Структурный рефакторинг многослойных программных систем

С.В. Назаров, Н.Н. Вилкова

Аннотация. Эволюция сложных программных систем требует от разработчика повышенного внимания к выбору их архитектуры. Сложившаяся ситуация с проектированием подтверждает, что практически всегда во время разработки программных систем появляются новые требования со стороны заказчика, и приходиться пересматривать первоначальную архитектуру. Не исключены и просчеты исполнителей на этапах проектирования и кодирования системы. Отсюда повышенный интерес к вопросам рефакторинга программных систем. В то же время актуальность проблемы проектирования архитектуры программных систем и ее рефакторинга пока не нашла должного отражения в опубликованной литературе. Вопросам архитектурного рефакторинга посвящено незначительное количество работ. В данной работе рассматривается задача структурного рефакторинга многослойных программных систем с целью повышения производительности системы. Предлагается подход к представлению структур многослойных программных систем. Дается математическая постановка задачи структурного рефакторинга в виде задачи линейного программирования с булевыми переменными и обсуждается подход к ее решению.

Ключевые слова: многослойная программная система, архитектура, рефакторинг, структура, производительность.

Введение

Считается, что концепция «рефакторинга» возникла в кругах, связанных со Smalltalk, но вскоре утвердилась у приверженцев и других языков программирования. М. Фаулер дал определение рефакторинга как небольшого изменения в исходном коде, которое способствует улучшению проекта кода без изменения его семантики [1]. Им же была высказана идея рефакторинга БД, однако об архитектурном рефакторинге программных систем речи не было. Надо сказать, что в настоящее время вопросам архитектурного рефакторинга посвящено незначительное количество работ. Основными следует отметить работы М. Ксензова [2, 3]. В то же время эволюция сложных программных систем требует от разработчика повышенного внимания к выбору архитектуры. Практически всегда во время разработки появляются новые требования со стороны заказчика, и приходиться пересматривать первоначальную архитектуру, в том числе и структуру базы данных. Условно выделяются следующие фазы архитектурного рефакторинга: фаза "раскопки" архитектуры, фаза трансформации архитектуры, фаза семантического анализа подсистем и фаза проецирования изменений модели на программный код. В данной работе рассматриваются вопросы рефакторинга многослойных программных систем (ПС), целью которого является повышение производительности системы.

1. Начальный этап создания программной системы

Как правило, значительная часть программных систем (ПС) создается в срочном порядке. Требуется автоматизировать (создать поддерживающую ПС) для некоторой совокупности взаимодействующих бизнес-процессов. Наспех составленное техническое задание передается

выбранной (возможно без предварительного анализа или на основе тендера) компьютерной фирме, которая обещает выполнить работу в требуемые (как правило, минимальные) сроки и за приемлемую стоимость.

Подобные фирмы чаще всего используют гибкие технологии создания программных систем, основанные на итерационном и инкрементном подходе к созданию ПС. Это может быть SCRAM или Agile-методология с элементами экстремального программирования. В таких технологиях действует правило: “проектируйте только то, что необходимо”. Действительно, когда стоимость разработки или издержки в случае неудачного дизайна очень высоки, может потребоваться полное предварительное проектирование и тестирование. При гибкой разработке можно избежать масштабного проектирования наперед (big design upfront, BDUF). Если требования к приложению четко не определены или существует вероятность изменения дизайна со временем, можно не тратить много сил на проектирование раньше времени. Этот принцип называют YAGNI («You ain’t gonna need it» − Вам это не понадобится).

Такой подход к разработке ПС позволяет достаточно быстро создать совокупность программных модулей, автоматизирующих заданный набор бизнес-процессов B. Однако зачастую эти модули создаются независимо друг от друга, и в этом случае могут быть пересечения по функциям, реализуемым модулями. Возможны (и это чаще) ситуации, когда один модуль может обращаться к другому для выполнения некоторых функций, реализуемых этим модулем. Здесь нужно заметить, что под модулем понимается достаточно произвольный структурный элемент ПС (в зависимости от уровня рассмотрения это: подсистема, компонент, отдельный программный модуль, группа классов, отдельный класс), который можно выделить, определив интерфейс взаимодействия между этим модулем и всем, что его окружает.

Очень часты ситуации, когда разрабатываемая программная система слабо документируется, и об архитектуре создаваемой программной системы, и ее целесообразности разработчики особенно и не задумываются. Однако, тем не менее, архитектура разрабатываемой системы существует, и она собственно создана ее авторами-разработчиками независимо от их желания. В первом приближении архитектуру ПС в этом случае можно представить некоторым множеством программных модулей:

$$M = \{m_{ij} \mid i = 1,2,\dots,N,\ \ j = 1,2,\dots,n_i\},$$

$$M = \bigcup_{i}^{N} M_i\ , \qquad |M| = K,$$

где N – количество бизнес-процессов;

K – количество модулей в программной системе;

i – номер бизнес-процесса $b_i \in B$;

j – номер модуля, реализующего j-функцию i-го бизнес-процесса;

n_i - количество функций, реализуемых i-м бизнес-процессом;

M_i – подмножество модулей, автоматизирующих i-й бизнес-процесс.

В общем случае справедливо соотношение

$$\bigcap_{i}^{N} M_i \neq \emptyset.$$

Каждый модуль m_{ij} можно представить следующими параметрами спецификации:

$$P_{ij} = \{Name, I_{ij}, O_{ij}, A_{ij}\},$$

где $Name$ – имя модуля m_{ij},

I_{ij}– параметры входного интерфейса модуля m_{ij},

O_{ij} – параметры выходного интерфейса модуля m_{ij},

A_{ij}– абстракция алгоритма, реализуемого модулем m_{ij}.

Заметим, что абстракция через спецификацию позволяет абстрагироваться от алгоритма, описанного в теле модуля, до уровня знания лишь того, что данный модуль должен в итоге реализовать. Это достигается созданием для модуля спецификации, описывающей эффект его работы, после чего смысл обращения к данному модулю становится ясным через анализ этой спецификации, а не самого тела модуля.

Существует отображение вида $O: B \to M$, которое создано разработчиками в процессе написания модулей ПС и определяет подмножества модулей, автоматизирующих функции конкретных бизнес-процессов. Таким образом, существуют отображения O_i: $b_i \to M_i \subset M$,

$i = 1,2,\dots,N$. При этом возможны непустые пересечения

$$M_i \cap M_j \neq \emptyset;\ i,j = 1,2,\dots,N.$$

Это свидетельствует о возможном дублировании некоторых функций бизнес-процессов в автоматизирующих их модулях ПС. Однако возможны ситуации, когда для некоторого отображения O_i $|M_i| < |b_i|$, из чего следует что ПС реализует не все функции бизнес-процесса b_i. Но даже если в этом плане нет претензий к разработанной программной системе, как отмечено выше, часто архитектура ПС не только предварительно не разрабатывается, но и недостаточно (или совсем) не документируется. Отсюда в интересах дальнейшей разработки системы или ее сопровождения возникает проблема "раскопки архитектуры", как ее часто называют в литературе [4, 5].

2. Представление созданной архитектуры ПС (раскопка архитектуры)

Общепризнано, что удобным и наглядным способом представления архитектуры программных систем является использование графов. В работе [4] строится модель ПС на основе исходного кода, когда может отсутствовать информация о составляющих систему блоках. В нашем случае рассматривается пример разработки ПС на основе гибкой технологии, когда в разрабатываемую систему последовательно добавляются новые модули. В этом случае каждый модуль m_{ij} системы можно представить именем *Name* и частью параметров O_{ij} из спецификации модуля – именами модулей, которые могут быть вызваны из модуля m_{ij} . Для удобства и простоты дальнейших построений каждый модуль будем представлять в следующем виде:

$$m_{ij} \rightarrow < namber1, namber2, namber3, \dots >,$$

где $namber1$ – номер модуля m_{ij} , отождествляемый с его именем *Name;*

$namber2$ – номер 1-го модуля, к которому может обращаться модуль m_{ij} ;

$namber3$ – номер 2-го модуля, к которому может обращаться модуль m_{ij} и т.д.

Таким образом, в целом перечень всех модулей и их взаимосвязей можно представить списком следующего вида:

$$S = S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_K,$$

где S_i, $i = 1,2,\dots,K$ – элементы списка следующей структуры:

$$S_1 = < 1, s, k, m, \dots >,$$
$$S_2 = < 2, l, t, f, \dots >,$$
$$\dots\dots\dots$$
$$S_K < K, g, z, u, \dots >,$$

где $s, k, m, l, t, f, \dots, g, z, u, \dots$ – номера модулей.

На основе списка *S* можно построить граф *G*, отображающий структуру ПС. Однако в таком представлении трудно сделать вывод о типе архитектуры программной системы и ее качестве.

Известно, что значительная часть современных программных систем имеет многослойную архитектуру. Выделяют различные типы такой архитектуры. В таких архитектурах модули нижнего слоя для выполнения своих функций не обращаются к другим слоям. Организация вышележащих слоев может быть различной. Поэтому при анализе полученной архитектуры ПС первой задачей является выделение слоев модулей.

Многослойная архитектура обеспечивает группировку связанной функциональности приложения в разных слоях, выстраиваемых вертикально, поверх друг друга. Функциональность каждого слоя объединена общей ролью или ответственностью. Слои слабо связаны, и между ними осуществляется явный обмен данными. Правильное разделение приложения на слои помогает поддерживать строгое разделение функциональности, что обеспечивает гибкость, а также удобство и простоту обслуживания.

Слои приложения могут размещаться физически на одном компьютере (на одном уровне) или быть распределены по разным компьютерам (n-уровней). Связь между компонентами разных уровней осуществляется через строго определенные интерфейсы. Например, типовое веб-приложение состоит из слоя представления (UI), бизнес-слоя (обработка бизнес-правил) и слоя данных (функциональность, связанная с доступом к данным).

Далее будем рассматривать программные системы одного определенного языкового уровня (с хорошо определенными синтаксическими единицами в соответствии с хорошо

определенными синтаксическими правилами и хорошо определенной семантикой элементарных операторов и синтаксических конструкций). Из рассмотрения исключим все вопросы, относящиеся к другим языковым уровням, таким, например, как интерпретация элементарных операторов в терминах более примитивных составляющих.

Элементарные операторы данного языкового уровня будем рассматривать как модули базового уровня, составляющие базовый слой. Это можно сделать потому, что все элементарные операторы повсеместно доступны: единственные возможные ограничения на использование элементарных операторов касаются входящих в них данных, т.е. фактических параметров активации модулей. Заметим, что здесь не учитываются привилегированные операторы машинного языка, которые не могут использоваться в прикладных программных системах.

Модули, построенные из модулей базового уровня, могут рассматриваться только как модули нулевого уровня. При этом не требуется, чтобы все модули нулевого уровня были равнодоступны: например, в программах, написанных на языке, допускающем блочную структуру, некоторые модули нулевого уровня могут быть локализованы в каком-либо блоке и, следовательно, доступны только внутри этого блока и его подблоков.

С другой стороны, при конструировании модулей высших уровней в некоторых языках можно использовать модули разных уровней и даже того же самого уровня, что и конструируемый модуль (рекурсия, сопрограммы).

Введем в рассмотрение матрицу R размером $K \times K$, каждый элемент которой образуется по правилу:

$$r_{ij} = \begin{cases} 1, & \text{если } j \in S_i, \\ 0 - \text{в противном случае.} \end{cases}$$

Далее можно следовать алгоритму, который дается ниже.

0. Начало, I = 0 (слой ПС).

1. Находим в матрице номера строк, все элементы которых равны нулю.

2. Фиксируем вершины с этими номерами, образующими I-слой.

3. I = I + 1.

4. Если остались столбцы с ненулевыми элементами, обнуляем столбцы с номерами найденных вершин. Переходим к п. 1.

5. Если все столбцы содержат только нулевые элементы, конец.

Надо заметить, что данный алгоритм позволяет построить послойную архитектуру ПС, которая удовлетворяет одному из вариантов, рассмотренных в [5]. Однако если в слоях ПС имеются горизонтальные связи или сильносвязанные модули, то полностью определить структуру ПС без дополнительного анализа не удастся.

3. Анализ на соответствие послойной архитектуре (выделение слоев)

Рассмотрим порядок проведения анализа на конкретном примере. Пусть задана списком S некоторая совокупность модулей ПС (9 модулей), которая представляется следующей матрицей R.

$$R = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Действуя по приведенному выше алгоритму, находим нулевые строки (8 и 9 зачеркнуты сплошной линией), и таким образом определяем модули нулевого слоя: $L_0 = \{8, 9\}$. Вычеркиваем 8 и 9 столбцы (пунктир из круглых точек). Находим нулевую строку с номером 7 (зачеркнута штрихпунктирной линией), определяющую слой $L_1 = \{7\}$. Вычеркиваем седьмой столбец (штриховая линия) и определяем слой $L_2 = \{4, 5\}$. Отмечаем строки 4 и 5 длинными штрихами. Вычеркиваем столбцы 4 и 5 (длинный штрих-пунктир). По ставшим нулевыми строкам определяем модули третьего слоя $L_3 = \{2, 3, 6\}$. Вычеркиваем строки 2, 3 и 6 (пунктир жирными точками). Вычеркиваем

столбцы с этими номерами (длинный штрих двойной пунктир) и определяем модули четвертого слоя $L_4 = \{1\}$.

Получив распределение модулей по слоям ПС, можно построить граф G программной системы (Рис. 1).

Анализируя полученный граф, следует отметить, что он не отвечает каноническим правилам многослойной структуры. В частности, модуль 6 не отвечает этим требованиям. Известно, что выделение слоев – хорошая основа для улучшения системы. Найти строгие слои в произвольной программной системе достаточно трудно, поскольку, как уже отмечалось, они могут содержать горизонтальные связи и сильносвязанные компоненты. Поэтому целесообразно расширить понятие слоя, позволив включать в произвольные слои сильносвязанные компоненты [2, 3]. Эти компоненты при таком подходе можно рассматривать, как атомарные модули. Заметим, что не всегда сильносвязанные компоненты на структурных диаграммах свидетельствуют о плохой архитектуре системы. Возможным дефектом архитектуры с поглощающими слоями может стать эффект "пропавшего слоя" – дефектная связь приводит к появлению модулей, которые по смыслу должны находиться на разных слоях.

4. Коррекция (трансформация) архитектуры в интересах ее рефакторинга

В общем случае под рефакторингом понимают процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы. В основе рефакторинга лежит последовательность небольших эквивалентных преобразований, сохраняющих функциональную семантику базового кода.

По ходу трансформаций часто встает задача выявления смысловой нагрузки модулей. Для решения подобных задач зачастую приходится исследовать реальный программный код, анализировать сигнатуры функций и комментарии, а при отсутствии последних и сам код функций. Задача специалиста, вовлеченного в процесс архитектурного рефакторинга, – по возможности минимизировать объем семантического

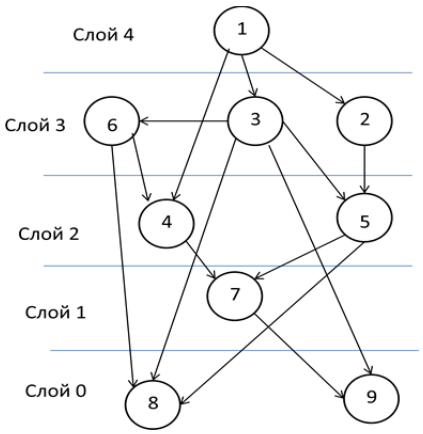

Рис. 1. Граф ПС

анализа (например, путем удаления вспомогательных блоков) и сделать его последовательным и направленным.

Первым уровнем рефакторинга можно считать такое изменение кода, которое не затрагивает структуру модулей (в выше принятом понимании) или классов (количество и взаимосвязи, интерфейсы) объектно-ориентированной программной системы, т.е. рефакторингу подвергается программный код внутри классов. Это могут быть методы и алгоритмы, реализуемые методами, поля и т.п. Второй уровень рефакторинга относится к изменению структуры модулей или классов программной системы, добавлению новых классов, выделению и разбиению больших классов, переносу или добавлению новых методов, выделению интерфейсов и др. Основными стимулами его проведения являются следующие задачи:

- необходимо добавить новую функцию, которая недостаточно укладывается в принятое архитектурное решение программного модуля;
- необходимо исправить ошибку, причины возникновения которой не выделены четко структурированной базовой внешней формой;
- проблематика в командной разработке, которая обусловлена сложностью логики программного продукта.

Следующий, третий, уровень рефакторинга М. Фаулер называет крупным рефакторингом [1]. Вся команда должна осознать, что «в игре» находится один из крупных рефакторингов, и действовать соответственно. Речь о четырех рефакторингах третьего уровня. Это разделение

наследования, преобразование процедурного проекта в объекты, отделение предметной области от представления и выделение иерархии.

Рефакторинг архитектуры программных систем является четвертым уровнем рефакторинга. Необходимость в архитектурном рефакторинге может быть связана со следующими причинами:

1. По мере развития программы в нее вносятся изменения, обусловленные текущей необходимостью. Часто изменения вносят программисты, которые не до конца понимают архитектуру ПС в целом, и постепенно код становится менее структурированным, а разбираться в нем все труднее. Архитектурный рефакторинг улучшает композицию ПС.

2. Повышение производительности ПС. Рефакторинг первого и второго уровней, несомненно, заставляет программу выполняться медленнее, но при этом делает ее более понятной и податливой для настройки производительности.

3. Потребность в функциональных изменениях ПС. Внедрение новой функциональности не должно затронуть логику системы. Изменение существующей архитектуры может быть хорошим шагом на пути внедрения новой функциональности, облегчающим дальнейшую эволюцию системы.

4. Смена платформы ПС. Смена платформы ПС должна минимально затрагивать существующий код. Желательно ограничиться изменениями только в узкой платформенно-зависимой прослойке системы. Выделение такой прослойки всегда сопряжено с необходимостью изменения архитектуры.

5. Обновление технологии разработки программного продукта, связанное, например, с переходом на более совершенную технологию программирования.

6. Преобразования, связанные с реорганизацией компании, ведущей разработку. Например, введение аутсорсинга. Этот шаг зачастую затрудняется проблемой выделения и передачи компонентов для внешней разработки. Изменение архитектуры ПС способно облегчить решение этой задачи.

Прежде чем говорить о коррекции архитектуры, следует задаться вопросом, как оценить качество структуры ПС? Из практики проектирования известно, что лучшее решение обеспечивается иерархической структурой в виде дерева. Степень отличия реальной проектной структуры от дерева характеризуется невязкой структуры. Известно, что полный граф с n вершинами имеет количество ребер равное $e_c = n * (n-1) / 2$, а дерево с таким же количеством вершин – существенно меньшее количество ребер $e_t = n - 1$.

Формулу невязки можно построить, сравнивая количество ребер полного графа, реального графа и дерева. Для проектной структуры с n вершинами и e ребрами невязка определяется по выражению:

$$Nev = \frac{e - e_t}{e_c - e_t} = \frac{(e - n + 1) \times 2}{n \times (n-1) - 2 \times (n-1)} = \frac{2 \times (e - n + 1)}{(n-1) \times (n-2)}.$$

Значение невязки лежит в диапазоне от 0 до 1. Если *Nev* = 0, то проектная структура является деревом, если *Nev* = 1, то проектная структура – полный граф. Ясно, что невязка дает грубую оценку структуры. Для увеличения точности оценки следует применить характеристики связности и сцепления [4].

Вернемся к структуре, приведенной на Рис. 1. Ясно, что модуль 6 не может находиться с модулем 3 в одном слое. Так как модуль 3 для выполнения своих функций обращается к модулю 6, то последний должен быть перемещен в нижележащий слой. Возможный вариант новой структуры показан на Рис. 2. Заметим, что в данном случае модуль 2 должен быть перемещен в нижележащий слой 3. Следует обратить внимание на увеличение количества слоев ПС после выполненной коррекции. Этот важный факт может привести к увеличению времени работы ПС. Отметим также, что после такой коррекции структуры не изменилась сложность ПС, определяемая по значению *Nev*, поскольку число вершин и ребер осталось прежним.

Возможен другой вариант коррекции архитектуры, связанный с объединением модулей 3 и 6 (на Рис. 3 это модуль 3-6). При этом не меняется количество слоев ПС, но изменяется число вершин и ребер (8 вершин и 12 ребер). Это несколько снижает сложность ПС по значению *Nev*. Однако объединенный модуль возрастает по объему и усложняется его программирование.

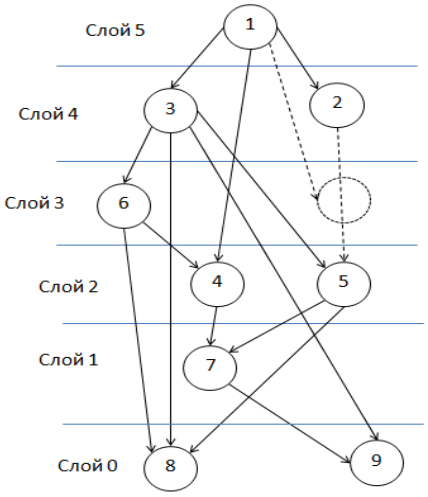

Рис. 2. Первый вариант коррекции архитектуры ПС

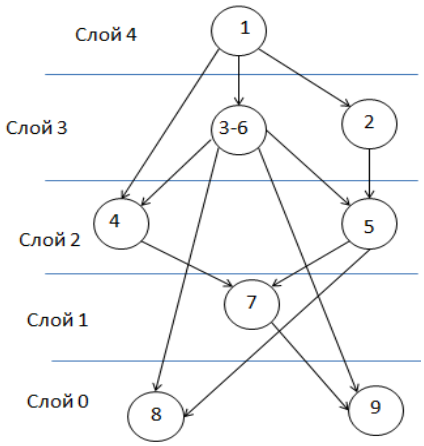

Рис. 3. Второй вариант коррекции архитектуры ПС

Вообще заметим, что формализовать процесс коррекции архитектуры ПС или тем более построить алгоритм коррекции довольно затруднительно. Однако в ряде случаев, выделив отдельные фрагменты структуры ПС, можно их преобразовать, стремясь к получению наилучшей структуры, например, к дереву. Чаще всего это удается сделать путем объединения (поглощения) модулей. Некоторые примеры такой коррекции приведены ниже.

На Рис. 4, а показана последовательная цепочка модулей 2 и 3, которые используются только модулем 1. Может быть, это стало следствием желания распараллелить работу по программированию этих модулей. Возможный вариант улучшения структуры ПС путем объединения модулей 2 и 3 показан на Рис. 4, б). Заметим, что если в этом случае модуль 1 обращается только к модулям 2-3 и 4, то сокращается число слоев ПС.

На Рис. 5, а показан случай, когда результаты работы модулей 1 и 2 используются только модулем 3. Улучшение структуры ПС можно получить объединением модулей 1 и 2, как показано на Рис. 5, б.

Другой случай объединения модулей 2 и 3 приведен на Рис. 6, а. Он возможен в том случае, если к модулям 2 и 3 обращается только модуль 1, а сами модули 2 и 3 обращаются только к модулю 4. В результате объединения модулей 2 и 3, как показано на Рис. 6, б упрощается структура ПС.

При коррекции архитектуры ПС, кроме объединения модулей и перемещения по слоям, возможны ситуации разделения модулей, как показано на Рис. 7. Однако в каждом конкретном случае решение о той или иной коррекции структуры ПС должно приниматься после детального его анализа и оценки целесообразности.

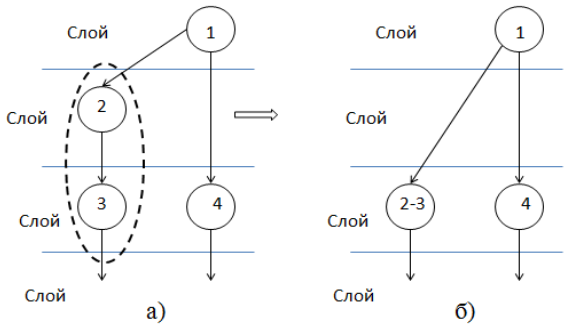

Рис. 4. Коррекция поглощением нижележащим слоем

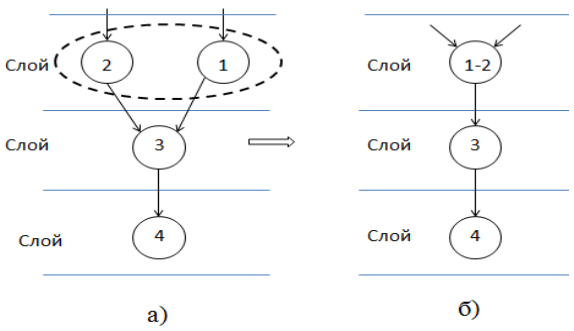

Рис. 5. Коррекция объединением (вариант 1)

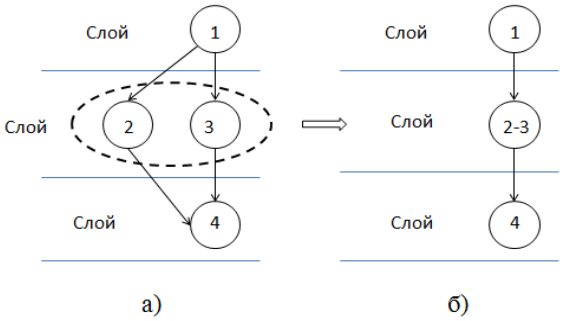

Рис. 6. Коррекция объединением (вариант 2)

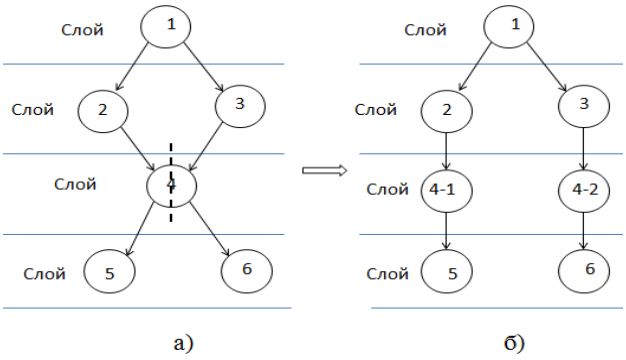

Рис. 7. Коррекция разделением

Результат коррекции архитектуры должен быть спроецирован на реальный программный код системы. При проецировании удаления модулей из модели необходимо определить множество строк и файлов, которое соответствует удаленному блоку в программном коде. После этого необходимо удалить из программного проекта выявленные строки и файлы. При проецировании на код переноса модуля в модели переносятся соответствующие строки и файлы в исходном коде программной системы и т.д. Производимые таким образом трансформации можно рассматривать как архитектурно-управляемый рефакторинг программного кода.

5. Рефакторинг структуры для повышения производительности ПС

5.1. Варианты многослойных структур

Предварительно остановимся на вариантах построения многослойных ПС. На Рис. 8 и Рис. 9 показаны две возможные общие структуры организации слоев программы. С понятием слоев ПС связана концепция многоуровневых виртуальных машин. Именно так Дейкстра рассматривал многослойные программные системы. На Рис. 8 показан подход, когда задача построения архитектуры программы рассматривается как создание “машины пользователя” или виртуальной машины (n), начиная с самого низшего уровня (0) аппаратуры (или, возможно, операционной системы). Последовательность уровней, называемых абстрактными машинами, определяется так, что каждая следующая машина строится на основе предыдущих, расширяя их возможности. Каждый уровень может ссылаться только на один, отличный от него самого, уровень, а именно тот, который непосредственно ему предшествует.

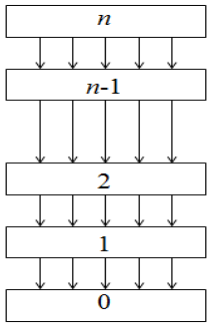

Рис. 8. Вариант классической структуры

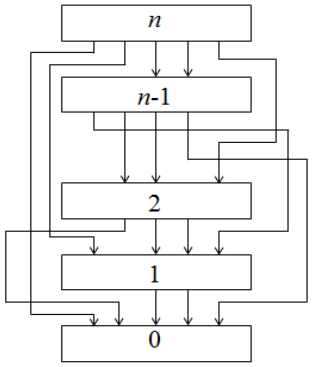

Рис. 9. Вариант структуры со ссылками слоев на все предшествующие

В структуре, изображенной на Рис. 9, уровни не являются полными абстракциями более низких уровней, каждый из них может ссылаться на все предшествующие уровни. Возможен и третий вариант, являющийся промежуточным между двумя первыми. В этом случае слою (i)

разрешается использовать только некоторые из команд, обеспечиваемых слоями (1), (2),…, (i-1). Каждый вариант имеет свои достоинства и недостатки.

Остановимся на особенностях основных вариантов многослойных структур. Если в варианте на Рис. 8 каждый слой имеет доступ к командам только одного слоя, разработчик должен иметь в виду только предыдущий слой. Хотя с точки зрения проектирования этот вариант кажется привлекательным, он может оказаться очень неэффективным. Например, если некоторое средство, предоставляемое слоем (2), потребуется в слое (i), то каждый из слоев (3), (4),..., (i-1) должен обеспечить это средство. Это значит, что запрос данного средства слоем (i) должен “просачиваться” вниз через слой (i-1), пока не достигнет слоя (2), который способен выполнить запрос. Такой подход связан с дополнительными затратами времени на трансляцию запросов [5]. Эти трудности, связанные с проблемой эффективности, могут склонить к принятию структуры по Рис. 2, в которой каждый слой (i), где 2< i < n, может непосредственно обращаться к слою (2).

Таким образом, с точки зрения производительности весьма актуальной становится задача определения оптимальной структуры многослойной ПС.

5.2. Постановка и формализация задачи

Представим структуру многослойной программной системы в обобщенном виде, показанном на Рис. 10. В данном случае каждый слой показан в виде одного модуля с возможностью организации связей с любым произвольным слоем системы. Такая обобщенная схема позволяет рассмотреть любую структуру n-слойной программной системы, лежащую в диапазоне структур, приведенных на Рис. 8 и Рис. 9. Произвольная структура описывается некоторым множеством булевых переменных:

$$X = \{x_{ij} \in \{0,1\} | \, i = n, n-1, \dots, 2; \; j = n-1, n-2, \dots, 1; i > j\}, \quad (1)$$

где $x_{ij} = 1$, если существует связь между слоями i и j, и $x_{ij} = 0$, если такой связи нет. Так как между смежными слоями всегда имеется связь, то

$$(\forall i \,|i = j+1)\,(x_{ij} = 1), i = n, n-1, \dots, 2. \quad (2)$$

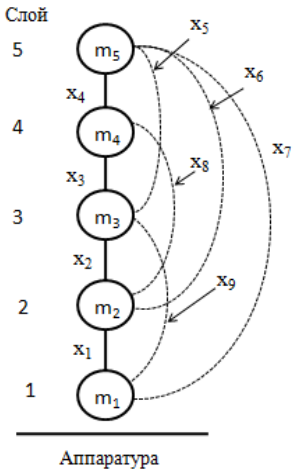

Рис. 10. Структура пятислойной программы

Если в многослойной структуре программы, представленной выражением (1), принимают единичное значение только переменные, описываемые условием (2), то эта программа имеет структуру, соответствующую варианту по рис. 8. Если справедливо условие

$$(\exists \, i \,|(i = n, n-1, \dots, 2)) \& (\exists \, j | (i - j \geq 2))(x_{ij} = 1), \quad (3)$$

то программа имеет структуру, соответствующую промежуточному варианту между вариантами структур, представленными на Рис. 8 и Рис. 9.

Если справедливо условие

$$(\forall \, i \,|i = n, n-1, \dots, 2)\; (\exists \, j \,|(i - j \geq 2))(x_{ij} = 1), \quad (4)$$

то программа имеет структуру, соответствующую варианту, представленному на Рис. 8.

Дальнейшую постановку задачи удобно провести на конкретном примере. Учитывая, что число слоев в большинстве существующих программ, как правило, не превышает трех-пяти, рассмотрим пятислойную программу, структура которой приведена на Рис. 10. В данном случае структура разработанной ПС имеет вариант классической архитектуры, т.е. удовлетворяет условию (2). Пунктирными линиями на Рис. 10 показаны возможные дополнительные связи между слоями ПС. Каждой линии поставлена в соответствие переменная, единичное значение которой означает наличие межслойной связи, нулевое – отсутствие такой связи.

Будем считать, что ПС прошла полный этап тестирования и в процессе отладки определены временные характеристики модулей. Установ-

лены также частоты обращения модулей произвольного слоя к модулям нижележащих слоев. Предположим, что передача (трансляция) запроса через слой i дополнительно (кроме выполнения собственных функций) загружает этот слой на некоторый промежуток времени t_i. Если $x_5 = 0$, (т.е. отсутствует связь между слоями 5 и 3), то модуль m_4 дополнительно работает в течение промежутка времени t_5. Если $x_5 = 1$ (т.е. вводится связь между слоями 5 и 3), то дополнительное время модулю m_4 не потребуется. Однако в этом случае будет необходимо в программу добавить межмодульный интерфейс для взаимодействия модулей m_5 и m_3, который несколько увеличит время работы модулей m_3 и m_5 (на величину t'_5). Таким образом, введение связи $x_5 = 1$ приведет в итоге к сокращению времени работы программы на величину, равную $t_5 - t'_5$, кроме того это увеличит программу на некоторую величину e_5 . Аналогичные рассуждения справедливы и для других переменных, показанных на Рис. 10.

Дополнительные связи между слоями программы сокращают время выполнения ее функций, но увеличивают размер программы. Необходимо также учесть тот факт, что дополнительно создаваемые связи между слоями могут работать с различной нагрузкой. Так, например, если создается связь, обозначаемая переменной x_5, то модуль m_4 освобождается от трансляции только тех запросов, которые модуль m_5 адресует модулю m_3 . Поэтому целесообразно каждой переменной x_i поставить в соответствие определенную интенсивность взаимодействия некоторой пары модулей λ_i.

Таким образом, задача структурного рефакторинга ПС сводится к определению такой структуры многослойной программной системы, которая обеспечивает наилучшую производительность программы при заданных ограничениях на размер дополнительных межмодульных интерфейсов.

5.3. Математическая постановка и решение задачи

В нашем случае структура многослойной программной системы может быть представлена вектором $X = \{x_i | \, i = 5, 6, \dots, 9\}$ (заметим, что всегда $x_1 = x_2 = x_3 = x_4 = 1$, так как эти переменные определяют связи между смежными слоями). Поэтому требуется найти такое значение X_{opt}, при котором обеспечивается максимальный выигрыш во времени работы ПС

$$Max\ T = \sum_{i=5}^{i=9} \lambda_i \times (t_i - t'_i) \times x_i \qquad (5)$$

при выполнении ограничения на допустимое увеличение программы за счет дополнительных межмодульных интерфейсов

$$\sum_{i=5}^{i=9} e_i \times x_i \le E_д, \qquad (6)$$

где $E_д$ – допустимое увеличение размера ПС.

Учитывая двоичный характер переменных, следует добавить ограничение

$$\forall\, i\ (x_i \in \{0,\ 1\}). \qquad (7)$$

Сформулированная задача относится к классу задач линейного программирования с булевыми переменными (в данном случае это задача о загрузке рюкзака). Малая размерность рассмотренной задачи позволяет ее легко решить полным перебором совокупностей переменных, представляющих допустимые решения в условиях принятых ограничений.

Заключение

Кажущаяся простота решения задачи довольно условна. Основная сложность связана с определением исходных данных для решения задачи, т.е. $\{t_i\}, \{t'_i\}, \{e_i\}$, что возможно только при наличии развитых средств профилирования и трассировки программного продукта. Заметим также, что в связи с неточностью получения исходных данных в рассматриваемой задаче, применять методы получения оптимального решения не имеет смысла. Можно ограничиться приближенными, быстро работающими алгоритмами. Результат решения должен быть спроецирован на реальный программный код системы.

Предложенный подход к решению задачи определения оптимальной структуры многослойной программной системы является достаточно упрощенным. Во-первых, в рассмотренном примере каждый слой программы представлен одним модулем. Во-вторых, достаточно сложно на этапе проектирования архитектуры системы получить исходные данные, которые требуются для решения задачи. В-третьих, в реальных программных системах, содержащих по нескольку модулей в каждом слое, размерность задачи может существенно вырасти и потребуется применить более

сложные алгоритмы решения задачи. Однако, несмотря на это, имея аналог проектируемой программы или достаточно полную ее модель, можно получить с некоторой степенью приближения требуемые исходные данные. В любом случае, по мнению автора, изложенный подход, возможно, заставит архитектора задуматься о рациональном выборе структуры многослойной программной системы.

Литература

1. Фаулер М., Бек К., Брант Д., Робертс Д., Апдайк У. Рефакторинг: улучшение существующего кода. – Спб: Символ-Плюс, 2009. – 432 с.
2. Ксензов М. Рефакторинг архитектуры программного обеспечения: выделение слоев. Труды Института системного программирования РАН, препринт 4, 2004, С. 211 – 227
3. Ксензов М. В. Рефакторинг архитектуры программного обеспечения. [Электронный ресурс]: http://www.ispras.ru/ru/proceedings/docs/2004/8/1/isp_2004_8_1_211.pdf
4. Миронов В.О.Применение графов для анализа сложных систем на основе исходного кода программ. [Электронный ресурс]: http://berestneva.am.tpu.ru/Papers/KONF2009/%
5. Назаров С.В. Архитектура и проектирование программных систем. М.: ИНФРА-М, 2013. – 352 с.

Назаров Станислав Викторович. Профессор Российского экономического университета имени Г.В. Плеханова, профессор Финансового университета при правительстве Российской Федерации. Главный научный сотрудник ЗАО «МНИТИ». Действительный член Международной академии информатизации. Окончил в Новочеркасский политехнический институт в 1962 году. Доктор технических наук. Автор более 300 печатных работ. Область научных интересов: операционные системы, исследование эффективности, проектирование и оптимизация программных и компьютерных систем. E-mail: s_nazarov@mail.ru

Вилкова Надежда Николаевна. Президент АРПАТ, заместитель председателя координационного совета по вопросам инновационного развития радиоэлектроники при Департаменте радиоэлектронной промышленности Минпромторга России. Генеральный директор ЗАО «МНИТИ». Окончила Саратовский экономический институт в 1976 году. Кандидат технических наук. Автор 120 печатных работ. Область научных интересов: проектирование сложных систем, оптимизация управления ресурсами. E-mail: mniti@mniti.ru

Structural refactoring of multilayered program systems

S.V. Nazarov, N.N. Vilkova

Abstract. Evolution of difficult program systems demands from the developer special attention to a choice of their architecture. Current situation with design confirms that practically always during development of program systems appear new requirements from the customer and we are to have to reconsider initial architecture. Also miscalculations of performers at system design and coding stages aren't excluded. It causes keen interest in questions of program systems refactoring. At the same time relevance of a problem of program systems architecture design and its refactoring didn't find due reflection in the published literature yet. The insignificant number of works is devoted to questions of architectural refactoring. In the real work is considered the problem of structural refactoring of multilayered program systems with the purpose of systems productivity increase and offered the approach to representation of multilayered program systems structures. And also is given the mathematical problem definition of refactoring in the form of a linear programming problem with Boolean variables and discussed a approach to its decision.

Keywords: multilayered program system, architecture, refactoring, structure, productivity.

Nazarov Stanislav Viktorovich. Professor of Financial University under the Government of the Russian Federation, chief researcher of JSC "MNITI", full member of the International academy of informatization, professor, Doctor of Engineering. Number of printing works: more than 250. Area of scientific interests: operating systems, efficiency research, design and optimization of program and computer systems. E-mail: s_nazarov@mail.ru

Vilkova Nadezhda Nikolaevna. President of "ARPAT", vice-chairman of coordination council on innovative radio electronics development at Department of the radio-electronic industry (Ministry of Industry and Trade of the Russian Federation), Director General of JST "MNITI", PHD in Engineering. Number of printing works: 120. Area of scientific interests: Projection of complex systems, resource management optimization. E-mail: mniti@mniti.ru