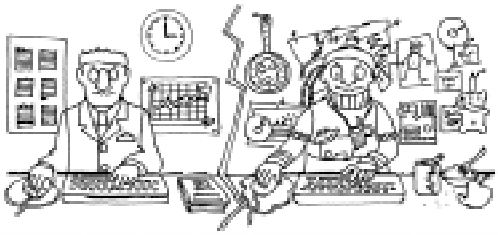
Терехов Андрей Николаевич,
Лен Эрлих

ПРЕОДОЛЕНИЕ РАЗРЫВА МЕЖДУ АКАДЕМИЧЕСКИМ И ИНДУСТРИАЛЬНЫМ ПРОГРАММИРОВАНИЕМ

ВВЕДЕНИЕ

Индустриальное программирование обычно ассоциируется с большими командами программистов, жесткими графиками и отработанными решениями и технологиями. С другой стороны, основная цель академических исследований состоит в том, чтобы искать новые решения и ломать сложившиеся стереотипы. К сожалению, уди-



вительно малый процент научных результатов доходит до промышленных продуктов, а если и доходит, то спустя очень большой промежуток времени.

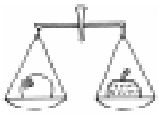
Один из авторов этой статьи – А.Н. Терехов – заведует кафедрой системного программирования СПбГУ и является генеральным директором ЗАО «ЛАНИТ-ТЕРКОМ». Другой автор, Лен Эрлих, – вице-президент компании Bridge-Quest, до этого более 5 лет возглавлял направление новых разработок компании Relativity Technologies Inc. (North Carolina, USA), а всего более 20 лет работает в области индустриального программного обеспечения.

Наше сотрудничество началось в 1991 году, за эти годы мы преодолели множество трудностей, связанных, в основном, с трудностями взаимопонимания, различным менталитетом, но, в конце концов, усилиями многих сотрудников наших компаний и кафедры был создан существенно наукоемкий продукт RescueWare Workbench. Этот продукт по результатам 2000 и 2001 гг. был признан компанией Gartner Group лучшим продуктом в области *legacy understanding* и *legacy transformation* (понимание и преобразование устаревших приложений).

Накопленный опыт, долгие обсуждения и споры дали нам повод представить на конференцию, посвященную памяти академика А.П. Ершова, доклад о проблемах и барьерах, стоящих между наукой *software engineering* и промышленностью программного обеспечения.

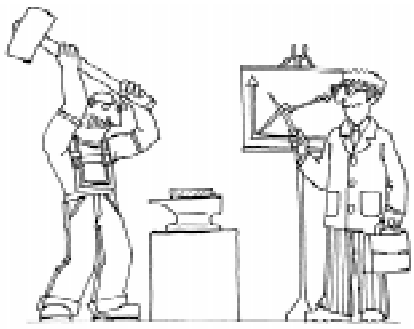
Кстати, мало кто знает, что в последние годы жизни А.П. Ершов уделял большое внимание проблемам промышленного производства программного обеспечения, 2 года работал академиком-консультантом в НИИ «Звезда» ЛНПО «Красная Заря» (г. Ленинград). В традициях того времени эта работа не особенно афишировалась, поскольку НИИ «Звезда» работал в области правительственной связи, поэтому впоследствии даже специалисты, хорошо знавшие А.П. Ершова лично, удивлялись, как это столь известный ученый тратил время на

такие «приземленные» проблемы. Сохранилось несколько документов, собственноручно написанных А.П. Ершовым, в которых он дал свое развернутое представление, в каких направлениях должна развиваться промышленная технология программирования. Удивительно, насколько точно он предугадал многие современные проблемы программирования.



1. РАЗЛИЧИЯ МЕЖДУ *SOFTWARE ENGINEERING* И ИНДУСТРИЕЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В принципе, противоречия между наукой *software engineering* (индустриальная разработка программного обеспечения) и промышленностью ПО мало чем отличаются от противоречий между другими науками и промышленностью. Наука, как ее понимают в университетах, по определению *не прибыльна*, в то время как для промышленности прибыль – основа существования. Семантический разрыв между научными результатами и промышленной разработкой преодолевается, причем с большим трудом, специальными организациями, например, венчурными фирмами. В Советском Союзе эту роль выполняли отраслевые институты, которые именно из-за их промежуточного положения в процессе перехода к рыночной экономике рухнули в первую очередь, что до сих пор негативно сказывается на российской промышленности.



С другой стороны, наука *software engineering* имеет свои особенности, сильно отличающие ее от классической математики или даже от *computer science*. Если доказанная теорема или оценка сложности какого-то алгоритма сама по себе является конечным результатом, причем для его получения необходимы только талант ученого, карандаш и бумага (может быть, следуя техническому прогрессу – еще персональный компьютер с TEX'ом), то в *software engineering* даже новый интересный метод и действующий прототип не гарантируют, что из них получится успешный программный продукт. Нужны большие коллективы, капиталовложения, строгая промышленная организация и многое другое. В этом смысле *software engineering* близка к физике элементарных частиц, сверхнизких температур и т. п.

К сожалению, то, что очевидно для физиков, часто вызывает неприятие университетских ученых-математиков. Это неприятие проявляется во многих ситуациях, начиная от сомнений в принадлежности *software engineering* к математическим наукам. Никакие ссылки на определения и программы обучения ACM или IEEE не помогают. Трудно защитить на Ученом Совете университета кандидатскую диссертацию: очевидно, что просто работающая небольшая программа не является достаточным результатом, а крупная коллективная работа противоречит традиционным требованиям к индивидуальности научного исследования.

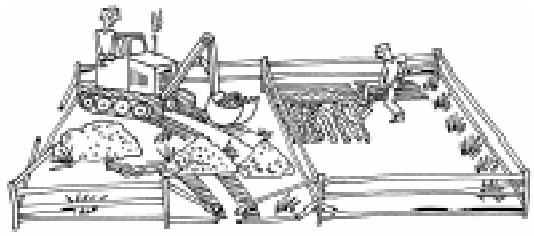
Ничто не ново под луной – еще в 1953 году было принято решение ВАКа о том, что результаты расчетов на только что появившихся ЭВМ не могут быть приняты в качестве кандидатской диссертации. Еще многие годы математики, разработавшие новые эффективные алгоритмы, вынуждены были как-то изворачиваться, представляя свои работы в смежных областях науки, хотя всем было очевидно, что основной результат – это эффективная программа, решающая практическую задачу новым способом.

С другой стороны, очень многие разработки промышленности, не претендующие на научную новизну, вызывают горькую усмешку университетских ученых. Например, в конце 90-х было создано (и, что самое смешное, продано) множество программ, решающих «проблему 2000 года» таким примитивным способом, как сканирование исходного текста программы на предмет

поиска идентификаторов, похожих на обозначение даты. Никто не предупреждал покупателя, что переменная DATE могла быть присвоена другой переменной, передана в качестве параметра, записана в базу данных, а затем прочитана другой программой, работающей в составе этого же приложения, и т. д. Но ведь аккуратное решение требует синтаксического анализа всех программ приложения (возможно, написанных на различных языках), выполнения анализа потоков информации, транзитивного замыкания графа ссылок и т. п. А это уже совсем другая задача, другого уровня сложности, который не каждый коллектив разработчиков может преодолеть. Зачем стараться – рынок и так съест!

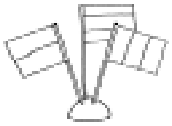
Однако не все так просто. Рынок – это действительно чуткий индикатор. Простейший сканер позволит найти до половины «подозрительных на дату» переменных, а остальные, возможно, дешевле найти вручную, чем покупать сложную и дорогую систему, которая все равно не гарантирует абсолютно корректного решения, так как начинает свою работу с того же поиска «подозрительных» переменных.

Таким образом, при решении каждой нетривиальной и содержательной задачи всегда нужно найти баланс между глубиной научной проработки, обеспечивающей полноту, корректность и эффективность решения, и стоимостью и временем разработки, которые нужно сокращать, даже за счет ослабления каких-то требований.



2. ИНДУСТРИАЛЬНЫЙ ПРИМЕР ВЗАИМОДЕЙСТВИЯ НАУКИ И ПРОМЫШЛЕННОСТИ

Нам хотелось бы проиллюстрировать перечисленные выше трудности и противоречия научного и производственного подходов на примерах и решениях, найденных в процессе разработки системы RescueWare. Эта система автоматизирует реинжиниринг устаревшего ПО, то есть перевод программ, написанных на Коболе, CICS, встроенном SQL, BMS, JCL, PL/I, ADABAS Natural и др., работающих в основном, на IBM mainframes, на языки C++, VB, Java на платформах Windows и UNIX. Реинжиниринг программ не сводится только к трансляции с одного языка на другой – совершенно другие способы организации диалога с пользователем, работа с устаревшими базами данных, восстановление утраченных знаний о программе делают эту задачу намного более трудной и объемной.



2.1. МНОГОЯЗЫКОВОСТЬ И МНОГОПЛАТФОРМЕННОСТЬ

Так как мы сразу ориентировались на создание *многоязыкового* транслятора, то одной из первых наших задач стало проектирование единого промежуточного языка (ПЯ) системы. Идея заключается в том, что преобразование программы происходит в два этапа: сначала она транслируется в ПЯ, а затем в целевой язык. Такой подход позволяет ограничиться в случае M входных и N выходных языков поддержкой $M + N$ трансляторов вместо $M * N$.

Под именем UNCOL этот подход был известен уже более 40 лет назад. Многие коллективы брались за его реализацию, но только единицы сумели довести UNCOL до практической реализации [1; 2]. В чем дело? Ведь с теоретической точки зрения все языки вычислительно эквивалентны, поэтому перевод должен быть простым. Трудности возникают, если мы будем измерять качество порожденной программы не только (и не столько) с точки зрения эффективности, но и с точки зрения естественности структуры программы в том или ином языке.

Каждый язык программирования предоставляет определенные выразительные средства и дисциплину их использования. При этом наибольшим качеством с точки зрения надежности и легкости сопровождения обладают те программы, которые удов-

летворяют ограничениям, накладываемым этой дисциплиной.

Проблема возникает тогда, когда представления о естественной форме программы в одном языке начинают противоречить требованиям естественности ее в другом.

Например, использование безусловного перехода является обычной ситуацией в Коболе, а в языке Java безусловных переходов нет совсем. Для решения этой проблемы могут потребоваться специальные (и порой нетривиальные) преобразования, сильно зависящие одновременно от свойств исходной и целевой платформы.

С точки зрения описания семантики, в любом языке могут быть выделены следующие уровни представления:

1. Представление потока управления.
2. Представление потока данных.
3. Представление значений.

ПЯ, таким образом, должен содержать абстрактные средства для представления программы на всех этих уровнях. При этом для преобразования исходной программы в целевую конструкции исходного языка сначала «поднимаются» до абстрактного промежуточного представления, а затем «опускаются» в конкретное представление целевого языка. Степень абстракции должна быть избрана таким образом, чтобы подобное понижение приводило к естественным с точки зрения целевого языка проекциям.

Важнейшим условием содержательности предлагаемого подхода является *ортогональность* трансляций в ПЯ и из него. Это означает, что представление исходной программы в ПЯ не должно зависеть от того, в какой целевой язык она будет трансформирована далее.

Наконец, ПЯ должен быть расширяемым, то есть содержать в себе средства, которые позволяют строить новые конструкции, не требуя внесения изменений во все работающие с ним просмотры.

Узким местом при разработке ПЯ является выбор типов данных и набора стандартных операций. Если набор конструкций управления разных языков в общем и целом хорошо поддается унификации, то системы типов могут различаться очень сильно. В то же время нецелесообразно просто объединить все типы всех исходных языков, поскольку добавление нового языка будет требовать существенного изменения существующих просмотров.

Для решения этой задачи, помимо обычного набора стандартных типов данных, в ПЯ добавляется формализм более высокого уровня – *конструктор типа*, который можно рассматривать как отображение, сопоставляющее нескольким типам новый тип. Например, в качестве конструкторов типов могут быть рассмотрены такие абстракции, как структуры, массивы и указатели. Действительно, структуру можно определить как конструктор типа, который получает набор типов полей и создает тип структуры с полями соответствующих типов.

Наконец, использование конструкторов типов позволяет легко осуществлять настройку динамической поддержки. Действительно, можно ввести в рассмотрение конструктор типа, трактуемый как некоторый абстрактный динамический тип данных, вся работа с которым осуществляется с помощью функций динамической поддержки. При этом добавление новой сущности будет требовать изменения только одного просмотра – именно, того просмотра, который эту сущность порождает. В дальнейшем вся ее обработка будет производиться совершенно прозрачно с использованием общих механизмов преобразования конструкторов типов.

Нам кажется, что данная задача является классическим примером семантического разрыва между академическими исследованиями и промышленным программиро-

ванием. Идея построения единого ПЯ имеет смысл только в действительно крупных проектах, которые выходят за рамки академических проектов. С другой стороны, у рядового программиста из промышленности просто нет знаний, необходимых для успешной реализации данного подхода.

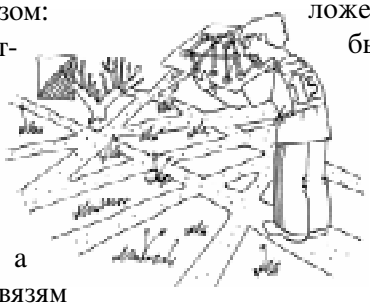
Отметим также, что «естественность» структуры ПЯ, о которой мы говорили выше, является одним из примеров плохо формализуемых понятий, необходимых для решения практических задач. Другим примером трудно формализуемых задач является определение критериев «добротности» программ [3].

2.2 РАЗБИЕНИЕ НА КОМПОНЕНТЫ. CLASS BUILDER

Другая интересная задача, с которой мы столкнулись в процессе создания средства автоматизированного реинжиниринга, это разбиение приложений на компоненты, модули или классы [4; 5].

Эту задачу можно описать следующим образом: есть объемное приложение, состоящее из многих файлов, в каждом из которых содержатся описания переменных и процедур. Переменные и процедуры из разных файлов взаимодействуют друг с другом через какие-то внешние объекты, которые мы назвали *dataports*. Обычно это операторы языка CICS, *embedded* SQL или какие-то другие инфраструктурные элементы.

В общем виде задача была формализована следующим образом: приложение представляется в виде графа, узлами которого являются объекты приложения (они имеют типы – переменная, процедура или внешний объект), а ребра соответствуют связям между ними (ребра также имеют типы – например, вызов одной процедуры из другой, использование переменной в процедуре, работа с внешним объектом через переменную). Кроме того, все ребра нагружены, то есть каждому ребру соответствует некоторое число (мощность), определяющее «силу» связи, например, для связи по вызову между процедурами мощность может определяться числом передаваемых параметров – чем больше передаваемых параметров, тем больше хочется поместить в одну компоненту вызывающую и вызываемую процедуры. Этот граф нам надо разделить на какие-то области «сильной связности». Для этого введем понятие силы притяжения между узлами – это сумма мощностей всех ребер между этой парой узлов, помноженных на коэффициент, определяемый типом ребра, минус некоторая константа, определяемая парой типов узлов.



Какую-то отрицательную часть формулы, то есть силу отталкивания, ввести необходимо, иначе всегда будут получаться единые монолиты. Например, естественным кажется введение силы отталкивания между *dataports* или между любыми процедурами – сначала полагаем, что все процедуры должны попасть в разные компоненты, а уж если две процедуры используют много общих переменных, то силы притягивания перевесят.

Далее полным перебором находим такие группы узлов, для которых сумма сил притяжений между собой максимальна, а с узлами других групп – минимальна. Понятно, что силы притяжений с узлами других групп берутся со знаком минус.

Абсолютно очевидно, что эта красивая идея на практике работать не будет, поскольку число узлов графа в реальных приложениях слишком велико, чтобы можно было воспользоваться переборными алгоритмами. Однако нам удалось найти некоторые эвристические подходы, позволившие добиться практических результатов.

Во-первых, мы зафиксировали определенные коэффициенты для различных типов ребер и сил отталкивания для различных типов узлов. Впрочем, пользователь может расставлять коэффициенты самостоятельно, если он считает это принципиальным для своего приложения.

Во-вторых, в полном графе приложения будем вначале рассматривать только

подграф, состоящий из узлов, соответствующих внешним объектам, и ребер и узлов любых других типов, лежащих на пути между внешними объектами. Назовем этот подграф редуцированным. Эвристика состоит в том, что мы считаем внешние объекты структурообразующими и поэтому вводим силу отталкивания только между ними. С другой стороны, если два внешних объекта используют много общих переменных и процедур, то ничто не мешает им попасть в общую компоненту.

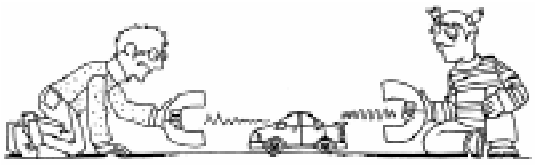
В-третьих, будем считать все ребра редуцированного графа ребрами одного типа, но мощность каждого ребра будем определять так, чтобы она была тем больше, чем больше разных связей (не только непосредственных, но и транзитивных, идущих через другие узлы) соединяет эти вершины. Для формализации этого «больше» воспользуемся физической моделью.

Для вычисления мощности редуцированных графов была предложена модель электрической сети – будем считать, что между любыми двумя узлами исходного графа, между которыми действует положительная сила притягивания, существует провод, проводимость которого положим равной этой силе (напомним что проводимость – это величина, обратная сопротивлению). Тогда в качестве мощности ребра редуцированного графа между двумя вершинами можно взять полную проводимость полученной электрической сети между выбранными вершинами (если у нас есть сеть из проводов, мы можем измерить сопротивление между ними). Полная проводимость вычисляется по уравнениям Кирхгофа, то есть решение сводится к решению системы линейных уравнений, сложность ее решения $N^{2,71}$(сложность обращения матрицы).

В-четвертых, чтобы избежать полного перебора узлов редуцированного графа при определении областей сильной связанности, мы воспользовались еще одной физической моделью.

Считаем, что узлы графа – это электрические заряды с одним полюсом, а ребра графа – это упругие соединения. Практики знают, что интересующая нас задача разбиения графа на подграфы с заданной оценкой качества разбиения имеет примерно такое же решение как задача поиска расположения на плоскости зарядов, связанных упругими соединениями, с целью минимизации общей энергии. Кстати, этой моделью мы часто пользуемся, чтобы улучшить размещение на экране каких-либо графов. Задача минимизации энергии решается методом, аналогичным методу «тяжелого шарика». Достижения абсолютного минимума этот метод не гарантирует, но этого и не требуется.

Итак, редуцированный граф мы разбили на компоненты. Теперь нам надо вернуться назад к полному графу. Для этого использовался алгоритм, который мы условно назвали «кристаллическим» – в некотором смысле мы моделируем процесс кристаллизации (например, превращения воды в лед). Как известно, абсолютно чистая вода даже при очень низкой температуре в лед не превратится – для начала образования кристалликов льда нужны какие-то неоднородности, которые называются центрами кристаллизации. Будем считать такими центрами группы выбранных узлов, получившиеся в результате разбиения редуцированного графа. Будем по очереди рассматривать оставшиеся узлы и присоединять каждый из них к тому кристаллу, к которому его сильнее притягивает, при этом при вы-



числении силы притяжения мы учитываем все узлы, которые уже есть в кристалле, а не только исходную «затравку». Понятно, что в такой ситуации результат в общем случае может зависеть от порядка, в котором мы рассматриваем узлы. Поэтому мы проводим процесс несколько раз, выбирая узлы в случайном порядке, а затем фиксируем узлы в тех центрах, куда они попадают чаще всего.

2.3. ПОСТРОЕНИЕ СРЕЗОВ ПРОГРАММ

Допустим, устаревшее приложение выполняет 10 функций, семь из которых уже не нужны, но три активно используются, причем, как часто бывает с устаревшими программами, никто не знает, *как* эти три функции работают. Возникает задача создания средства для глубокого анализа старых программ, которое может помочь сопровождающему программисту найти и выделить из сложной программы необходимую ему функциональность, поместить соответствующую часть исходной программы в отдельный модуль и переиспользовать в дальнейшем, например, для перевода на современную языковую платформу.

Решение этой задачи основано на построении статических срезов программ, а также их модификаций, позволяющих использовать механизм срезов для анализа различных аспектов архитектуры исходного приложения, в том числе и для системного анализа.

Под срезом программы по данному оператору понимается подмножество операторов программы, образующих для оператора тот же контекст выполнения, что и вся программа. Иначе говоря, срез – это программа, содержащая данный оператор и еще какие-то операторы исходной программы, а именно те, от которых он зависит.

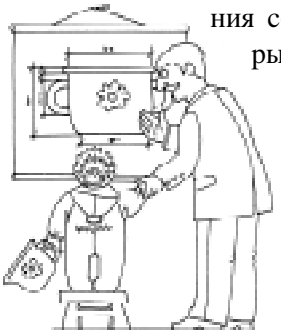
Можно ли минимизировать срез? В своей работе [6] Вейзер показал, что алгоритм построения минимального статического среза является невычислимым. Однако существует ряд алгоритмов, которые строят статический срез, в большинстве случаев совпадающий с минимальным.

В [7] было предложено усовершенствование статического среза для задач отладки, названное авторами динамическим слайсингом. Помимо уменьшения размеров среза, динамический слайсинг делает возможным анализ индексов массивов и переменных-указателей.

Большинство алгоритмов построения статических срезов работают на графах потоков управления и графах зависимости по данным. Алгоритмы построения и анализа этих графов хорошо известны с 60-х годов, в частности, благодаря трудам новосибирских ученых школы А.П. Ершова.

Тривиальные алгоритмы построения статического среза, состоящие в обходе графов от конца к началу, не корректны из-за операторов безусловной передачи управления и вызовов процедур.

На практике все же удается кое-что сделать. Во-первых, предварительно выполняются работы по элиминации *goto* и процедурной реструктуризации, так как это и само по себе полезно для улучшения понимаемости программы и облегчения сопровождения. Во-вторых, мы не считаем зазорным привлечь к решению пользователя, для чего нами реализовано мощное средство HyperCode [8], которое дает пользователю адекватное представление о программе в легко воспринимаемом виде (навигация по структуре программы в виде ее синтаксического дерева с одновременным отображением соответствующих фрагментов исходного текста, отношения между различными конструкциями программы, такими как «описание–использование» и т. д.).



Для автоматизации получения срезов в RescueWare реализованы следующие методы ([9]):

– вычислительно-ориентированное вычленение бизнес-правил;

– доменно-ориентированное вычленение бизнес-правил;

– структурно-ориентированное вычленение бизнес-правил;

– глобальное вычленение бизнес-правил.

Все эти методы предполагают порождение синтаксически корректных автоном-

ных программ, во всех деталях сохраняющих семантику соответствующих фрагментов исходной системы.

Вычислительно-ориентированное вычленение бизнес-правил (ВБП) формирует функциональный срез программы, исходя из того, какой путь исполнения операторов и какие определения данных требуются для вычисления значения заданной переменной в определенной точке программы. Основанное на алгоритмах анализа потоков данных, вычислительно-ориентированное ВБП лучше всего подходит для того, чтобы вычленить бизнес-правило, связанное с вычислением заданной деловой характеристики, исходя из местоположения результата этого вычисления в программе. Так, например, вычислительно-ориентированное ВБП вычленит из программы алгоритм вычисления процентной ставки по долговому обязательству при условии, что пользователь укажет на нужную переменную и на такой оператор в программе, после выполнения которого в переменную уже заслан результат вычисления.

Операция доменно-ориентированного вычленения бизнес-правил порождает функциональный срез программы, получающийся из нее при фиксированном значении одной из входных переменных. Основанное на теории специализации программ доменно-ориентированное ВБП лучше всего подходит для того, чтобы разделить вычисления, характеризующиеся наличием множества транзакций и смешанного ввода, на ряд «узко специализированных» бизнес-правил, каждому из которых соответствует однаединственная транзакция. Например, программа, работающая с многочисленными типами кредитных карт, может быть разбита на несколько бизнес-правил, каждое из которых «занимается» своим особым типом кредитной карты, и представлена как их группа.

Структурно-ориентированное вычленение позволяет разбить программу, написанную в виде сплошного «полотна», на ряд независимых бизнес-правил, исходя из ее физической структуры. Вдобавок создается дополнение к головной программе для того, чтобы вызывать извлеченные бизнес-правила последовательно, в правильном порядке и с использованием верных параметров вызова. При этом гарантируется семантическая эквивалентность выполнения последовательного вызова компонент и выполнения исходной сплошной программы. Этот метод лучше всего подходит для того, чтобы разбивать старые программы большого размера на такие части, с которыми было бы легко обращаться.

Наконец, глобальное вычленение позволяет сразу применить ко многим программам приложения любые из перечисленные выше методов и тем самым фактически поддерживает вычленение исходных бизнес-правил в масштабе системы.

Несмотря на автоматизацию, выбор точек приложения и последовательности использования различных методов вычленения бизнес-правил остается, разумеется, за человеком-аналитиком, производящим декомпозицию исходной системы. Так, естественными начальными точками применения вычислительно-ориентированного ВБП являются места в программе, где производится запись вычисленных значений в базу данных или вывод их на экран. Понятно, что осмысленный выбор кандидатов на роль бизнес-правил требует знаний о характере выполняемых системой бизнес-функций.

При рассмотрении реальных проектов часто оказывается, что используемые методы меняются от модуля к модулю; более того, часто оказывается полезным последовательно применять различные методы ВБП к одной и той же программе.

Литература

1. Проект БЕТА. Отчет ВЦ СО АН СССР.

2. GCC home page. http: //www.gnu.org/software/gcc/gcc.html

3. *I.V. Pottosin* A «Good Program»: An Attempt at an Exact Definition of the Term // Programming and Computer Software. Vol. 23. № 2, 1997. P. 59–69.

4. *А.А. Терехов.* Автоматизированное разбиение устаревших систем на классы // Автоматизированный реинжиниринг программ / под ред. *А.Н Терехова* и *А.А. Терехова.* Изд. СПбГУ, 2000. С. 229–250.

5. *S. Jarzabek, P. Knauber.* Synergy between Component-based and Generative Approaches, In Proceedings of ESEC/FSE'99, Lecture Notes in Computer Science № 1687, Springer Verlag. P. 429–445.

6. *M. Weiser.* Program Slicing // IEEE Transactions on Software Engineering. July 1984, № 4. P. 352–357.

7. *T. Ball, S. Horwitz.* Slicing Programs with Arbitrary Control Flow // Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging. 1993.

8. *М.А.Бульонков, Д.Е. Бабурин.* HyperCode – открытая система визуализации программ // Автоматизированный реинжиниринг программ / под ред. *А.Н Терехова* и *А.А. Терехова.* Изд. СПбГУ, 2000. С. 165–183.

9. *А.В. Друнин.* Автоматизированное построние программных компонентов на основе устаревших программ // Автоматизированный реинжиниринг программ / под ред. *А.Н Терехова* и *А.А Терехова.* Изд. СПбГУ, 2000. С. 184–205.

Терехов Андрей Николаевич,
доктор физ.-мат. наук, профессор,
зав. кафедрой системного программирования СПбГУ,
директор НИИ Информационных Технологий СПбГУ.

Эрлих Лен,
кандидат физ.-мат. наук,
вице-президент компании «Bridge-Quest» (США).