIM в PSM и из PSM в исходный код.

Связи

Можно расширить рассмотренный пример большим количеством классов и ввести связи между ними.



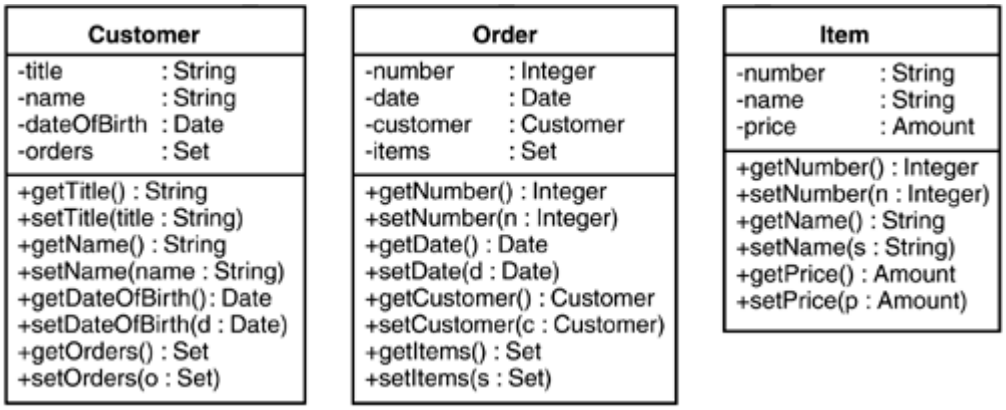
На рисунке добавлены два класса «Заказ» и «Предмет». Модель показывает, что «Клиент» может иметь несколько «Заказов», и к каждому заказу могут относиться несколько «Предметов». В PSM после преобразования публичных атрибутов нужно заменить связи между указанными классами чем-либо, присутствующем в конечном языке программирования. Естественно, для того, чтобы отобразить ассоциации с размерностью больше единицы, приходится использовать набор классов. Дополнительные правила преобразования:

- Для связи в PIM должно быть доступно следующее в PSM:

- В классах на обоих концах связи должен присутствовать частный атрибут, с именем другого класса, участвующего в связи;
- Тип этого атрибута - класс со стороны конца связи, если их несколько - набор классов;
- для добавленного атрибута должны быть определены методы set и get.

- Для направленной связи вышеупомянутое справедливо только для класса, из которого выходит стрелка.

Преобразование, которое содержит правила преобразования, как для атрибутов, так и для связей в PSM, показано на следующем рисунке:



Здесь PSM более отличается от PIM, чем в первом примере. Сложнее понять связи между классами. Если в нашем распоряжении есть только PSM, то можно видеть, что «Заказ» связан с «Клиентом» по полю «Клиент». Размерность связи может равняться нулю или единице. Мы не сможем понять, направленная это связь или нет. Связь между «Заказом» и «Предметом» потеряна полностью, потому что в Java нет ограничений на тип элемента в функции setItems класса «Заказ», а класс «Предмет» не содержит ссылки на «Заказ». Определение обратной трансформации становится невозможным.

В вышеупомянутом правиле было выбрано наличие операций присвоения для атрибутов. В этом случае тип параметра – множество. Например, метод setOrders (o: Set) в классе «Клиент». Альтернативой является только ввод операции addOrder (o: Set). Это облегчит добавление единичных заказов. Выбор делается человеком или самим инструментом.

Часто разработчику хочется иметь полностью контролировать преобразование. Например ему хочется заменить тип Set на HashSet и ввести обе операции: setOrders и addOrder. В этом случае ему придется вручную менять правила преобразования [14].

Перспективы развития MDA

Если развитие MDA будет успешно продолжаться, то, возможно, через несколько лет этап написания кода системы полностью выпадет из процесса разработки. Внимание будет уделяться описанию моделей и правил преобразования между ними.

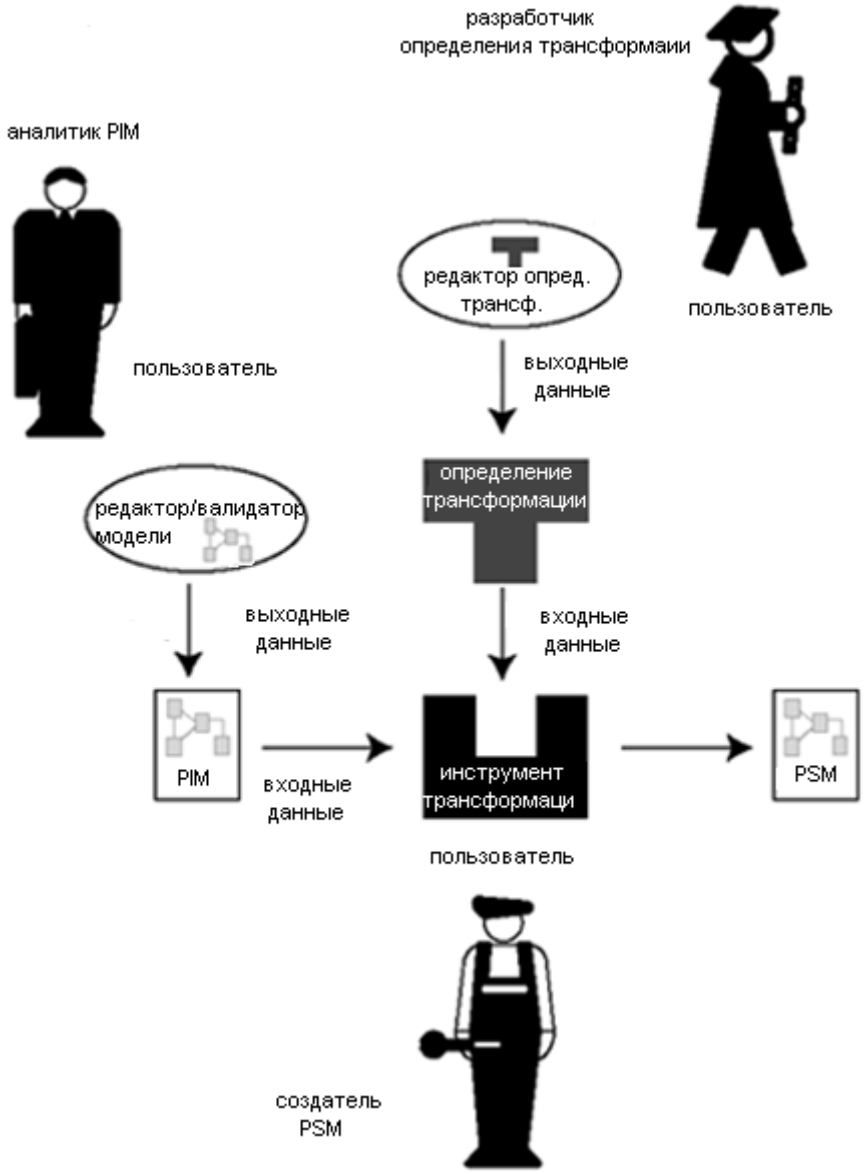
На самом ли деле MDA так хорош? Действительно, тот факт, что полная версия рабочей программы автоматически генерируется из модели, вызывает определенный скептицизм. Но подход MDA уже сегодняшней ступени развития может обеспечить преимущества в мобильности, производительности и платформенном взаимодействии. Кроме того, если обратиться к истории, в 60-х годах, при переходе с ассемблерных языков на процедурные многие специалисты так же скептически относились к этому нововведению. Они объясняли это несовершенством тогдашних компиляторов и рядом других причин, действительно имевших место в то время. Однако, несмотря ни на что, переход на процедурные языки произвел революцию в программировании. Возможно, то же ждет и MDA [15].

Процесс разработки

При сравнении процесса разработки с применением MDA и без можно заметить очень много общего. Все равно остаются этапы сбора требований, тестирования и внедрения. Меняются же этапы анализа, низко-уровневого дизайна и написания кода.

Во время анализа должна быть разработана PIM. Эта задача, скорее всего, будет поручаться небольшой группе высококвалифицированных специалистов. Другая группа людей будет ответственна за преобразование PIM в PSM. В эту группу будут входить люди, обладающие обширными знаниями о различных платформах и технологиях, архитектурах разнообразных систем и правил преобразования, предоставляемых существующими инструментами. Они будут принимать решение

относительно архитектуры конечной системы. Мнение разработчиков PIM будет им более полезно, чем сама PIM. Например, если разработчик PIM своевременно доведет до создателя PSM информацию о том, что система будет использоваться тысячами пользователей, создатель PSM выберет подходящую для этого платформу.



Другой задачей создателя PSM будет являться реагирование на изменения PIM или правил преобразования. Модель и правила могут меняться независимо друг от друга. При изменении бизнес требований к системе затрагивается только PIM. А при изменении платформы меняются

только правила преобразования. На эти изменения нужно своевременно среагировать.

Поскольку создателю PSM будут нужны правила преобразования, то процесс разработки расширится этапом написания таких правил. Этим будет заниматься еще одна группа специалистов. Такие люди частично будут нужны компаниям, производящим софт, но в основном они буду пользоваться спросом у поставщиков инструментов преобразования. Примерная схема MDA-процесса с указанием участников, инструментов и артефактов показана на следующем рисунке [16].

Заключение

Архитектура MDA возникла не на пустом месте. Само ее появление и возможность реализации обусловило наличие ряда стандартов и технологий, на практике доказавших свою полезность. Концептуальной основой появления MDA стали спецификации OMA, ORB, CORBA. Перевести замысел в практическую плоскость позволили технологии объектно-ориентированного программирования (ООП), стандарт CWM, языки UML, XML, MOF. Работами по созданию новой архитектуры программирования занялся консорциум OMG (Object Management Group).

По мнению создателей, архитектура MDA является новым витком эволюции технологий программирования, так как описывает процесс разработки в целом. Новизна MDA заключается в том, что описание процесса разработки в ней выполнено с использованием современных средств представления и позволяет автоматизировать создание приложений. И весьма вероятно, что через некоторое время архитектура MDA станет общим промышленным стандартом в разработке программного обеспечения [2].

На данном этапе эта методология находится больше в области академических исследований, чем практического применения. Хотя множество деталей еще требуют уточнений и разработки и потребуются еще годы работы перед тем, как мы увидим практические применения MDA, общее видение и направление развития уже достаточно хорошо определены. Тем не менее, многие считают идею генерации работающих систем автоматически из моделей утопической [5].

Список литературы

1. Куриленко И.Е., Борисов А.В. Современные архитектурные подходы к построению программного обеспечения // Сб. тр. XVIII междунар. науч.–техн. конф. Информационные средства и технологии .–Т.2. – М.:Издательский дом МЭИ, 2010. - С.176-184.
2. http://citforum.ru/gazeta/13/
3. http://www.osp.ru/os/2003/09/183391/
4. http://www.winsov.ru/xml005.php
5. http://www.rusnauka.com/10_NPE_2011/Informatica/3_82746.doc.htm
6. Understanding the Model Driven Architecture (MDA) Sinan Si Alhir
7. http://ru.wikipedia.org/wiki/Common_Information_Model
8. http://www.developers.org.ua/lenta/articles/mda-introduction/
9. Model Driven Architecture by Richard Soley and the OMG Staff Strategy Group
10. http://www.omg.org
11. http://www.wikipedia.org
12. Frankel, David. Model Driven Architecture: Applying MDA to Enterprise Computing. New York: John Wiley & Sons, 2003.
13. Anneke Kleppe. MDA Explained: The Model Driven Architecture™: Practice and Promise
14. Константин Грибачев. Delphi и Model Driven Architecture: Разработка приложений баз данных
15. Model Driven Architecture by Richard Soley and the OMG Staff Strategy Group
16. Understanding the Model Driven Architecture (MDA) Sinan Si Alhir