УДК 004.4′24

М.Н. Кравцов

Харьковский национальный университет им. В.Н. Каразина

АНАЛИЗ ЭФФЕКТИВНОСТИ ПРИМЕНЕНИЯ СУЩЕСТВУЮЩИХ ТЕХНОЛОГИЙ СИНТЕЗА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Рассмотрены существующие технологии синтеза программного обеспечения (ПО), особенности их реализаций, их влияние на процесс разработки ПО. Предложена методика количественной оценки их эффективности в рамках конкретного проекта разработки ПО, с учетом многослойной архитектуры и распределения трудоемкости по слоям. Исследование проводится в целях выявления потенциальных усовершенствований существующих технологий синтеза ПО, а также создания новых технологий синтеза, обладающих большей эффективностью.

Ключевые слова: *разработка ПО, кодогенерация, интроспекция, паттерн, база данных, веб-приложение, метаданные.*

Введение

На данный момент существует несколько методик синтеза программного обеспечения (ПО) и его реализаций, призванных упростить и ускорить процесс разработки программных продуктов. Строго говоря, любой компилятор или интерпретатор компилирующего типа является кодогенератором, соответственно разработчик, использующий наиболее распространенные языки программирования общего назначения, так или иначе имеет дело с кодогенерацией в некоторой форме. Однако в данной статье рассматривается только аспект прикладной кодогенерации – когда трансляции не происходит, а синтезируемый код находится на том же уровне, что и код, создаваемый вручную. Наиболее очевидным и распространенным примером реализации являются интегрированные среды разработки. В процессе собственной эволюции среды разработки прошли путь от примитивных текстовых редакторов с подсветкой синтаксиса до масштабных приложений, содержащих довольно богатый набор встроенных кодогенерирующих инструментов. В качестве распространенных примеров можно привести Microsoft Visual Studio фирмы Майкрософт, среды Delphi и C++ Builder фирмы Борланд.

Современные методики кодогенерации применяются не только в средах разработки. Общепринято, что универсальным критерием эффективности технологии синтеза является сокращение общего времени, затрачиваемого разработчиком в процессе разработки программного продукта [1]. Это может выражаться как в сокращении времени, затрачиваемого непосредственно на кодирование, так и в сокращении времени, затрачиваемого на тестирование и отладку. Строгое и точное вычисление эффективности применения технологии синтеза для каждого конкретного проекта вряд ли можно назвать экономически оправданным. Это обусловлено тем, что точный результат даст только реализация проекта двумя командами разработчиков примерно одной квалификации и опыта разработок для предметной области проекта: одна из них будет использовать технологию синтеза в процессе работы над проектом, а вторая – нет. Последующее сравнение затраченного на разработку времени этими двумя командами и даст требуемую оценку.

В виду описанных выше причин для анализа эффективности применения инструментальных средств синтеза программного обеспечения требуется более дешевая и простая методика, которая позволит получить конкретный, количественно выраженный в человеко-часах или процентах, ответ на вопрос: на сколько сократит данный конкретный инструмент время разработки данного проекта?

1. Формулировка проблемы

Требуется дать ответ на вопрос: возможно ли уменьшить трудоемкость процесса разработки программного обеспечения для конкретного проекта путем использования конкретного инструментального средства и если да, то насколько?

2. Решение проблемы

Следует сразу же оговорить следующие моменты:

− сама постановка задачи уже предполагает известным значение некоторой «базовой» трудоемкости проекта, которая соответствует технологии, не предполагающей использования рассматриваемого инструментального средства;

− положительный эффект от сокращения рисков, связанных с человеческим фактором, не учитывается.

Общепринятым для современной IT-индустрии является многослойная архитектура [2, 3], которая

характеризуется декомпозицией информационных систем и отдельных приложений на несколько слоев или уровней (рис. 1):

- уровень данных;
- уровень доступа к данным;
- уровень бизнес-логики;
- уровень представления.

Рис. 1. Типовая многослойная архитектура информационной системы

Соответственно наиболее распространенные на сегодняшний день инструментальные средства предлагают процедуры автоматизации, применяющиеся в рамках процесса построения этих слоев программного обеспечения. Уже давно известны и хорошо себя зарекомендовали визуальные построители графических интерфейсов, изначально получившие широкое распространение в виде продуктов фирмы Borland: Delphi и C++ Builder. Среди инструментов для слоя бизнес-логики можно выделить продукты фирмы Rational (в настоящее время поглощена фирмой IBM). Что касается доступа к данным то здесь, на взгляд автора, пальму первенства держит Visual Studio .NET фирмы Microsoft

Так или иначе в общем виде эффект от применения конкретного инструмента можно записать в виде:

$$\Delta T = T_{EST} - \sum_{i=1}^{1} \Delta T_i + L ,$$

где T_{EST} – предполагаемая трудоемкость проекта, без использования инструментального средства; ΔT_i – предполагаемое сокращение трудоемкости разработки i-го слоя, с использованием инструментального средства; L – затраты времени, связанные с внедрением инструментального средства.

Ранее было сказано, что T_{EST} известно исходя из постановки задачи, поэтому нас будет интересовать определение ΔT_i и L. При этом L – величина сильно зависимая от конкретного состава команды разработчиков и организации. Так при наличии у команды определенного опыта работы с этим инструментальным средством и наличии установленного инструментального средства на всех рабочих станциях, L может быть равно нулю.

Методика определения ΔT_i, с одной стороны, зависит от методики определения T_{EST}, а с другой стороны, также сильно зависит от распределения трудоемкости по слоям программного обеспечения в проекте и собственно ориентации самого инструментального средства.

2.1. Инструментальные средства уровня представления. Естественно, что инструмент для построения интерфейсов не сократит время разработки приложения существенно, если 90% трудоемкости приходятся на работу с базой данных и бизнес-логику. Однако в ряде случаев не стоит пренебрегать возможностью сократить эти 10%, приходящиеся на интерфейс. Попробуем определить, на сколько экономит время визуальный построитель интерфейсов, для начала проведем небольшой эксперимент – создадим примитивное приложение для .NET, всего лишь одна форма с одной кнопкой. Перетащим с панели инструментов кнопку на форму и засечем время, которое на это ушло – 2 секунды, если тащить очень размеренно. Еще одна секунда и среда создает метод для обработчика события нажатия на кнопку. При этом среда SharpDevelop поместила в исходный файл следующее:

```
      .      .      .      .      .
private this.button1 =
 new System.Windows.Forms.Button();
this.button1.Location =
 new System.Drawing.Point(243, 142);
this.button1.Name = "button1";
this.button1.Size =
 new System.Drawing.Size(75, 23);
this.button1.TabIndex = 0;
this.button1.Text = "button1";
this.button1.
 UseVisualStyleBackColor = true;
this.button1.Click =
 new System.EventHandler(
  this.Button1Click);
.      .      .      .      .
void Button1Click(object sender, Even-
 tArgs e)
{
.      .      .      .      .
}
```

Даже если предположить, что разработчик совершенно точно знает, что именно ему следует набрать, чтобы получить именно такую кнопку в этом месте формы и ни разу не ошибется при наборе текста, то это займет явно не три секунды. У автора это заняло 3 минуты 34 секунды, да, если потренироваться, можно было бы это время и сократить, но даже чисто физически за 3 секунды набрать этот текст не получилось бы. Использование этого инструмента в данной ситуации привело к сокращению трудоемкости в 60 раз! Конечно, в реальных проек-

тах и при поддержании хорошего стиля количество такого «удачно генерируемого» кода составляет в лучшем случае половину от кода, относящегося к уровню представления, что все равно неплохо. Особенно если учесть, что данная среда разработки распространяется бесплатно и максимум что требуется для ее использования, это скачать 10-ти мегабайтный дистрибутив и установить приложение из готового инсталляционного пакета. Разумеется, построение интерфейса не сводится только к перетаскиванию кнопок, обычно как минимум надпись на кнопке и координаты устанавливаются вручную, но это не радикально повлияет на ситуацию. Кроме того, при отладке визуальный построитель позволяет увидеть картину целиком без необходимости компилировать и запускать приложение после каждой небольшой корректировки. В целом при разработке desktop-приложений использование визуальных инструментов экономит время всегда и значительно, при этом не принципиально, какой именно инструмент используется, в большинстве своем они выполняют стандартные функции стандартным образом.

С веб-приложениями ситуация несколько иная. С одной стороны, они позволяют разработчикам, вообще не имеющим представления об HTML, строить более-менее сносные интерфейсы. С другой стороны, код, генерируемый инструментами для веб, обычно представляет собой джунгли из тэгов с установленными inline-стилями, в которых прописано абсолютное позиционирование элементов. Тем не менее, большинство из этих инструментов позволяют отключить функцию абсолютного позиционирования и, хотя увиденное в визуальном построителе, может очень сильно отличаться от того, что будет показано в браузере, они обеспечивают сокращение трудоемкости построения интерфейса.

Построители веб-интерфейсов обеспечивают значительное сокращение времени на первичном размещении элементов на формах и генерации корректных обработчиков событий формы. По сравнению с десктоп-построителями они теряют только преимущество удобного позиционирования элементов, что несколько снижает их относительную эффективность, остающуюся, однако, высокой, что позволяет значительно сокращать время на разработку интерфейса. Вопрос использовать или не использовать визуальные генераторы интерфейсов на данный момент особого интереса не представляет, так как любой разработчик, знакомый с этими инструментами, будет использовать ту или иную реализацию такого инструмента, которая по пессимистическим оценкам сократит время на разработку слоя представления данных на

$$\Delta T_1 = 25\% - \frac{25\%}{60} = 24,584\% .$$

При этом под пессимистической оценкой предполагается, что генерируемый, не требующий ручной корректировки, код составляет 25% от общего кода уровня представления.

2.2. Инструментальные средства уровня бизнес-логики. Программирование уровня бизнес-логики, пожалуй, наименее поддается автоматизации. Причина заключена в специфичности кода для каждого отдельного проекта.

Для автоматизации программирования на этом уровне используют продукты, позволяющие генерировать объектные модели в виде кода из визуально построенных диаграмм. Разумеется, что в этом случае генерируются только заготовки для классов и интерфейсов с пустыми объявлениями методов. В отличие от уровня представления удельный вес генерируемого кода слоя значительно ниже, не более 15%. Если такого кода больше, это может говорить о присутствии анти-паттерна Anemic Domain Model [3], что является плохой практикой. Кроме того, такие инструментальные средства не всегда поддерживают возможность реверсивного инжиниринга, то есть генерации внутреннего представления модели инструментального ПО из готового исходного кода. На практике это означает одноразовое применение инструмента в начале проекта, с последующим сопровождением слоя бизнес-логики полностью вручную. Это естественно, так как поддержание актуальности текущей объектной модели в коде и на языке представления инструмента в большинстве случаев будет более затратным, чем прямое ручное сопровождение.

В итоге получается, что эффект от применения инструмента для слоя бизнес-логики будет уменьшаться с количеством изменений в объектной модели. Неудивительно, что такие средства не получили широкого распространения.

Для получения количественной оценки – требуется найти отношение количества генерируемого кода к общему количеству кода в слое. Это сильно зависит от стиля программирования и архитектуры приложения, но, если команда придерживается хорошего стиля разработки, не придерживаясь неявно ни Anemic Domain Model [4], ни The Blob [5], то генерируемый код будет составлять не более 15% от общего объема кода. Объявления полей в классах и пустые заглушки методов вряд ли будут занимать более 7 – 10% исходного кода бизнес-логики, объявления же интерфейсов слегка улучшат картину, так как не предполагающие реализации объявления интерфейсов могут быть сгенерированы полностью. Однако следует помнить, что визуальное построение объектной модели происходит не мгновенно и затраты времени на создание объявления класса вручную в редакторе вполне сопоставимы с затратами на создание класса в визуальной среде. Поэто-

му реальная польза от таких инструментов проявляется косвенно – визуально намного легче проектировать и верифицировать объектную модель. И, если такой инструмент используется при проектировании, то он может быть использован с максимальной эффективностью в процессе кодирования – раз модель уже создана, то почему бы и не выжать из нее эти 15% сокращения времени на кодирование бизнес-логики? При этом использование инструмента только лишь с целью сокращения написанного вручную кода вряд ли даст ощутимый эффект. С оговоркой, что инструмент был использован при проектировании, пессимистичная оценка эффективности составит: $\Delta T_2 = 15\%$.

2.3. Инструментальные средства уровня доступа к данным. В отличие от программирования на уровне бизнес-логики программирование на уровне доступа к данным может быть успешно поддержано средствами автоматизации. Условно такие средства можно разделить на:

− библиотеки объектно-реляционного отображения;

− генераторы классов-оболочек.

Существует большое количество различных библиотек объектно-реляционного отображения. Основная проблема при их использовании – конфигурирование. Для того чтобы задать правила отображения объекта из объектной модели на таблицу или группу таблиц в реляционной базе данных, обычно надо их внести в конфигурационный файл. Это требует специфичных знаний, а затраты на написание конфигурационных файлов иногда оказываются сопоставимы или даже превышают по трудоемкости прямую реализацию объектно-реляционного отображения. Однако при всем этом некоторые библиотеки получили очень широкое распространение, и требования по умению с ними обращаться нередко встречается в описаниях вакансий. Чаще всего в этом контексте упоминают Hibernate. При наличии некоторых специфических знаний такие библиотеки избавят разработчика от рутинных операций с базой данных и написания тривиального кода. Обычно их использование оправдано для проектов с развитой бизнес-логикой.

Генераторы классов-оболочек позволяют генерировать как объекты приложения, так и функции для их сериализации в базу данных. Широкое распространение получил инструмент этого класса XSD от Microsoft, интегрированный со средой разработки Visual Studio этой же компании.

Преимущество генераторов по сравнению с библиотеками состоит, в первую очередь, в том, что они позволяют автоматически получать код, поддающийся трассировке. Это облегчает локализацию проблем, возникающих внутри сгенерированного кода.

В реализации генератора от Microsoft автоматически строится слепок реляционной базы данных со всей структурой, включающей таблицы, связи, ограничения целостности и т.д. в виде объектной модели. При этом либо инструмент может сгенерировать SQL-запросы для каждой таблицы на основе заданного SELECT, либо разработчик может написать свои, которые будут сохранены во время построения в файле метаданных модели.

Оценить количественно эффект от использования инструментов этого класса достаточно тяжело. Это обусловлено тем, что прямое вычисление соотношения генерируемого кода к общему коду слоя не всегда является возможным:

− некоторые библиотеки позволяют генерировать код «на лету», что затрудняет точный подсчет объема сгенерированного кода;

− генераторы же оболочек зачастую генерируют значительный объем избыточного кода, который не является необходимым для конкретного приложения.

Тем не менее, удается вычислить отношение между временем, затраченным на реализацию типовой операции с использованием инструмента, и временем, затраченным на ее реализацию, без его использования. Так ручная реализация объектно-реляционного отображения для эталонного объекта, включающая разработку функций создания, обновления и удаления, требует на 210% больше времени, чем его реализации с помощью Hibernate и на 80% времени больше, чем реализация его с помощью Visual Studio .NET. Стоит также отметить, что Visual Studio мгновенно генерирует код реализации паттерна Data Transfer Object [5, 6] для данного объекта и реализация объектно-реляционного отображения не вписывается напрямую в сценарий, который закладывали в этот инструмент разработчики.

На это следует обратить внимание – в подавляющем большинстве случаев наилучший эффект от использования инструмента получается если инструмент используется так, как предполагалось разработчиками. То есть инструмент в некотором роде даже диктует и предписывает определенную архитектуру и некие правила построения приложения, при которых он проявляет себя наилучшим образом. Отступление от этих правил приводит к снижению эффекта от применения инструмента, и может даже оказаться, что применение инструмента в итоге повлекло за собой большие затраты, чем те, которые понадобились бы для реализации данной части проекта вручную.

На практике, исходя из опыта автора, пессимистическая оценка эффективности составляет $\Delta T_3 = 30\%$.

Выводы

Разумное совместное применение инструментальных средств и методик проектирования [7, 8]

позволяет значительно сократить затраты на реализацию проекта.

Не всякое инструментальное средство синтеза способно дать положительный эффект при разработке конкретного проекта. Также при принятии решения об использовании конкретного инструмента следует учитывать его доступность и возможные временные затраты, которые могут потребоваться разработчикам для его освоения.

Эффективность конкретного инструментального средства в рамках конкретного проекта зависит от удельного веса целевого программного слоя в проекте, таким образом визуальный построитель интерфейса пользователя вряд ли будет полезен при разработке невизуальных компонентов, но даст очевидный эффект при разработке приложения с графическим пользовательским интерфейсом. При разработке веб-приложений визуальные построители интерфейсов дают меньший эффект, чем при разработке десктоп-приложений.

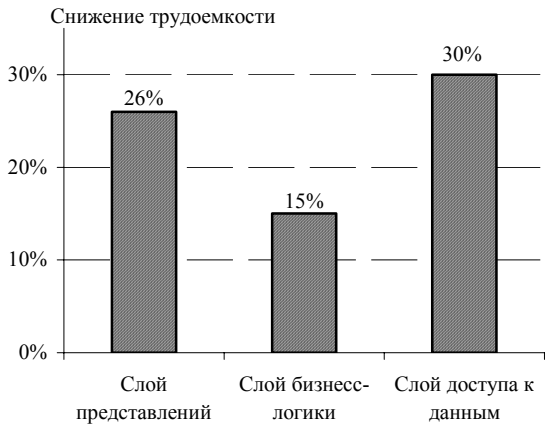

Рис. 2. Прогнозируемое снижение трудоемкости за счет применения кодогенераторов

По пессимистическим оценкам инструментальные средства позволяют сократить время разработки конкретного слоя на 15 – 30% (рис. 2). Слой бизнес-логики является наименее поддающимся автоматизации, что с одной стороны, означает необходимость больших затрат при его реализации, а с другой – высокую концентрацию метаданных в нем. Это позволяет рассматривать слой бизнес-логики как потенциальный источник информации для более совершенных инструментальных средств, использующих минимум информации (не требующей предварительной подготовки) и синтезирующих максимум кода, который иначе пришлось бы писать вручную.

Список литературы

1. Брукс Ф. Мифический человеко-месяц. – М.: Символ – Плюс, 2001. – 304 с.

2. Таненбаум Э., ван Стеен М. Распределенные системы. Принципы и парадигмы. – С.-Пб.: Питер, 2003. – 877 с.

3. Соммервилл И. Инженерия программного обеспечения. – М.: Вильямс, 2002. – 624 с.

4. Fowler M. Anemic Domain Model. – http:// www.martinfowler.com/bliki/AnemicDomainModel.html.

5. The Blob. – http://www.antipatterns.com/briefing/ sld024.htm.

6. Data Transfer Object. – http://msdn.microsoft.com/ en-us/library/ms978717.aspx.

7. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма и др. – С.-Пб.: Питер, 2003. – 366 с.

8. Фаулер М. Архитектура корпоративных программных приложени й. – М.: Вильямс, 2004. – 544 с.

Поступила в редколлегию 18.08.2008

Рецензент: д-р техн. наук, проф. Г.Н. Жолткевич, Харьковский национальный университет имени В.Н. Каразина, Харьков.

АНАЛІЗ ЕФЕКТИВНОСТІ ЗАСТОСУВАННЯ ІСНУЮЧИХ ТЕХНОЛОГІЙ СИНТЕЗУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

М.М. Кравцов

Розглянуті існуючи технології синтезу програмного забезпечення (ПЗ), особливості їх реалізацій, їх вплив на процес розробки ПЗ. Запропонована методика кількісної оцінки їхньої ефективності в рамках конкретного проекту з розробки ПЗ з урахуванням багатошарової архітектури та розподілення працеємності за шарами. Дослідження проводитьсяИсследование проводится з метою виявлення потенційних вдосконалень існуючих технологій синтезу ПЗ, а також створення нових технологій, що матимуть більшу ефективність.

Ключові слова: *розробка ПЗ, кодогенерація, інтроспекція, паттерн, база даних, веб-додаток, метадані.*

EFFICIENCY ANALYSIS OF EXISTING SOFTWARE SYNTHESIS TECHNOLOGIES

M.N. Kravtsov

Existing software synthesis technologies, peculiarities of their implementations, their influence on software development process have been reviewed. Quantitative measure of their efficiency, within particular software development project, has been offered. It takes into account multi-layered software architecture and layer-wise workload distribution. The research is performed to identify potential improvements in existing software synthesis and also to develop newer, more efficient technologies.

Keywords: *software development, code generation, introspection, pattern, database, web-application, metadata.*