Санкт-Петербургский Государственный Университет
Математико-механический факультет

Кафедра системного программирования

Механизм автоматической генерации мигратора базы данных информационной системы при изменениях модели предметной области

Курсовая работа студента 445 группы
Михайлова Дмитрия Петровича

Научный руководитель Нестеров А.В.

Санкт-Петербург
2012

Оглавление

Введение

Разработка систем, предназначенных для сбора, обработки и хранения информации, всегда была и остаётся одним из главных сегментов рынка разработки программного обеспечения. Его стремительное развитие привело к возникновению разнообразных средств автоматизации процесса разработки. Одним из них стало использование объектно-ориентированного визуального моделирования с использованием CASE-средств для описания моделей системы.

Такие CASE-средства часто содержат инструменты для генерации реляционных баз данных, но не многие из них позволяют автоматически генерировать графический пользовательский интерфейс для взаимодействия с ними. Возможно, это связано с тем, что необходимый интерфейс в значительной степени зависит от требований конечного пользователя.

Несмотря на это, в ходе переноса технологии REAL-IT на платформу Microsoft .NET Framework была выделена общая для всех таких приложений функциональность и разработаны реализующие её библиотеки [1].

С помощью REAL-IT/.NET был разработан ряд промышленных проектов, работа над которыми выявила некоторые недостатки существующего подхода к генерации информационных систем.

1 Постановка задачи

Во время разработки и поддержки информационных систем, автоматически сгенерированных при помощи технологии REAL-IT/.NET, иногда происходят уточнения модели предметной области. При этом для осуществления соответствующих изменений графического пользовательского интерфейса требуется загрузить описание интерфейса из XML-файла, внести в него изменения с помощью REAL-IT и сгенерировать новые формы и реляционную базу данных.

Каждый раз при выполнении этой операции с уже установленной и работающей информационной системой возникает проблема переноса данных из старой базы данных в только что сгенерированную, схема которой может отличаться. Написание кода с такой функциональностью вручную возможно, но требует значительного времени, которое может быть ещё больше увеличено при повреждении данных вследствие ошибки. Возможность и удобноство внесения правок в сгенерированный код является одним из достоинств REAL-IT как CASE-системы, но в данной ситуации оказывается необходимой систематическая реализация однотипной функциональности вручную.

Задачей данной работы является разработка механизма, который позволил бы сгенерировать код, переносящий данные, автоматически. Логически она может быть разделена на две части:

1. Разработка механизма сравнения двух версий модели предметной области, который позволил бы обнаружить изменения в ней и запросить у пользователя ожидаемую реакцию на каждое из них.
2. Реализация генератора кода, который мог бы сгенерировать мигратор между базами данных, применяющий к данным преобразования, соотвествующие обнаруженным и подтверждённым изменениям, а также переносящий без изменений данные, не требующие преобразований.

2 Реализация

Процесс генерации мигратора можно разбить на несколько последовательных этапов, каждый из которых представлял отдельную подзадачу реализации.

2.1 Загрузка старой версии модели

Одной из особенностей последней версии технологии REAL-IT/.NET является возможность использовать для графического моделирования предметной области встроенные средства интегрированной среды разработки Microsoft Visual Studio 2010. При этом из созданной модели автоматически генерируются описанные на языке C# классы, соответствующие сущностям предметной области, которые затем компилируются в Class Library DLL, что позволяет хранить описание модели в стандартном для платформы формате.

Существовавшее в REAL-IT/.NET средство загрузки модели при помощи рефлексии и построения её внутреннего представления использовало для определения пути к ней настройки приложения (System.Configuration.ConfigurationSettings.AppSettings), поэтому первой частью работы стала реализация возможности загружать вторую модель из произвольного .dll-файла, не меняя выбранных путей для генерации кода.

2.2 Сравнение двух версий модели

В общем случае модель представляет собой ориентированный граф, а результат сравнения двух произвольных графов не предоставляет возможности представить переход от одного к другому как композицию простых преобразований. В связи с этим при разработке механизма сравнения версий модели было принято решение считать верным предположение о том, что все возможные изменения модели локальны (не пересекаются по наборам вершин, которые затрагивают) и принадлежат небольшому набору разновидностей атомарных преобразований.

В рамках этого предположения сравнение версий модели осуществлялось по следующему алгоритму.

Атомарные преобразования хранятся в списке, который изначально пуст. Для каждой из двух версий модели хранится динамическое множество классов, не участвующих ни в каких нетривиальных преобразованиях и ещё имеющих возможность в каком-то из них поучаствовать, а в случае классов из старой версии — и перенестись в новую без изменений. Назовём эти классы «свободными». В начале работы для каждой модели «свободны» все классы, присутствующие в ней.

Просматривается полный изначальный набор классов старой версии модели, те из них, которые к моменту, когда до них дошла очередь, уже не «свободны», игнорируются, поскольку класс не может участвовать в двух преобразованиях из предположения об их локальности.

Если класс оказывается «свободен», в новой версии модели проверяется наличие соответствующего «свободного» класса (наличие соответствующего класса «несвободным» означает, что произошло изменение, не являющееся одним из известных атомарных, и о нём следует оповестить пользователя).

В случае отсутствия соответствующего нового класса, пользователю предлагается выбрать одно из двух возможных атомарных преобразований, которые могли к этому привести: разделение на несколько новых классов (в этом случае старый и все новые классы удаляются из «свободных») или слияние с другим старым классом в один новый (тогда из «свободных» удаляются все три), - а также отказаться от каких-либо действий в его отношении. Атомарное преобразование, если оно было выбрано, добавляется в список.

Если соответствующий новый класс существует и «свободен», для каждого скалярного поля старого класса и каждой его ссылки на другой класс проверяется, что соответствующее поле или ссылка есть и в новом классе. В случае, когда какой-то из них нет, пользователю

предлагается задать новый класс как функцию старого. Если он соглашается, оба класса удаляются из «свободных» и преобразование добавляется в список.

Если все старые поля находятся в новом классе, просматриваются ссылки из нового класса. В случае, когда среди них есть ссылка на класс, соответствующего которому старого не существует, пользователю предлагается задать оба новых класса как функцию старого. При согласии преобразование добавляется в список.

Таким образом, существует четыре типа атомарных преобразований:

1. Разделение старого класса на несколько новых;
2. Слияние двух старых классов в один новый;
3. Преобразование старого класса в соответствующий ему новый с изменением набора полей;
4. Добавление к старому классу ссылки на ранее не существовавший класс с сохранением всех старых полей и ссылок.

После окончания выполнения алгоритма старые классы, которых нет в множестве «свободных», больше не могут отображаться в старых формах графического пользовательского интерфейса, поэтому, в зависимости от того, существует соответствующий им новый класс или нет, соответствующие им формы должны быть либо перегенерированы, либо удалены.

2.3 Генерация мигратора

Процесс генерации кода мигратора тоже происходит в несколько последовательных этапов.

2.3.1 Шаблон мигратора

Генерация кода мигратора начинается с файла с шаблоном, который содержит код запроса пути к новой базе данных, в которую нужно переместить данные, а также код соединения с двумя базами. Помимо этого, в нём реализованы методы для обращения к таблицам и для их копирования без изменений.

Также шаблон содержит метод Migrate(), который после замены всех плейсхолдеров в его теле становится методом, который осуществляет все необходимые операции по переносу данных из одной базы в другую.

2.3.2 Копирование не изменённых таблиц

Для каждого старого класса, который к концу сравнения остался «свободным», генерируется вызов метода по переносу соответствующей ему таблицы из старой базы в новую.

Список таких вызовов помещается в тело метода Migrate() вместо плейсхолдера.

2.3.3 Генерация преобразований

Все четыре класса, описывающих атомарные преобразования, являются наследниками общего предка, который содержит ссылки на объект, осуществлявший сравнение, и класс, при просмотре которого преобразование было сгенерировано, а также виртуальный метод Apply(), который генерирует из соответствующего шаблона утилитный класс со статическими методами конкретно для этого преобразования и возвращает строку с вызовом одного из этих методов, который полностью осуществляет перенос данных из таблицы в таблицу с промежуточным преобразованием.

Список из полученных таким образом строк подставляется вместо второго плейсхолдера в шаблон мигратора, после чего его генерация завершена.

Необходимо отметить, что, хоть в коде сгенерированных статических методов и производятся простейшие операции с данными (например, копирование значений из полей старого класса в соответствующие поля нового), предполагается, что код менее тривиальных преобразований будет дописан туда вручную, для чего в теле метода выделено (помечено комментарием) место.

Заключение

Результаты

На данный момент разработаны:

- Пустой проект мигратора;
- Шаблоны кода мигратора;
- Генератор кода.

Существующее решение доказывает возможность автоматической генерации мигратора между базами данных с разными схемами при не слишком больших различиях между ними, даже если изначально набор осуществлённых изменений модели данных не известен.

Генератор кода и автоматически сгенерированный мигратор были опробированы на примере промышленной информационной системы и её базы данных.

В качестве недостатков текущей версии необходимо отметить очень небольшой набор обрабатываемых изменений модели предметной области, который впоследствии может быть значительно расширен или пересмотрен.

Дальнейшее развитие

Данная работа может быть продолжена по следующим направлениям:

1. Пересмотр или расширение списка обрабатываемых атомарных изменений с целью лучшего соответствия решения реально возникающим задачам.
2. Тестирование другого подхода к сравнению версий модели (к примеру, возможно редактирование модели из REAL-IT, которое позволит сохранить пошаговый набор изменений, даже если они затрагивают одну и ту же вершину графа модели).
3. Интеграция с системами контроля версий с целью автоматического определения изменений модели предметной области.
4. Разработка механизма автоматической генерации изменений в коде графического пользовательского интерфейса на основе существующего механизма сравнения моделей предметной области.

Список литературы

1. А. В. Нестеров. Перенос технологии REAL-IT на платформу Microsoft .Net. 2007
2. А. Н. Терехов, К. Ю. Романовский, Д. В. Кознов, П. С. Долгов, А. Н. Иванов. Real: методология и CASE-средство для разработки систем реального времени и информационных систем. Программирование, 1999, N 5.
3. А. Н. Иванов. Автоматизированная генерация информационных систем, ориентированных на данные.
4. Peter Drayton, Ben Albahari, Ted Neward. C# in a Nutshell.
5. Джеффри Рихтер. CLR via C#.