Механизмы поддержки циклической разработки ИС в рамках модельно-ориентированного подхода

А.Н.Иванов
iw@tercom.ru
Санкт-Петербургский государственный университет
198504, Университетский пр., 28
Санкт-Петербург, Россия

Аннотация

В статье рассматриваются проблемы, возникающие при переходе между различными фазами жизненного цикла программных продуктов при итеративной разработке. Эти проблемы заключаются в том, что если на очередной итерации были внесены изменения на фазе $N-1$, то на фазе N необходимо совместить последствия этих изменений с артефактами фазы N, созданными в рамках предыдущей итерации. Выделяются две такие проблемы, возникающие при разработке информационных систем с использованием модельно-ориентированного подхода — необходимость поддержки "ручных" изменений кода при его перегенерации по высокоуровневым моделям и сохранение содержимого базы данных при обновлении ее схемы. Предлагаются различные подходы к решению указанных проблем, описываются конкретные механизмы, реализованные в технологическом решении REAL-IT и опробованные в ряде промышленных проектов.

Введение

Традиционно жизненный цикл программных продуктов принято разделять на фазы анализа, проектирования, реализации и эксплуатации [9]. На каждой фазе создаются свои артефакты[1] (документы, диаграммы, исходные коды и т.д.). Будем называть

[1]Под артефактами [1] понимаются все промежуточные продукты проекта, порождаемые или используемые в нем при работе над окончательным продуктом.

множество артефактов, полученных в результате выполнения какой-либо фазы жизненного цикла, *моделью* системы для этой фазы. В случае каскадного процесса разработки модель каждой следующей фазы строится на основе модели предыдущей. В том случае, если процесс разработки носит итеративный характер, на всех итерациях, кроме первой, при переходе к следующей фазе возникает задача построения ее модели на основе как модели предыдущей фазы, так и модели предыдущей итерации данной фазы. В том случае, когда переход между фазами осуществляется разработчиком "вручную", задача интеграции моделей целиком ложится на его плечи. Если же переход осуществляется с использованием автоматических или автоматизированных средств (например генераторов кода), то именно они и должны поддерживать эту интеграцию. Использование таких средств при переходе от одной фазы к другой мы и будем понимать под модельно-ориентированным подходом.

В предлагаемой работе рассматриваются два аспекта данной проблемы — учет "ручных" изменений кода, сгенерированного по моделям фазы проектирования (переход с фазы проектирования на фазу реализации) и сохранение информации в базе данных при ее обновлении (переход с фазы реализации на фазу эксплуатации). Выбор именно этих аспектов обусловлен следующими факторами:

- основным артефактом фазы реализации является исходный код разрабатываемой системы, который, частично или полностью, может быть сгенерирован по модели проектирования;
- при создании информационных систем именно содержание базы данных является основным артефактом фазы эксплуатации.

Использование генераторов кода является обычной практикой при создании ПО. Самым распространенным видом кодогенераторов на сегодняшний день являются компиляторы с языков высокого уровня — без них разработка современного ПО представляется немыслимой. В рамках модельно-ориентированного подхода предполагается использование генерации кода по моделям системы, построенным на фазах анализа

и проектирования — моделям предметной области, схемам основных алгоритмов и т.д. Примером данного подхода является генерация пользовательского интерфейса по моделям предметной области [21].

Мы будем называть модели анализа и проектирования высокоуровневыми, отличая их от моделей фазы реализации (реализационных), к которым относятся, в первую очередь, исходные тексты программ.

Для создания высокоуровневых моделей широко применяются языки визуального моделирования [6], позволяющие оформлять эти модели в виде диаграмм. Следует отметить, однако, что не все диаграммные модели являются высокоуровневыми в нашем смысле — они могут носить и реализационный характер, т.е. служить для визуализации исходного кода разрабатываемой системы. При этом они являются семантически полностью эквивалентными представляемому коду. Например, язык диаграмм описания схемы реляционной базы данных IDEF1x [13] позволяет специфицировать все особенности схемы данных, описываемые средствами языка SQL [16].

Высокоуровневые модели, в отличие от реализационных, используются для рассмотрения тех или иных свойств системы, абстрагируясь от деталей их реализации. Одни и те же виды диаграмм иногда можно рассматривать как высокоуровневые и как реализационные, в зависимости от степени детальности представленной на них информации. Например, SDL-диаграмма, в символах которой указан код на языке программирования, является реализационной, а та же диаграмма без кода — высокоуровневой [17].

Код, полученный путем непосредственной генерации по высокоуровневым моделям, содержит дополнительную семантику, вносимую генераторами (что, в частности, означает, что такие генераторы всегда нацелены на конкретный проект или класс систем). Однако для отдельных элементов системы эта семантика может быть неподходящей или недостаточной. В таком случае сгенерированный код требует “ручной доводки”. Однократная модификация кода может быть произведена программистом “вручную”, однако при итеративном характере процесса разработки появляется указанная выше задача автоматизации переноса “ручных” изменений в новую версию кода.

Другая проблема перехода между фазами возникает при изменении схемы базы данных в том случае, если база данных уже эксплуатируется и содержит реальные данные — эти данные нуждаются в переносе. Если уже было выпущено несколько версий разрабатываемого продукта и они были переданы большому количеству различных клиентов, то перенос информации должен быть осуществлен автоматически. Сложность этой задачи связана еще и с тем, что может быть неизвестной точная версия работающей у клиента базы данных. В то же время, к операции переноса данных предъявляются повышенные требования по надежности, поскольку потеря данных клиента при переходе на новую версию системы недопустима.

1 Обзор существующих подходов

В настоящий момент наиболее распространенными видами генераторов кода являются компиляторы, мастера (wizards) и генераторы, встроенные в различные CASE-пакеты.

В большинстве компиляторов есть возможность состыковывать сгенерированный код с модулями (библиотеками процедур, классов и т.д.), написанными непосредственно на целевом языке (языке, в который осуществляется трансляция). Для использования таких модулей требуется описать их сигнатуру в терминах исходного языка (языка, подающегося на вход транслятору). Кроме того, в ряде компиляторов (например Microsoft Visual C++ [8], Turbo Pascal [10] и др.) предусмотрена возможность использования “ассемблерных вставок” — добавление фрагментов кода на целевом языке непосредственно в код на исходном языке.

В отличие от компиляторов, исходная модель для которых — набор файлов — определяется до их запуска, мастера строят эту модель в диалоге с пользователем. После генерации построенная модель обычно либо забывается (как, например, в генераторах экранных форм, встроенных во многие средства разработки — например, ERwin [3] и Microsoft Access [12]), либо сохраняется в сгенерированном коде в форме специфических комментариев и при следующем запуске мастера восстанавливается из них — так работают, например, мастера создания классов (Class Wizards) в Microsoft Visual Studio [8]. Собственно генерация осуществляется

этими мастерами только один раз — при создании класса. При этом создается единая модель, хранящаяся в файлах с исходными текстами класса и представляющая собой программный код, размеченный специальными комментариями. Вся дальнейшая работа над классом с помощью мастера является модификацией этой модели.

Существует, однако, ряд мастеров, в которых исходная модель сохраняется, а генерация основывается на применении шаблона объектно-ориентированного проектирования "Generation Gap" [2]. Использование этого шаблона предполагает для каждого создаваемого модуля генерацию двух классов — предка, содержащего всю функциональность, и наследника, ничего не добавляющего к функциональности предка. При последующей перегенерации перегенерируется только класс-предок. Все "ручные" изменения вносятся в код наследника. Такой подход реализован, например, в системах ibuild [24], Forte [15], Orbix [22].

Большинство современных CASE-пакетов позволяет создавать как высокоуровневые, так и реализационные диаграммы. При этом связь с кодом поддерживается только для последних, связь же между диаграммами разного уровня обычно не поддерживается вообще. Так, например, создатели языка UML [20] не предоставляют четких рекомендаций по связыванию функционалистики системы, которую предлагается описывать на диаграммах случаев использования, с ее реализацией, осуществляемой классами и их членами. Соответственно, многочисленные CASE-пакеты, основанные на этом языке, не содержат автоматизированных средств для поддержки соответствия между высокоуровневыми и реализационными моделями. Поэтому данный обзор ограничивается рассмотрением связи реализационных диаграмм с исходным кодом. Эта связь будет рассмотрена на примере двух популярных объектно-ориентированных CASE-пакетов — Rational Rose и Together, использующих для этой цели различные подходы.

Механизм поддержки циклической разработки Rational Rose [19] наиболее полно реализован в расширении для C++ [18]. По модели классов осуществляется генерация скелета кода, т.е. кода без реализации методов. В случае изменения скелета в коде требуется построить на его основе новую модель в Rational Rose с помощью специальной утилиты реверс-инжиниринга. По-

сле этого с помощью специальной утилиты сравнения можно сравнить построенную модель с исходной и внести соответствующие изменения в исходную модель. При перегенерации методы, у которых есть тела, оставляются без изменений.

В Together [23] модель классов, по которой осуществляется генерация, синтаксически эквивалентна модели классов целевого языка (в данном случае — Java). Этим данный пакет отличается от предыдущего, в котором модель классов является универсальной и может содержать элементы, отсутствующие в языках программирования (например дискриминаторы наследования). При этом Together сохраняет только ту информацию, которая не влияет на семантику кода, а отвечает за его визуализацию. Файлы с исходными текстами, с точки зрения Together, являются элементом модели проекта, поэтому генерации как таковой не происходит.

Отсутствие возможности синхронизировать высокоуровневые диаграммы с кодом не позволяет говорить о поддержке итеративного модельно-ориентированного подхода в рассмотренных CASE-пакетах, а только об итеративном визуальном моделировании.

Вторая из рассматриваемых нами проблем — модификация структуры базы данных при необходимости сохранения ее содержимого в той или иной степени решается производителями CASE-пакетов, предназначенных для моделирования структуры базы данных. Наиболее полное, на наш взгляд, из этих решений представлено в пакете ERwin.

В ERwin имеется возможность сравнить созданную модель базы данных с реальной базой данных. При этом элементы схемы (таблицы, поля таблиц и т.д.) из модели сопоставляются с аналогичными в базе данных. Автоматически сопоставляются одноименные элементы. Пользователь может добавить новое сопоставление или удалить какое-то из существующих, после чего возможно сгенерировать SQL-скрипт, который будет содержать операторы изменения структуры базы данных (добавление, удаление и переименование элементов структуры), приводящий ее к новой модели.

В этом решении можно выделить две проблемы. Первая связана с тем, что каждый элемент модели может иметь набор дополнительных свойств, однако синхронизировать модель на

уровне значений отдельных свойств ERwin не может. Вторая проблема заключается в том, что существуют нетривиальные модификации базы, требующие "ручной" обработки — например, разделение одной таблицы на две или наоборот, слияние двух таблиц в одну. Подобные модификации требуют внесения в порожденный скрипт "ручных" изменений для их обработки. Кроме того, подобное решение предполагает, что программист непосредственно имеет дело с клиентской базой данных или с ее точной копией. Таким образом, не рассматривается возможность наличия баз данных разных версий и необходимости передачи клиенту приложения или скрипта, который сможет нужным образом модифицировать его базу данных, независимо от ее версии.

2 Постановка задачи

Модель, которая подается на вход модельно-ориентированным генераторам кода, не является исчерпывающим описанием системы (в отличие от текста на языке программирования, который подается на вход компилятора). Следовательно, "ручные" изменения сгенерированного кода или его дополнения неизбежны. С другой стороны, "ручное" сопровождение всего сгенерированного кода, оторванное от визуального моделирования, требует больших трудозатрат.

Сгенерированный код является композицией двух элементов — модели приложения, созданной разработчиком приложения, и архитектурного решения, созданного автором генератора. Оба эти элемента со временем могут изменяться: модель — вследствие уточнения знаний о предметной области, а также изменений в ней, а архитектурное решение — вследствие изменений в целевой архитектуре и среде программирования, выходе новых версий ПО, изменения требований ко всему классу разрабатываемых систем (например переход на Intranet-технологию). Кроме того, и модель, и генераторы могут содержать ошибки, которые необходимо исправлять. Таким образом, даже те фрагменты порожденного кода, которые не требуют "ручных" изменений, придется неоднократно приводить в соответствие с моделью и архитектурным решением.

Самым простым и надежным способом поддерживать это со-

ответствие является повторная генерация. При этом возникает противоречие между необходимостью внесения “ручных” изменений в порожденную программу и необходимостью периодически производить перегенерацию.

2.1 Возможные решения

Сформулируем возможные способы разрешения описанного выше противоречия.

1. Разделение сгенерированного кода на компоненты, одни из которых не требуют модификации, а другие подвергаются модификации в процессе доработки и сопровождения программного кода. Перегенерация используется только для компонент первого вида, компоненты второго вида сопровождаются “вручную”. В качестве компоненты при этом используется структурная единица кода, которую можно целиком заменить на другую (обычно файл).

2. Ведение учета “ручных” изменений сгенерированного кода и их повторное внесение после каждой перегенерации.

3. Выделение модификаций кода в отдельные компоненты с автоматическим добавлением их в итоговую программу. Основные способы осуществления такого добавления — это использование препроцессора целевого языка, директив сборщика (linker), специально написанных для этой цели программ (обычно скриптов) или препроцессоров (например фреймовых [14]), а при объектно-ориентированном подходе — использование полиморфизма (шаблон проектирования “Generation Gap”).

4. Расширение источников информации для кодогенерации — использование дополнительных моделей, расширение набора свойств элементов модели. При этом количество “ручных” изменений уменьшается за счет того, что специфичную семантику, которая вынуждает вносить эти изменения, удается внести в модель, используемую для генерации.

5. Выделение “ручных” изменений из кода системы путем автоматического анализа измененного текста программы и автоматическое внесение их при перегенерации.

2.2 Анализ возможных решений

Первый способ эффективен, если объем изменений внутри компоненты достаточно велик и трудно локализуем. Если же изменения локальны (например затрагивают одну строчку кода или одну процедуру), то затраты на их учет таким способом становятся слишком велики, а надежность — слишком мала. Все изменения кода, которые иначе были бы внесены автоматически путем перегенерации, в такой ситуации разработчику придется вносить "вручную".

Второй способ требует соблюдения строгой дисциплины внесения изменений. Разработчик должен не забывать как вносить информацию о любом изменении кода в реестр, так и восстанавливать эти изменения после каждой перегенерации. Такой процесс малонадежен, а трудозатраты и вероятность ошибки прямо пропорциональны количеству изменений.

Третий способ требует дополнительных затрат на создание стратегии внесения изменений на основе стандартных средств переиспользования кода. Подобная стратегия может потребовать изменений в архитектуре генерируемого приложения. К тому же, выбранная стратегия всегда накладывает дополнительные ограничения на изменения, вносимые в код, — они должны быть структурированы соответствующим образом. Кроме того, полное отсутствие "человеческого" надзора за слиянием новой версии порожденного кода с изменениями, выполненными для предыдущей версии, уменьшает надежность этого способа — после перегенерации код меняется, и старые изменения могут оказаться не нужны, неприменимы или требовать модификации.

Четвертый способ эффективен при наличии большого числа однородных изменений, семантику которых можно однозначно и формально описать. В отличие от остальных способов, этот направлен не на поддержку "ручных" изменений, а на их ликвидацию. Однако попытка с его помощью полностью избавиться от "ручных" изменений приводит к значительному усложнению модели и средств кодогенерации.

Пятый способ похож на третий, но предполагает, что разработчик может изменять по своему усмотрению сгенерированный код, а специальная программа определенным образом помечает эти изменения их при внесении с тем, чтобы после перегенерации их можно было автоматически повторно внести в код. Данный

подход требует дополнительного изучения — в настоящий момент в ряде промышленных средств разработки и версионного контроля имеются средства автоматизированной синхронизации различных версий текстовых файлов, но все они демонстрируют неудовлетворительный результат при попытке проводить этот процесс полностью автоматически.

Наш опыт показывает, что на практике первый способ обычно сочетается со вторым или третьим: первый способ используется для крупных модификаций компонент, последние два — для внесения небольшого количества хорошо локализуемых изменений. При этом второй способ целесообразно использовать при небольшом суммарном количестве модификаций, а третий — если таких модификаций оказывается много. Трудоемкость первого способа растет линейно от числа изменений, а третьего — как $C + \log(N)$, где C — константа, а N — число изменений, причем C достаточно велико.

3 Технологическое решение REAL-IT

Данное исследование проводилось на базе модельно-ориентированного технологического решения REAL-IT, которое нацелено на разработку информационных систем с использованием визуального моделирования предметной области и кодогенерации по созданным моделям. Это решение было разработано на кафедре системного программирования СПбГУ на базе объектно-ориентированного CASE-пакета REAL [11] и используется для разработки и сопровождения промышленных информационных систем. Подробное описание решения можно найти в [5].

3.1 Архитектура приложения

Работающая система, созданная при помощи технологического решения REAL-IT, состоит из следующих основных компонент:

1. База данных. Генерируется автоматически.
2. Программный интерфейс для доступа к базе данных. Генерируется автоматически.

3. Библиотеки поддержки экранных форм. Не зависят от разрабатываемой ИС, создаются “вручную”.

4. Библиотеки поддержки разрабатываемой ИС. Создаются “вручную”.

5. Библиотеки экранных форм. Формы генерируются автоматизированно. При необходимости внести в них изменения модифицируются “вручную”. Некоторые экранные формы создаются целиком “вручную”.

6. Запускающая программа. Создается “вручную” на основе шаблона.

3.2 Использование генераторов

Генераторы используются для построения схемы базы данных, программного интерфейса к ней и экранных форм по модели предметной области. При этом база данных и ее программный интерфейс генерируются автоматически — полученный код однозначно задается входной моделью, а экранные формы — автоматизированно. В процессе генерации каждой из экранных форм пользователь может указать дополнительные параметры [4].

4 Предлагаемое решение

Рассмотрим последовательно компоненты системы, при создании которых используются генераторы, и возможность применения в них тех или иных способов поддержки итеративного процесса разработки, перечисленных в п. 2.1.

4.1 База данных

Итак, на диаграммах, используемых в REAL-IT для генерации базы данных, можно описать все главные свойства ее элементов. Это позволяет не вносить в сгенерированный код схемы базы данных “ручных” изменений, а значит, и не заботиться об их восстановлении после перегенерации.

Однако при изменении схемы базы, независимо от способа ее разработки, возникает следующая проблема: как правило, информационная система уже существует и используется, а значит, содержит определенное количество данных. И эти данные не должны пропасть при установке у клиентов новой версии с измененной схемой базы данных.

Для решения этой проблемы можно предложить следующие способы:

1. Преобразовать структуру старой базы данных в соответствии с изменениями в схеме (например как это делается в ERwin).

2. Перенести информацию из старой базы данных в новую "пустую" базу.

Первый способ зачастую более предпочтителен для больших баз, особенно если изменений в схеме мало и они носят инкрементальный характер (например, если в схему добавлены только новые элементы — таблицы, атрибуты, ключи и т.д.). Однако такой подход довольно сложен в реализации, поскольку необходимо точно идентифицировать все расхождения между старой и новой версиями схемы. При сложных изменениях схемы могут понадобиться временные таблицы для хранения промежуточных результатов. Кроме того, соображения надежности требуют, чтобы перед выполнением подобной операции вся информация, хранящаяся в базе, была заархивирована. Это означает, что копирование информации все равно будет выполнено, но не в новую базу, а в архив (правда, такое копирование происходит обычно несколько быстрее).

В REAL-IT используется второй способ — по новой схеме создается пустая база данных, а прежние данные копируются в нее из старой базы. Это гарантирует, что структура новой базы будет соответствовать схеме. Перенос информации осуществляется специальным программным модулем — модулем миграции. Такой модуль расширяется разработчиками ИС при каждом нетривиальном изменении схемы базы данных. Кроме того, инструменты генерации создают некоторую дополнительную информацию при создании кода схемы базы данных по диаграммам для того, чтобы осуществлять контроль над информацией о версии схемы данных, с которой работает приложение.

Для этого контроля используется информация о номере версии и контрольной сумме схемы базы данных. Эта информация заносится при генерации базы данных в специальную служебную таблицу, а при генерации программного интерфейса к базе данных — в его код. Оба эти числа хранятся в модели базы данных в CASE-пакете REAL. Контрольная сумма пересчитывается при каждой перегенерации, и если она изменилась, то номер версии увеличивается.

Когда приложение открывает базу данных, происходит сравнение версий. Если они не совпали, то пользователю выдается соответствующее предупреждение с предложением запустить процедуру миграции.

Отметим некоторые проблемы миграции данных, возникающие при использовании REAL-IT.

1. Хранение информации о типах объектов. Каждый класс из модели предметной области при генерации схемы базы данных получает уникальный, в пределах данной базы, номер. Этот номер используется, например, для определения реального типа объекта данных, что актуально в связи с наличием в модели наследования классов [5]. Если при перегенерации базы допустить изменение номеров, приписанных классам, то при миграции данных придется обновлять привязку к типам для всех объектов. Чтобы этого не делать, номер классу присваивается при первой в его жизни генерации и сохраняется в дальнейшем. Он хранится в репозитории CASE-пакета REAL.

2. Изменение перечня предопределенных объектов и их свойств. При генерации базы данных имеется возможность сгенерировать не только схему базы данных, но и конкретные записи в ее таблицах, соответствующие экземплярам классов, существование которых необходимо для функционирования системы. Например, в системе “Студент” такими объектами являются оценки — “3”, “4”, “5”, “зачет”, “незачет”. Изменение свойств предопределенного объекта не нуждается в специальной обработке — новая версия объекта используется вместо старой, однако изменение набора объектов может потребовать специального рассмотрения. Возможны два случая:

(a) добавление нового предопределенного объекта. Проблемы возникают, если в мигрируемой базе уже есть объект данного типа, добавленный пользователем, который можно считать идентичным новому предопределенному объекту. В процессе миграции необходимо опознать такой объект и, если он есть, удалить его, осуществив перепровязку всех имевшихся ссылок на него от других объектов на новый объект;

(b) удаление существовавшего ранее предопределенного объекта. Проблемы возникают, если в базе существуют объекты, ссылающиеся на удаляемый. В таком случае необходимо изменить идентификатор объекта, чтобы он перестал восприниматься системой как предопределенный и перепровязать все ссылки.

4.2 Программный интерфейс базы данных

Программный интерфейс базы данных предназначен для предоставления объектно-ориентированного интерфейса к реляционной базе данных [7]. Классы программного интерфейса содержат, главным образом, методы для доступа к атрибутам объектов базы данных. Основное назначение этих методов — обеспечить статический контроль за использованием атрибутов.

При использовании набора записей “напрямую”, через стандартную библиотеку доступа к базе данных (например Microsoft DAO, Microsoft ADO), указываются названия атрибутов в виде строковых констант, поскольку транслятор во время компиляции не имеет информации о том, какие атрибуты существуют у объектов данных. Это приводит к ошибкам периода исполнения.

Особенно велика вероятность таких ошибок при изменении схемы (например, при переименовании или удалении атрибута) — старый код, который работает с базой данных, будет попрежнему транслироваться, но перестанет корректно работать. При доступе к данным через программный интерфейс ошибки такого рода будут выявлены во время компиляции. Таким образом, существование программного интерфейса поддерживает итеративный процесс разработки.

Потребность в изменении поведения классов программного интерфейса возникает очень редко, поскольку основное их пред-

назначение — служить "оберткой" для данных. Однако иногда такая потребность все-таки возникает, поскольку эти классы могут содержать некоторые элементы бизнес-логики. В таких случаях можно описывать методы классов непосредственно в схеме базы данных — эти методы добавляются при генерации программного интерфейса.

4.3 Пользовательский интерфейс

Как уже говорилось выше, интерфейс генерируется в интерактивном режиме, т.е. информацией для генерации служит не только модель в CASE-пакете, но и настройки, которые выбирает разработчик в процессе генерации того или иного элемента интерфейса (экранной формы). Для обеспечения возможности перегенерации эти настройки сохраняются в модели системы.

Однако несмотря на то что многие свойства интерфейса разработчик может настроить в процессе генерации, из всех генерируемых элементов системы именно интерфейс наиболее часто требует последующего внесения изменений. Перечислим основные типы "ручных" изменений в порядке убывания частоты их использования, выявленные нами по результатам выполнения ряда промышленных проектов с использованием REAL-IT:

1. Изменение расположения и размеров элементов управления.
2. Изменение поведения элементов интерфейса.
3. Добавление новых элементов интерфейса и замена элементов интерфейса на нестандартные.
4. Составление сложной формы из нескольких сгенерированных.

4.3.1 Изменение расположения и размеров элементов управления

Генератор по умолчанию располагает элементы управления на форме по некоторому строго определенному алгоритму. Порядок расположения некоторых элементов управления можно настроить в процессе генерации, но эти настройки не включа-

ют в себя возможность полного задания расположения элементов управления. Такое решение было принято в связи с тем, что управлять расположением элементов управления гораздо удобнее в визуальном режиме, который обеспечивается целевой средой разработки (например редактором форм Microsoft Visual Basic). С точки зрения реализации генераторов, было бы несложно дать возможность разработчику указывать в диалоге точное расположение и размеры элементов управления, но подобное решение было бы неудобным — скорее всего, генерация одной формы требовала бы в таком случае несколько итераций — указать размеры, посмотреть, что получилось, исправить размеры и т.д. Другой вариант — дать возможность пользователю указывать параметры элементов управления во время генерации визуально, но для этого потребовалось бы включить в генераторы собственный визуальный редактор форм, т.е. продублировать те возможности, которые уже есть в среде разработки. В REAL-IT было принято другое решение — использовать для настройки расположения элементов управления саму среду разработки. Настройка осуществляется после генерации, а при последующей перегенерации пользователю предлагается сохранить расположение элементов управления предыдущей версии формы, т.е. используется пятый способ из раздела 2.1.

4.3.2 Изменение поведения элементов интерфейса

Здесь применимы все те стратегии, которые описаны в разделе 2.1. Выбор конкретной стратегии зависит от используемого языка программирования и среды разработки. В REAL-IT в качестве основного способа внесения изменений используется механизм наследования с полиморфизмом. При необходимости модификации поведения экранной формы для нее создается класс-наследник, в котором переопределяются ее методы.

4.3.3 Изменение визуального представления (замена и добавление элементов управления)

Если целевая среда разработки поддерживает наследование экранных форм (как, например, Borland Delphi), то можно использовать те же решения, что и при изменении поведения, описанные в предыдущем пункте. Если же такой возможности нет

(как, например, в Microsoft Visual Basic), то изменения приходится вносить непосредственно в сгенерированную форму. Можно ли в такой ситуации поддержать процесс восстановления этих изменений при перегенерации (реализовать пятый способ из раздела. 2.1)?

В отличие от изменений алгоритмов поведения экранной формы, добавление или замена элемента управления обычно легко локализуются в тексте программы — в большинстве сред разработки можно легко определить набор элементов управления и их типы. Как сказано выше, для настройки расположения и размеров элементов можно брать соответствующую информацию из предыдущей версии экранной формы. В данном случае этой информации не достаточно, поскольку если элемент управления присутствовал в старой версии формы и отсутствует во вновь сгенерированной, то это может означать одно из двух: либо этот элемент был добавлен "вручную" на старую версию формы, либо он был сгенерирован, а его отсутствие в новой версии является следствием изменения исходной модели. Таким образом, генератор должен иметь возможность определять, какие из элементов управления, присутствующих на предыдущей версии формы, были сгенерированы, а какие — добавлены разработчиком "вручную". Можно предложить два способа решения этой проблемы:

1. сравнивать версию формы, которая была сгенерирована в последний раз, с версией, которая есть в данный момент, выявляя в последней "ручные" изменения;

2. помечать создаваемые "вручную" элементы управления каким-либо способом, доступным для проверки генератором.

Второй способ представляется более простым и предпочтительным.

Выявив "ручные" изменения в списке элементов управления формы, генератор может автоматически внести эти изменения на форму при перегенерации.

4.3.4 Составление сложной формы из нескольких сгенерированных

Иногда требуется использовать в приложении составные формы, не поддерживаемые генераторами непосредственно. Такие формы приходится создавать "вручную". В REAL-IT предлагается два способа сопровождения таких форм: во-первых, можно выделить основную форму, а элементы остальных форм считать добавленными "вручную"; во-вторых — поддерживать форму целиком в "ручном" режиме.

5 Реализация

Все перечисленные в разделе 4 стратегии, кроме п. 4.3.3, в настоящий момент реализованы в технологическом решении REAL-IT. П. 4.3.3 находится в стадии разработки. Пока не поддерживается составление сложной формы из нескольких простых (п. 4.3.4).

5.1 База данных

В данном разделе рассматривается реализация механизма миграции данных, обсуждавшегося в п. 4.1. По результатам выполнения ряда промышленных проектов с использованием REAL-IT нами был выявлен набор наиболее типичных модификаций схемы базы данных. К ним относятся:

1. Добавление в схему новых таблиц.
2. Добавление новых атрибутов в существующие таблицы.
3. Добавление новых связей между существующими таблицами.
4. Замена одной таблицы двумя связанными (при этом атрибуты старой таблицы делятся между новыми).
5. Удаление таблиц, связей и атрибутов.
6. Изменение типа атрибута, при котором все значения старого типа можно автоматически привести к новому типу. Наиболее распространенные случаи — это замена числового

типа на строковый или добавление новых значений в перечислимый тип.

7. Изменение типа атрибута с логического на перечислимый.

8. Переименование неудачно выбранного идентификатора атрибута.

Модификации, перечисленные в пунктах 1-3, 5, 6, мы будем называть тривиальными. В REAL-IT входит компонента UniMigrator, содержащая общую для всех приложений функциональность по миграции данных. При этом переносится информация из всех таблиц и их полей, имеющихся как в новой, так и в старой базе данных. Значение каждого атрибута автоматически приводится к его типу в новой базе. Тем самым, осуществляются все тривиальные преобразования. Кроме того, UniMigrator осуществляет перенос информации о предопределенных объектах по алгоритму, изложенному в п. 4.1.

Для осуществления других видов преобразований данных разработчики ИС могут расширить UniMigrator путем встраивания в него соответствующей компоненты — мигратора приложения. UniMigrator является частью стандартной библиотеки поддержки периода исполнения REAL-IT, он автоматически загружается при запуске системы и при каждом открытии базы данных проверяет совпадение версии схемы базы данных, сохраненной в базе и в коде программы. В случае, если открывается более старая версия, пользователю предлагается выполнить ее обновление. При этом база данных замещается на шаблонную базу (пустую) и осуществляется перенос информации. При копировании данных UniMigrator предоставляет COM-интерфейс, содержащий набор событий, позволяющий мигратору приложения влиять на процесс копирования. Мигратор приложения создается или модифицируется при каждом нетривиальном изменении схемы и содержит код, преобразующий данные в соответствии с этим изменением.

5.2 Программный интерфейс базы данных

Для модификации программного интерфейса базы данных используется возможность описания методов у классов, представляющих таблицы. В текущей реализации REAL-IT для каждого

класса из модели предметной области в программном интерфейсе генерируется несколько классов и модулей языка Microsoft Visual Basic. Для указания того, куда именно должен попасть описанный метод, используется специальное пользовательское свойство CASE-пакета REAL[2]. При этом определенные пользователем методы не только могут быть новыми, но и могут заменять собой методы, создаваемые генератором по умолчанию (если в указанном классе или модуле уже существует метод с такой сигнатурой). В такой ситуации старые методы также будут сгенерированы, но с другим названием, чтобы их можно было вызывать из новых методов. Кроме того, для осуществления некоторых типичных модификаций кода созданы специальные пользовательские свойства для элементов модели предметной области (классов, их атрибутов, ассоциаций между классами и т.д.), значения которых подставляются как фрагменты кода в генерируемые методы. Например, таким образом можно описать дополнительную проверку на значения того или иного атрибута.

5.3 Пользовательский интерфейс

Как было сказано выше, пользовательский интерфейс системы состоит из набора компонент, каждая из которых генерируется в интерактивном режиме. При этом все настройки, сделанные пользователем, сохраняются в модели системы. Это позволяет проводить перегенерацию всего интерфейса в пакетном режиме (чаще всего это бывает необходимо при изменении генераторов) — пользователю предлагается полный список всех компонент, которые уже были сгенерированы, из которых он может выбрать те, которые подлежат перегенерации. В случае, если проект, в который осуществляется генерация, уже содержит вариант генерируемой компоненты с тем же набором элементов управления, их расположение может быть восстановлено автоматически.

Для поддержки изменений поведения элементов интерфейса (п. 4.3.2) используются как пользовательские свойства, так и механизм наследования. В последнем случае модификации конкретной компоненты интерфейса выделяются в отдельный класс или набор классов.

[2]Пользовательское свойство — это возможность добавить для элемента метамодели новый атрибут. Подобные возможности расширения метамодели имеются во многих современных CASE-пакетах, например в Rational Rose.

Заключение

В работе рассмотрены проблемы поддержки "ручных" изменений кода при его перегенерации по высокоуровневым моделям и сохранение содержимого базы данных при обновлении ее схемы. Предложены механизмы их разрешения, описана реализация этих механизмов в рамках технологического решения REAL-IT.

Указанные механизмы активно используются на практике разработчиками ряда прикладных систем, позволяя проводить многократную перегенерацию кода при изменении модели или шаблонов генерации. В частности, предложенный способ перегенерации форм позволил достаточно легко перейти от генерации экранных форм в виде окон к генерации их в виде ActiveX-элементов управления с сохранением дизайна форм. Это позволило разработчикам прикладных систем перейти на компонентную архитектуру приложения. В то же время, практика показывает необходимость дальнейшего развития этих механизмов, в частности, реализацию стратегии поддержки модификации интерфейса, описанной п. 4.3.3.

В ходе реализации выявились недостатки языка Microsoft Visual Basic 6.0 — неполная реализация наследования, препроцессора и других средств структуризации кода, которые могли бы быть использованы для выделения "ручных" модификаций кода в отдельные компоненты. В связи с этим представляется перспективной апробация представленных механизмов в системах, разработанных на языках программирования, более полно поддерживающих объектно-ориентированную парадигму, в частности, на языке Java, а также на языке Microsoft Visual Basic.NET.

Список литературы

[1] Буч Г., Рамбо Д., Якобсон А. Унифицированный процесс разработки программного обеспечения. — СПб.: Питер, 2002. — 496 с.

[2] Влиссидес Д. Применение шаблонов проектирования. Дополнительные штрихи. — М.: Вильямс, 2002. — 144 с.

[3] Горин С., Тандоев А. Применение CASE-средства ERwin 2.0 для информационного моделирования в системах обработки данных // СУБД. — 1995. — № 3.

[4] Иванов А.Н., Стригун С.А. Технологическое решение REAL-IT: автоматизированная разработка пользовательского интерфейса информационных систем //Наст. сборник. — С. 124-147.

[5] Иванов А.Н. Технологическое решение REAL-IT: создание информационных систем на основе визуального моделирования // Наст. сборник. — С. 89-100.

[6] Кознов Д.В. Визуальное моделирование компонентного программного обеспечения: Дисс. канд. физ.-мат. наук. — СПб.: 2000. — 82 с.

[7] Кондратьев А. CASE-средства и объектные базы данных // Объектно-ориентированное визуальное моделирование. — СПб.: 1999. — С. 57-77.

[8] Круглински Д., Уингоу С., Шеферд Д. Программирование на Visual C ++ 6.0. Для профессионалов. — СПб.: Питер, 2000. — 864 с.

[9] Соммервилл И. Инженерия программного обеспечения. — М.: Вильямс, 2002. — 624 с.

[10] Сухарев М. Turbo Pascal 7.0. Теория и практика программирования. — М.: Наука и Техника, 2003. — 576 с.

[11] Терехов А.Н. и др. Real: методология и CASE-средство разработки информационных систем и программного обеспечения систем реального времени // Программирование. — 1999. — № 5. — С. 44-51.

[12] Харитонова И., Михеева В. Microsoft Access 2000. Разработка приложений. — СПб.: BHV, 2000. — 832 с.

[13] Adelsberger H., Korner F. Data Modeling with IDEF1X // LNCS. — 1995. — Vol. 973. — P. 355-391.

[14] Bassett P. Framing Software Reuse — lessons from the real world. — Prentice Hall: Yourdon Press, 1997. — 384 p.

[15] Customizing Forte Express Applications. — Fort—Software, Inc., 1997. — 134 p.

[16] Database Language SQL: ISO/IEC 9075:1992. — 1992. — 693 p.

[17] Specification and Description Language (SDL): ITU-T Recommendation Z. 100. — 1993. — 204 p.

[18] Rational Rose 98. Roundtrip Engineering with C++. — Rational Software Corp., 1998. — 455 p.

[19] Rational Rose 98. Using Rational Rose. — Rational Software Corp., 1998. — 267 p.

[20] Rumbaugh J., Jacobson I., Booch G. The Unified Modeling Language Reference Manual. — Addison-Wesley, 1999. — 576 p.

[21] Schlungbaum E. Model-Based User Interface Software Tools. Current state of declarative models. — Atlanta: 1996.

[22] Siegel J. CORBA Fundamentals and Programming. — N.-Y.: Wiley, 1996. — 693 p.

[23] Together Evaluation Guide. — TogetherSoft LLC, 1999. — 46 p.

[24] Vlissides J., Tang S. A Unidraw-based User Interface Builder // Proc. of the ACM SIGGRAPH Fourth Annual Symposium on User Interface Software and Technology. — 1991. — P. 201-210.