

# ОБЩИЕ ВОПРОСЫ

## Моделирование SOA: Часть 1. Идентификация сервисов



Джим Амсден,  
инженерно-технический работник, IBM

Источник: сайт представительства корпорации IBM в России, раздел «developerWorks Россия»,

[http://www.ibm.com/developerworks/ru/library/1002\\_amsden/index.html](http://www.ibm.com/developerworks/ru/library/1002_amsden/index.html), 13.12.2007

Уровень сложности: простой

*Перед вами первая из пяти статей серии, посвященной разработке программного обеспечения на основе сервис-ориентированной архитектуры (service-oriented architecture, SOA). В статье демонстрируется использование UML-моделей и профиля программирования сервисов IBM® Software Service Profile для разработки SOA-решений, учитывающих бизнес-требования, но независимых от реализации. Автор описывает цели и задачи бизнеса и бизнес-процессы, разработанные для решения этих задач и достижения целей, а затем объясняет, как можно использовать описанные процессы для идентификации значимых для бизнеса сервисов, необходимых для выполнения требований, соответствующих этим процессам.*

### Как моделирование способствует повышению эффективности SOA

Эффективность SOA заключается в способности обеспечить динамичность бизнеса посредством интеграции и многократного использования бизнес-процессов. В SOA это достигается двумя способами: путем поддержки решений, созданных на основе многократно используемых сервисов, инкапсулирующих функциональные возможности, изолированные от конкретной реализации, и предоставления средств для управления связностью между отдельными функциональными возможностями. Связующим звеном между бизнес-требованиями и развертываемым решением, основанным на использовании сервисов, может стать моделирование. Модели SOA переводят разработку на более высокий уровень абстракции, что позволяет сосредоточиться на бизнес-сервисах. Принципы управляемой моделями

разработки могут использоваться для создания реализаций SOA при помощи таких платформ, как Java™ 2 Platform, Enterprise Edition (J2EE) или IBM® CICS®, которые позволяют выполнить функциональные и нефункциональные требования и в то же время способствуют динамичности бизнеса.

Термин “сервис-ориентированная архитектура” (service-oriented architecture, SOA) используется в нескольких значениях. Специалисты-практики обычно используют термин SOA как для определения стиля архитектуры, так и для описания общей ИТ-инфраструктуры, поддерживающей функционирование созданных при помощи этого архитектурного стиля ИТ-систем. Такое толкование с точки зрения технологий является полезным, но неполным.

Чтобы потенциал ИТ-инфраструктуры на базе SOA (далее просто SOA) проявился в полной мере, ей необходимо быть значимой для бизнеса, то есть она должна управляться бизнесом и быть предназначенной для поддержки бизнеса. Нам необходим способ разработки SOA-решений, ассоциируемых с бизнес-требованиями, которые они удовлетворяют. Но если бизнес-требования даны в виде обычного списка элементов, а уровень абстракции SOA документирован в нескольких XML-документах, описывающих коллекцию Web-сервисов, этого довольно трудно добиться.

На самом деле, нам необходим метод для формального описания бизнес-требований; это позволит перейти на более высокий уровень абстракции, благодаря чему реализация SOA будет в большей степени похожа на бизнес-сервисы, а нам станет понятно, как эти сервисы смогут решать бизнес-задачи и обеспечивать достижение целей бизнеса.

Это свяжет размещаемое решение с его предполагаемой ценностью для бизнеса. В то же время нам нужен метод для изоляции бизнес-задач от постоянно изменяющихся SOA-платформ, которые их поддерживают.

Решить эти задачи можно с помощью моделирования и использования принципов разработки, управляемой моделями (model-driven development, MDD). Модели позволяют абстрагироваться от деталей реализации и сосредоточиться на тех проблемах, от которых зависят архитектурные решения. До некоторой степени описываемый нами подход применяет к разработке SOA-решений один из фундаментальных принципов SOA: изоляция проблем и слабая связанность. В этом материале мы четко разграничиваем задачи и обязанности бизнес-аналитиков и сотрудников отделов информационных технологий.

Обычно бизнес-аналитики концентрируются на организационных и операционных требованиях бизнеса, необходимых для достижения бизнес-целей и решения задач, создающих определенную бизнес-концепцию. Часто они не имеют представления (вследствие недостаточной подготовки этих специалистов) об ИТ-задачах, например, многократном использовании, сцеплении и связности, распределенности, обеспечении безопасности, персистентности, целостности данных, параллелизме, восстановлении после сбоев и т. д. Кроме того, инструменты моделирования бизнес-процессов не часто включают функции, необходимые для решения этих задач, а если такие функции и присутствуют, то они вряд ли могут эффективно использоваться бизнес-аналитиками.

### О данной серии, посвященной моделированию SOA

В статье демонстрируется использование UML-моделей и профиля программирования сервисов IBM® Software Service Profile для разработки SOA-решений, учитывающих бизнес-требования, но независимых от реализации. Как правило, человек лучше усваивает идеи, изучая конкретные, типичные и готовые примеры. Здесь мы как раз и используем этот подход. Мы не будем тратить много времени на подробное описание UML, а сосредоточимся на том, как использовать UML в проектировании и разработке.

Хотя серия статей в целом посвящена созданию Web-сервисов из моделей SOA, для начала мы расскажем о бизнес-задачах, которые попытаемся решить, чтобы сделать основой решения некоторую ценность для бизнеса. Затем мы изучим бизнес-процессы, объясняющие, что необходимо сделать в данном бизнесе, чтобы решить эти задачи. На основе полученной информации создаются функциональные бизнес-требования, которые должно выполнять SOA-решение. Затем мы используем эти бизнес-процессы как вспомогательную информацию для идентификации сервисов, которые будут полезны для создания SOA-решений, удовлетворяющих этим требованиям.

Изучая статьи этой серии, мы будем создавать спецификации и реализации сервисов, которые будут обеспечивать выполнение описанных требований при помощи архитектуры, обеспечивающей многократное использование этих сервисов в будущем и динамичность бизнеса. И наконец, мы воспользуемся принципами MDD для создания Web-сервисов, которые можно будет развернуть в производственной среде и выполнить.

Чтобы добиться еще большей реалистичности решения, мы будем использовать имеющиеся инструменты IBM® Rational® и IBM® WebSphere® для создания артефактов решения и связи их между собой, благодаря чему мы сможем проверить соответствие нашего решения требованиям и более эффективно управлять изменениями. В таблице 1 перечислены все процессы, которые мы будем использовать при разработке примера, и инструменты для создания артефактов.

Роль	Задача	Инструменты
Бизнес-исполнитель	Формулирование бизнес-задач и целей	Rational RequisitePro
Бизнес-аналитик	Анализ бизнес-требований	WebSphere Business Modeler
Разработчик архитектуры программного обеспечения	Разработка архитектуры решения	Rational Software Architect
Разработчик Web-сервисов	Реализация решения	IBM® Rational® Application Developer и IBM® WebSphere® Integration Developer

Таблица 1. Роли, задачи и инструменты процесса разработки

В этой серии статей мы сосредоточимся на том, как собрать бизнес-требования, создать модели сервисов, которые будут обеспечивать их выполнение, создать и разместить решения, воплощающие в жизнь эти проекты. Мы также расскажем об инструментах, которые помогут нам сделать это. В статьях будет описан минимальный набор функций моделирования SOA, предлагаемых в настоящее время инструментами IBM, которые перечислены в таблице 1; эти функции можно использовать для моделирования требований потребителя и поставщика сервиса на любом архитектурном уровне. Мы не будем рассказывать обо всех методах или технологиях сбора требований, принципах анализа и оценки реализаций сервисов или секционирования сервисов на различных архитектурных уровнях. Более подробную информацию по этим важным темам можно найти в IBM® Rational Unified Process® (RUP®) for SOA и RUP for SOMA (см. ссылки в разделе Ресурсы). Эти подключаемые модули для IBM® Rational® Method Composer содержат процессы разработки, инструкции, обучающие материалы по использованию инструментов и статьи, в которых демонстрируются дополнительные методы использования инструментов для разработки моделей сервиса и решений

### Пример. Обработка заказов на приобретение

Наш пример построен на примере Purchase Order Process (Обработка заказов на приобретение) из OMG UML Profile и Metamodel for Services (UPMS) RFP и примере из спецификации BPEL 1.1 (см. раздел Ресурсы). Спецификация BPEL 1.1 содержит только часть решения, потому что в ней не определены коррелирующие наборы, бизнес-

данные не отличаются полнотой и, кроме того, в решении не предусмотрена обработка или компенсация ошибок. В данной версии примера добавлены некоторые данные для уточнения решения - в частности, данные, необходимые для корреляции.

Мы начнем с того, что воспользуемся инструментом IBM® Rational® RequisitePro® для того, чтобы описать бизнес-мотивацию для утверждения бизнес-задач, которые необходимо выполнить, и целей, которые должны быть достигнуты. После этого мы расскажем о высокоуровневом бизнес-процессе, записанном при помощи IBM® WebSphere® Business Modeler и выражающем организационные и функциональные бизнес-требования, необходимые для обеспечения решения задач и достижения целей. Эти мотивации и модели процессов создают контекст для идентификации сервисов, отвечающих бизнес-требованиям.

Поняв бизнес-требования, мы сможем продолжить разработку сервисов, обеспечивающих выполнение этих требований. Первый шаг в разработке SOA-решения - идентификация сервисов. Для этого мы будем рассматривать бизнес-процесс как контракт для требований к сервису. После этого разрабатываются спецификации и поставщики сервисов, которые объединяются способом, обеспечивающим выполнение бизнес-требований и одновременно решающим задачи архитектуры программного обеспечения.

Описанный процесс идентификации подходящих сервисов по бизнес-модели также известен как декомпозиция предметной области. В IBM® Rational Unified Process® (RUP®) SOMA описывается метод декомпозиции предметной области и некоторые другие методы, которые, если использовать их в сочетании друг с другом, обеспечивают повышенную точность идентификации всех сервисов, необходимых бизнесу.

### Бизнес-требования

Сценарий: Некий консорциум компаний принял решение об организации совместной работы по созданию сервиса многократного использования для обработки заказов на приобретение. Задачи проекта:

- Утвердить общие средства обработки заказов на приобретение;
- Обеспечить своевременную обработку заказов и доставку заказанных товаров;
- Способствовать минимизации наличного складского запаса и расходов на обслуживание склада;
- Минимизация расходов на производство и отгрузку.

---

*Полностью эта статья находится на сайте представительства корпорации IBM в России по адресу [http://www.ibm.com/developerworks/ru/library/1002\\_amsden/index.html](http://www.ibm.com/developerworks/ru/library/1002_amsden/index.html)*

## Почему американская информатика кажется неизлечимой (Why American Computing Science seems incurable, by Edsger W. Dijkstra)

Эдсгер Дейкстра

Источник: русский перевод <http://club.shelek.ru/viewart.php?id=167> на сайте клуба программистов «Весельчак.У», 11/02/2004

Оригинал статьи "Why American Computing Science seems incurable " (EWD1209) размещается по адресу <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD12xx/EWD1209.html>

Факсимиле рукописного оригинала -

<http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1209.PDF>



Обсуждая, как чувствует себя академическая информатика в США, мы должны уделять больше внимания компьютерной индустрии, чем в других странах, и должны мы это делать по двум причинам. Во-первых, в сравнении с другими странами, здесь нет барьера, отделяющего американский студенческий городок от окружающего общества, во-вторых, большая часть, если не вся, мировой компьютерной индустрии сосредоточена в США. Следовательно, мнения и предрассудки, господствующие в американской компьютерной индустрии, для академической информатики здесь неоспоримы, невзирая на то, являются ли они движущими вперед или парализующими.

Ради достижения стабильности предприятия идеал для менеджера - организация, максимально независимая от индивидуальных способностей служащих. Преобладание этого идеала - это хорошо документированный международный феномен. Этот идеал возник в индустрии высоких технологий, в которой он чрезвычайно неуместен и отталкивает от сотрудничества промышленников с учеными, особенно с блестящими и оригинальными учеными. Американская ситуация усугубляется общим неверием в ее образовательную систему и глубоко укоренившимся недоверием к интеллектуалам.

Кстати. Насколько типично американским является это недоверие, хорошо иллюстрируется фактом, что американское слово "яйцеголовый" не переводится на голландский язык. Я поискал его в моем "Websters New Collegiate Dictionary" (1973), выдержка из которого относится

к непереводаемому слову: яйцеголовый сущ.: интеллектуал, далекий от жизни ученый <с точки зрения практических людей, которые свысока относятся к планам и мечтам "яйцеголовых" W.L.Miller>.

По голландским меркам, цитата из Миллера является однозначной дискредитацией "практичных людей". (Конец Кстати).

В соответствии с этим, повышенное внимание со стороны индустрии оказывает сильное давление на университеты - не поощрять увлечения вроде академического образования, а углубляться в профессиональную подготовку того или иного рода.

Но в случае информатики индустриальные предрассудки влияют не только на преподавание, исследования страдают от них ничуть не меньше. Например, я припоминаю одного так называемого "кандидата в преподаватели", который докладывал о своей диссертации на звание доктора философии - так было принято, - темой которой являлась система "распараллеливания" весьма скромного класса Fortran-программ. Докладчик не претендовал на какое-либо новое видение предмета или на то, что вы могли бы научиться чему-то интересному, изучая его тезисы. Единственное обоснование его работы заключалось в ее конечном продукте, а именно - программном обеспечении для распараллеливания Fortran-программ. Это программное обеспечение было весьма важно, поскольку (i) имеются тысячи Fortran-программ и (ii) система является полностью

автоматической - абсолютное требование для ее принятия промышленностью. (Я не выдумал все это!)

Я думал, что основной критерий, по которому должны оцениваться наши академические исследования, - это насколько они улучшают учебные материалы, но этот бедняга принял в качестве критерия качества "применимость в промышленности", и в результате его творение работало в таких узких рамках условий, что его основным достоинством стала бездумная легкость применения.

Есть и другие примеры того, насколько исследования страдают от давления промышленности.

Быть лучшим программистом - означает быть способным разрабатывать более эффективные и достоверные программы и знать, как делать это эффективно. Это относится к экономному использованию ячеек памяти или машинных циклов, а также к избеганию сложностей, которые увеличивают количество рассуждений, необходимых для удержания строгого интеллектуального контроля над разработкой. То, что, по моему мнению, для этого нужно, - это совершенствование математических навыков, при этом я употребляю слово "математика" в смысле "искусство и наука эффективных рассуждений". В действительности задачи разработки высококачественных программ и построения высококачественных доказательств весьма сходны, настолько сходны, что я уже не могу их различать: я не вижу осмысленных различий между методологией программирования и математической методологией в общих чертах. Короче говоря, повсеместное вторжение компьютеров сделала способность к применению математического метода важнее, чем когда-либо.

По жестокой шутке истории, впрочем, американское общество выбрало именно двадцатое столетие для того, чтобы становиться все более и более нематематическим (кстати, явление, рассмотренное Моррисом Клайном и вызвавшее у него глубочайшее сожаление). Мы достигли парадоксального состояния, когда из всех так называемых "развитых наций" именно США сильнее всех зависят от программируемых компьютеров и хуже всех интеллектуально оснащены в данном направлении. Предположение о том, что проблема программирования может быть вылечена математическими средствами, мгновенно отвергается как совершенно нереалистичное.

В результате Разработка Программ ограждена от возможности стать поддисциплиной Информатики. Имеет место значительная озабоченность корректностью, но она почти полностью направлена на верификацию программ a posteriori, потому что опять же это легче укладывается в мечту о полной автоматизации. Но, разумеется, многие рассматривают верификацию a posteriori как установку телеги впереди лошади, потому что процедура "сначала программирование, потом проверка" поднимает насущный вопрос, откуда берется программа, подвергающаяся

проверке. Если же она выведена, то верификация сводится к простой проверке вывода. А между тем методология программирования, переименованная в "программную инженерию", стала настоящим раем для гуру и знахарей.

Будучи лишенной того, что обычно рассматривается как основное направление компьютерной науки, американская информатика превратилась в большого неудачника. И мы не вправе винить в этом университеты, поскольку, когда промышленность, наиболее нуждающаяся в их научной помощи, неспособна понять, что это высокотехнологичный бизнес, лучшие университеты оказываются бессильны. Университетам следует быть более просвещенными, чем их окружение, и они способны на это, но не очень (точнее, стараются этого не показывать). В нынешней политической ситуации непохоже, чтобы что-то улучшилось вскоре; в ближайшем будущем нам придется жить среди предрассудков, что программирование - это "настолько просто, что им могут заниматься даже члены республиканской партии".

*Остин, 26 августа 1995*

*Prof. dr. Edsger W. Dijkstra*

*Department of Computer Sciences*

*The University of Texas at Austin*

*Austin, TX 78712-1188*

*USA*

---

*От редакции OM/RE: публикуемая фотография Prof. dr. Edsger W. Dijkstra (©2002 Hamilton Richards) взята с сайта <http://www.cs.utexas.edu/users/EWD/>*

## Два взгляда на программирование (Two views of programming by Edsger W. Dijkstra)

Эдсгер Дейкстра

Источник: русский перевод <http://club.shelek.ru/viewart.php?id=211> на сайте клуба программистов «Весельчак.У», 14 августа 2004

Оригинал статьи “Two views of programming” (EWD540) размещается по адресу <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD05xx/EWD540.html>

Машинописное факсимиле оригинала - <http://www.cs.utexas.edu/users/EWD/ewd05xx/EWD540.PDF>



В окружающем нас мире мы можем встретить два радикально противоположных взгляда на программирование:

- Взгляд А: Программирование в основном весьма просто.
- Взгляд В: Программирование – это очень сложно.

Это противоречие можно объяснить тем, что в этих двух взглядах одно и то же слово «программирование» употребляется в двух совершенно различных значениях, и на этом успокоиться. Тем не менее то, какой из взглядов преобладает, А или В, оказывает глубокое влияние не только на кадровую политику организаций, использующих компьютеры, и учебные программы высших учебных заведений, но даже и на направление развития и исследований в самой компьютерной науке. Таким образом, представляется важным исследовать природу различий между этими двумя смыслами и по возможности выявить базовые предположения, которые делают каждый из них применимым. В этом и есть назначение данного документа.

В этом исследовании у меня есть одно препятствие: в этой дискуссии я не являюсь нейтральной стороной. Я – убежденный сторонник взгляда В и рассматриваю взгляд А как основную причину многих печальных заблуждений. С другой стороны, я не считаю, что наличие собственного мнения дисквалифицирует меня как автора, особенно если я заранее предупрежу об этом своих читателей и не буду притворяться нейтральным. В процессе анализа мы раскроем, как эти различные взгляды на программирование (которое является человеческой деятельностью!) связаны с различными мнениями Человека. Это уже само по себе является весьма ценным пониманием, так как объясняет почти религиозное рвение, с которым разворачиваются

сражения между сторонниками противоположных взглядов (догм?).

Ранняя история автоматических вычислений делает взгляд А очень понятным. До того, как у нас появились компьютеры, программирование вообще не являлось проблемой. Затем появились первые машины: по сравнению с нашими нынешними машинами они были просто игрушками, и по сравнению с тем, что мы пытаемся делать сейчас, они использовались лишь для «микро-приложений». Если на этом этапе программирование и было проблемой, то весьма незначительной. Добавьте к этому источники трудностей, которые в то время поглощали – или лучше сказать узурпировали? – большую часть нашего внимания:

1. Арифметические устройства были слишком медленные по отношению к тому, что мы хотели делать с их помощью: эти башмаки почти всегда оказывались слишком тесными, и ради эффективности программы допускались все возможные трюки кодирования (и очень немногие из них реально не использовались).
2. Разработка и конструирование арифметических устройств были настолько новой и, следовательно, трудной задачей, что, если очередная аномалия в коде инструкции могла избавить от каких-либо кульбитов, от них обычно избавлялись, – также, разумеется, потому что мы имели так мало опыта в программировании, что мы не могли так уж хорошо распознавать «аномалии в программном коде»; в результате, помимо необходимости использования трюков в коде, также была великолепная возможность их применять.
3. Памяти всегда было слишком мало, и это вместе с ненадежностью первого оборудования препятствовало



более разумному использованию машин.

В это время программирование представлялось в первую очередь как битва с ограничениями машины, битва, которую нужно было выиграть хитростью. Это было систематическое использование специфических особенностей каждой машины: это был расцвет виртуозного кодирования.

В течение следующих десяти-пятнадцати лет процессоры стали в тысячи раз быстрее, памяти стало в тысячи раз больше, и языки программирования высокого уровня вошли в обиход. И именно в это время с одной стороны программирование все еще прочно ассоциировалось с тесными башмаками, в то время как с другой – чувствовалось, что башмаки жмут все меньше и меньше, и ожидалось, что еще через пять лет технического прогресса проблемы программирования вообще исчезнут. Именно в этот период появился взгляд А. Именно в конце этого периода, вдохновленный взглядом А, был разработан COBOL с провозглашенным намерением, что он должен сделать программирование, выполняемое профессиональными программистами, ненужным, позволив «пользователю» (не в это ли время слово «пользователь» стало общеупотребимым?) записывать то, что он хочет, на «простом английском», который любой может прочесть и понять.

Все мы знаем, что эта прекрасная мечта не воплотилась в жизнь. Следующие пять лет принесли нам вместо исчезновения всех проблем, связанных с программированием, кризис программного обеспечения, и COBOL, вместо того чтобы выжить профессиональных программистов, стал грандиозным механизмом программирования для еще большего их числа; и еще десять лет спустя мы все еще имеем машины, в которых ошибки базового программного обеспечения вызывают в среднем час простоя на каждые пятнадцать часов работы. Очевидно, что до сих пор остались серьезные проблемы программирования...

Забавная вещь: несмотря на полную очевидность обратного, взгляд А все еще жив. В объяснение этого странного факта некоторые с укоризной указывают пальцем на большие организации: либо на организации, применяющие компьютеры, которые, привлекая большие трудовые ресурсы, разделяющие взгляд А, тем самым потеряли возможность свободно расстаться с ним, либо на фирмы-производители компьютеров и образовательные институты, которые поддерживают широко распространенный взгляд А, который им полагается представлять как основной для их рынка. Даже если этот палец грозит поделом, тем не менее я не могу принять это как полное объяснение живучести взгляда А, и вынужден предположить, что взгляд А удовлетворяет более глубокие, физиологические потребности.

Откуда взялся взгляд В? Были люди, которые чувствовали, что появление больших и быстрых машин заменит тесные башмаки хотя бы на башмаки по размеру, и что несмотря на это эффективность выполнения программ останется

серьезной заботой программиста, заботой, которая станет даже более важной по мере роста машин и приложений, и что более сложные установки поставят более трудные проблемы. Также было замечено, что переключение с машинного кода на языки высокого уровня вовсе не гарантирует тех преимуществ, на которые возлагали столько надежд. В частности, программисты продолжали столь же охотно выдавать большие куски непонятного кода, и единственное различие было в том, что теперь они делали это в более грандиозных масштабах, и что высокоуровневые ошибки пришли на смену низкоуровневым. Они также поняли, что появление языков программирования высокого уровня не уменьшает потребности в тщательности: избыточность языков высокого уровня лишь уменьшает вредный эффект от некоторых видов небрежности. И тогда появился взгляд В. (Взгляд В не является реакцией на кризис программного обеспечения, который всплыл в 1968, так как он на много лет старше. Фактически взгляд В предсказал этот кризис, но даже это подтверждение не убило взгляд А).

После этой интерлюдии по поводу появления взгляда В вернемся к нашему вопросу: как и почему, лицом к лицу с признанными проблемами программного обеспечения, взгляд А (а именно – программирование в основном весьма просто) все еще здравствует. Вот ответ: из-за веры, причем не веры в лучших программистов, а веры в лучшие языки программирования или (диалоговые?) системы программирования, а также веры в лучшие технологии программного менеджмента.

Я придерживаюсь мнения, что программирование – один из наиболее сложных разделов прикладной математики, поскольку оно также является одним из наиболее сложных направлений инженерии, и наоборот. Когда я попытался разъяснить одному из моих коллег-математиков, почему я придерживаюсь этого мнения, он довольно бесцеремонно отказался выслушать мои доводы и вместо этого обвинил меня и моих единомышленников-компьютерщиков в том, что мы до сих пор не создали язык программирования, который сделал бы программирование настолько простым, насколько ему и подобает быть! Возможно, мне стоило бы спросить его, почему математики до сих пор не разработали нотацию, которая позволила бы любому, невзирая на отсутствие профессиональной подготовки, заниматься математикой?

Копнув чуть глубже, выясняется, что сторонники взгляда А не отрицают потенциальной сложности программ и их разработки, но верят, что жизнь программистов будет становиться все легче, поскольку все наиболее сложные части задачи будет брать на себя машина. Они указывают на появление языков программирования высокого уровня, которые уже сделали программирование гораздо легче, чем во времена старых машин, и опрометчиво экстраполируют, что в будущем программирование станет вовсе тривиальным. Но оправдана ли эта экстраполяция? Я много программировал, как в машинных кодах, так и на языках высокого уровня, и последние несомненно более удобны,

поскольку в этом случае многие решения, относящиеся к внутренним деталям программы, такие, как распределение памяти, не приходится принимать явно, поскольку ими занимается алгоритм распределения памяти компилятора. Переход к языкам высокого уровня освобождает нас от многих тривиальных забот. Это сделало программирование деятельностью с небольшим количеством нудной работы, с преобладанием изобретательности: именно та часть работы, которая занимала целые дни, исчезла! Вывод, который следует из появления языков программирования высокого уровня, о потребности в программистах большего интеллектуального калибра, полностью подтвердился моими наблюдениями в Западной Европе (где я мог следить за разработками последнее время): в конце 60-х многие крупные организации, использующие компьютеры, испытывали проблемы в подборе подходящей работы для программистов, нанятых 50-е годы, поскольку профессия переросла их интеллектуальные способности.

Однако ни эти наблюдения, ни указания на провал попытки COBOL'a выжить профессиональных программистов не произвели никакого впечатления на правоверных. Они объясняют, что традиционные языки программирования высокого уровня потерпели крах из-за их «процедурности», и что провал COBOL'a очевиден, поскольку ввиду недостаточной интерактивности он на самом деле не является простым английским, но вот через пять или десять лет дальнейший прогресс в области Искусственного Интеллекта (для посвященных – AI, т.е. Artificial Intellect) позволит нам построить «контекстно-зависимые», «основанные на знаниях», «автоматизированные системы для мышления и понимания», такие, что «пользователю достаточно будет только побеседовать с ними».

Должно быть, я неизлечимый скептик, но мне весьма трудно поверить, что подобным надеждам суждено сбыться. Имеются некоторые видимые подтверждения таких ожиданий – я цитирую отрывок из недавно полученного письма: «...в общем, передача более совершенному компьютеру того, что мы сегодня называем человеческими навыками, знанием и разумом». Я не намерен повторять здесь фрагменты горячих дискуссий, которые мы проводили по поводу значимости Искусственного Интеллекта, да здесь это и ни к чему.

Во-первых, оглядываясь назад, приходим к неизбежному заключению: смешивание надежды на будущее AI с завтрашней реальностью было бы безрассудством, и весьма безответственно быть неподготовленными на случай, если мечты по поводу AI останутся мечтами по крайней мере на протяжении нашей жизни. Или, говоря другими словами, глядя на серьезность сегодняшних проблем программирования, обычная осторожность заставляет нас не забывать о взгляде В.

Во-вторых, первоочередной задачей программиста, если он хочет, чтобы его творения заслуживали доверия, будет разрабатывать их настолько понятными, что он сможет нести за них ответственность, и, несмотря на ответ на вопрос, как много из его нынешней деятельности может быть переложено на машину, мы должны всегда помнить, что ни «понимание», ни «ответственность» не могут быть

классифицированы как деятельность: это скорее «состояние разума» и в принципе не может быть передано машине.

Я считаю неразумным, в особенности для ученого-компьютерщика, недооценивать влияние психологической школы, которая, считая человеческий разум слишком сложным и трудно поддающимся изучению, занялись вместо этого изучением крыс, и даже ограничивает это изучение – как я слышал недавнюю формулировку – «наиболее механической формой поведения – зачастую настолько механической, что даже крысам не дают проявить свои высшие возможности». Представляя свои грубые, механические модели в качестве допустимого приближения к человеческому разуму, они опасно затуманили различие между человеком и машиной, и мы наблюдаем два взаимно дополняющих друг друга феномена: антропоморфный взгляд на машины и механический взгляд на людей.

Эта неразбериха, вне всякого сомнения, – плод усилий верховных жрецов AI. Преобладание в основном антропоморфной терминологии в компьютерной науке – «память», «интерпретатор», «язык программирования», «рукопожатие», «диалог», и это лишь немногие примеры, – это предупреждение, которое не следует игнорировать. Я не знаю, как думать и разговаривать, обходясь без метафор; я знаю также, что каждая метафора несет в себе опасность скрытого подтекста. В данном случае антропоморфной терминологии в компьютерной науке мы давно уже достигли стадии, когда опасность путаницы перевешивает достоинства аналогии.

Кроме того, кажется, что механический подход к людям среди компьютерных ученых (и их руководителей) распространен шире, чем представляется мне нормальным. Ибо я подозреваю, что именно этот механический подход ограничивает деятельность программистов механическими действиями по написанию кода, и затем измеряет «производительность труда программиста» количеством произведенных им строк кода. (Когда весьма широко известный и очень уважаемый ученый-компьютерщик использовал недавно эту меру производительности труда программиста в своей лекции, от слушателей поступило предложение говорить не о «количестве произведенных строк кода», а о «количестве израсходованных строк кода», и что лектор, следовательно, занес их не в ту графу учета баланса расходов и доходов. Лектор ответил, что он вынужден использовать эту меру производительности, поскольку не располагает никакой альтернативной, которая позволяет вести точный учет!) Это не может больше рассматриваться как безобидное заблуждение, поскольку принятие этой бессмысленной «меры производительности» профессиональными программистами гарантированно стимулирует их к написанию рыхлого кода.

Влияние психологии было рассмотрено здесь, поскольку оно объясняет цепкость, с которой так много людей склонны тяготеть ко взгляду А.

В основном это не вина производителей компьютеров, которые желают вести дела так, будто они продают простейшую продукцию; и не вина руководителей программных проектов,



которые предпочитают рассматривать деятельность программистов как простой и предсказуемый процесс; и не вина учебных заведений, которые хотели бы подготовить студентов к достижению гарантированного успеха.

Это – следствие комфортной иллюзии, что Человек – это лишь сложный автомат, которая, подобно наркотику, приносит своим жертвам кажущееся освобождение от бремени ответственности. Признание программирования серьезным вызовом интеллекту вернуло бы полный вес этой ноши обратно на их плечи.

---

*prof. dr. Edsger W. Dijkstra*  
*Burroughs Research Fellow*

-----  
**От редакции OM/RE:** публикуемая фотография Prof. dr.  
*Edsger W. Dijkstra* (©2002 Hamilton Richards) взята с сайта  
<http://www.cs.utexas.edu/users/EWD/>