Петербург Александр Исаакович,

1950 г. р. Кандидат технических наук. Доцент кафедры «Проектирование аэрогидрокосмических систем» Московского авиационного института. Главный конструктор систем управления ОАО «ГНПП «Регион»». Автор концепции и внедрения бортовых интегрированных систем управления высокоскоростных высокоточных подводных объектов.

Тычинский Юрий Дмитриевич,

1971 г. р. Кандидат технических наук. Доцент кафедры «Системного анализа и управления» Московского авиационного института. Руководитель бригады математического и программного обеспечения ОАО «ГНПП «Регион»». Автор разработки и внедрения кроссплатформенных сетевых технологий программирования для бортовых систем автоматических самонаводящихся подводных аппаратов.

В пособии рассматривается множество примеров конкретных разработок и технологий, выявляются основные причины их успеха или неудачи. Выявленные причины обобщаются, делается попытка их систематизации, даются общие рекомендации по проектированию. Пособие может рассматриваться, как введение в проблематику проектирования сложных систем, знакомство с основными подходами и терминологией. Основной акцент делается на цифровых, в частности, на программных системах и технологиях, поскольку подобные системы являются в настоящее время наиболее сложными и масштабными.

Пособие предназначается для студентов и аспирантов информационных специальностей, а также для начинающих разработчиков.

ISBN 978-5-904336-29-5

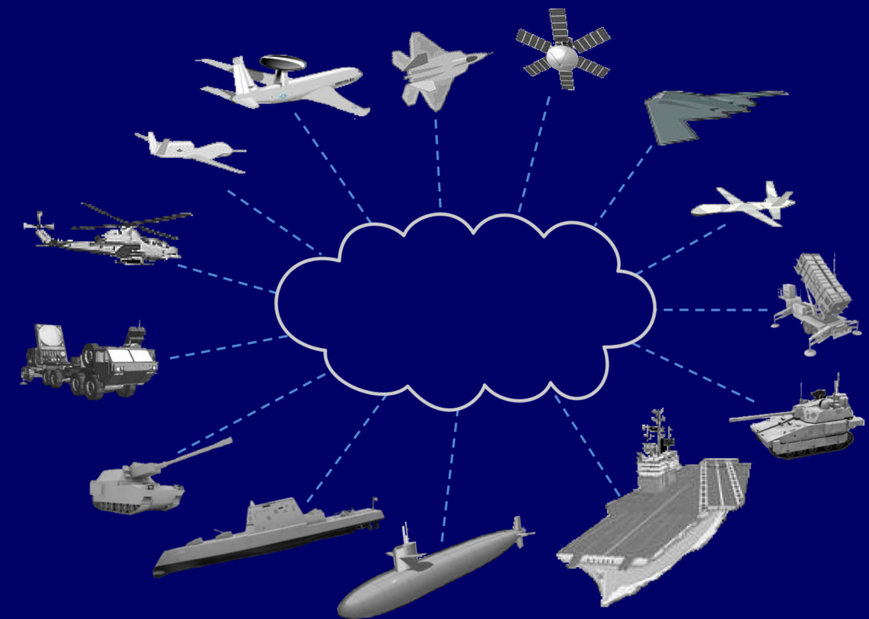
Международный издательский центр «Этносоциум»

Петербург А.И., Тычинский Ю.Д.

ТЕНДЕНЦИИ И ПРИНЦИПЫ ПРОЕКТИРОВАНИЯ СЛОЖНЫХ СИСТЕМ

Цифровые системы

Учебное пособие



Международный издательский центр
ЭТНОСОЦИУМ

А.И. Петербург, Ю.Д. Тычинский

ТЕНДЕНЦИИ И ПРИНЦИПЫ ПРОЕКТИРОВАНИЯ СЛОЖНЫХ СИСТЕМ

Цифровые системы

Учебное пособие

Москва
Этносоциум 2013

УДК 621
ББК 32

Рецензенты:

Заведующий кафедрой «Компьютерные системы и сети»
МГТУ им. Н.Э. Баумана, д. т. н., профессор *В.В Сюзев*

Эксперт, главный научный сотрудник ОАО «КБ Электроприбор»,
доктор технических наук, профессор *В.В. Сафронов*

Петербург А.И., Тычинский Ю.Д. Тенденции и принципы проектирования сложных систем. Цифровые системы. Учебное пособие. – М.: Международный издательский центр «Этносоциум», 2013. – 126 с.

ISBN 978-5-904336-29-5

В пособии рассматривается множество примеров конкретных разработок и технологий, выявляются основные причины их успеха или неудачи. Выявленные причины обобщаются, делается попытка их систематизации, даются общие рекомендации по проектированию. Пособие может рассматриваться, как введение в проблематику проектирования сложных систем, знакомство с основными подходами и терминологией. Основной акцент делается на цифровых, в частности, на программных системах и технологиях, поскольку подобные системы являются в настоящее время наиболее сложными и масштабными.

Пособие предназначается для студентов и аспирантов информационных специальностей, а также для начинающих разработчиков.

ISBN 978-5-904336-29-5 **УДК 621**
ББК 32

© Петербург А.И., Тычинский Ю.Д., 2013.
© Международный издательский центр «Этносоциум», 2013.

ОГЛАВЛЕНИЕ

Введение 5

1. Ключевые приемы разработки сложных систем . . . 9

1.1. Разделение труда, кооперация предприятий 9
1.2. Промежуточный продукт, расширение рынка, усложнение продукта 11
1.3. Декомпозиция сложных систем 13
1.4. Агрегирование 14
1.5. Проблемы стыковки подсистем. Инфраструктура системы 16
1.6. Системы с открытой архитектурой 19
1.7. Общесистемные ресурсы 25
1.8. Многоуровневые структуры 25
1.9. Основная проблема проектирования 27

2. Цифровые системы 30

2.1. Программные технологии 33
2.2. Низкоуровневое программирование. 34
2.3. Языки высокого уровня. 35
2.4. Объектно-ориентированные языки 37
2.5. Графические языки, CASE-технологии. 39
2.6. Идиомы. Паттерны проектирования 41
2.7. Роль архитектуры 43
2.8. Каркасы, фреймворки 46
2.9. Рефакторинг 49
2.10. Проблема увязки конкурирующих технологий . . 49
2.11. Увязка языков программирования. Декомпозиция программ на исполнимые модули . . . 51
2.12. Многозадачность. 53
2.13. Сетевые ОС 55
2.14. Удаленный вызов процедур 55

2.15. Программное обеспечение промежуточного слоя 57
2.16. Кроссплатформенность 60
2.17. Виртуальная машина 61
2.18. Технологии Java 64
2.19 .NET Framework 67
2.20. Протокол SOAP 69
2.21. Веб-службы 72
2.22. Сервис-ориентированные архитектуры 74
2.23. Серебряная пуля 76

3. Организация процесса разработки программного обеспечения. 79

3.1. Эволюция процессов разработки программного обеспечения. 79
3.2. Автоматизация разработки 85
3.3. Когда базар строит собор. 87
3.4. Какая организация процесса разработки лучше . . 94

4. Примеры организации систем 96

4.1. Эволюция информационных систем предприятия. 96
4.2. Интеграция систем 105
4.3. Монолитные системы. 106
4.4. Сервис-ориентированные системы реального времени 107
4.5. Интернет 116
4.6. Всемирная паутина 119
4.7. Причины успеха Интернета 121

Выводы 123

Литература 125

Введение

Относительно недавно человечество вступило в качественно новый этап технического развития. Этот этап характеризуется появлением и стремительным развитием систем глобального масштаба, охватывающих весь Земной шар. Подобные системы сложны настолько, что традиционные методы и представления не позволяют человеческому мозгу воспринимать их во всей полноте. Для проектирования систем такого масштаба нужны новые технологии. Эти технологии пока только формируются, бурно и хаотично, но в них уже можно проследить ключевые принципы, которые становятся общепринятой нормой проектирования. При внимательном рассмотрении оказывается, что большинство принципов проектирования известны достаточно давно, однако с усложнением систем их значимость возрастает многократно. Следование этим принципам, особенно в сочетании с современным инструментарием, приносит удивительно эффективные результаты не только в масштабных системах, но и в сравнительно небольших.

Цель данного пособия – продемонстрировать ключевые принципы и тенденции проектирования сложных систем на показательных примерах из реальной жизни; сформировать общее понимание и терминологию проблемной области. Пособие направлено на то, чтобы помочь студентам и начинающим разработчикам максимально перенять накопленный человечеством передовой опыт проектирования.

Принципы проектирования сложных систем, так или иначе, сводятся к основополагающей идее – максимальному использованию накопленного опыта. Имеется в виду не только профессиональный опыт конкретных разработчиков и их предшественников, но также и опыт проектировщиков из смежных отраслей. В конечном итоге – опыт всего человечества. Можно утверждать: прогресс невозможен без накопления опыта, иначе каждое поколение «изобретало бы велосипед» заново.

Каждый может согласиться с поговоркой: «Одна голова хорошо, а две – лучше». Еще больше потенциальных возможностей у коллектива из десятков и сотен разработчиков. А если разработчиков миллион? В таком случае возможен качественно новый результат, своего рода синергия усилий большого числа квалифицированных участников разработки.

Данный феномен давно осознан мировым сообществом и воспринимается, как сам собой разумеющийся факт. Проблемы возникают в другом. А именно, где найти квалифицированных разработчиков? Как эффективно наладить их совместную работу? Где взять средства для оплаты их труда?

В современном мире найдены эффективные инструменты решения перечисленных проблем. Эти инструменты постоянно совершенствуются. Примером могут быть такие транснациональные компании, как Microsoft, Google, Intel, IBM, Apple, Boeing, Airbus и пр. Количество сотрудников каждой такой компании составляет десятки, а то и сотни тысяч. Филиалы расположены в десятках стран мира. Изучение их деятельности позволяет бесплатно перенять драгоценный опыт коллективной разработки и управления большими проектами.

Грамотное использование современных технологий позволяет добиться небывалой производительности разработок. Существуют примеры, когда удачная организация коллективного труда позволяет небольшой группе разработчиков-координаторов конкурировать с мировыми гигантами. Одним из таких примеров является ОС Linux. Этот проект, начатый Линусом Торвальдсом в 1991 г, вскоре завоевал лидирующие позиции в таких сегментах рынка, как суперкомпьютеры, серверы, встраиваемые, мобильные системы и пр. В свое время он был одним из основных конкурентов ОС Windows корпорации Microsoft. Значимость коллективного труда для успеха ОС Linux выражается одним из т.н. законов Линуса: «при достаточном количестве глаз ошибки выплывают на поверхность» (англ. «given enough eyeballs, all bugs are shallow»).

ОС Linux является лишь одним из примеров успешного использования коллективного труда, а не панацеей на все случаи. Из каж-

дого проекта, даже неудачного, при внимательном анализе можно извлечь полезные выводы. По этой причине в пособии рассматривается множество показательных примеров, как удачных, так и не очень.

Навскидку, можно перечислить следующие основные концепции коллективной разработки и эффективного использования накопленного человечеством опыта.

1. Непосредственное привлечение к разработке квалифицированных специалистов. Успешные компании создают для специалистов условия и стимулы, чтобы те были заинтересованы работать именно у них, а не у конкурентов или в других отраслях. Часто оказывается выгодным привлечение фирм-контрагентов или отдельных специалистов по договорам для временных работ. Новой формой привлечения является т.н. распределенная разработка, когда разработчики проекта находятся в разных городах (странах) и решают вопросы через Интернет. Несмотря на очевидные накладные расходы данный вид разработки весьма эффективен.

Кроме непосредственного привлечения широко распространено косвенное привлечение опыта как собственных, так и сторонних разработчиков. Ниже следует целый ряд подобных примеров.

2. Использование опыта предшествующих поколений. Проекты редко рождаются на пустом месте. Практически всегда имеются прототипы, наработки, технологии, методики, документация и пр., знакомство с которыми позволяет перенять опыт. Часто предшествующие наработки используются непосредственно в виде отдельных модулей или подсистем в новом продукте или, наоборот, в продукте-предшественнике меняются (модифицируются) некоторые из модулей. В современном мире невозможно говорить о конкурентоспособности проекта, если он не обладает преемственностью. Для тех фирм, которые не умеют накапливать знания и каждый новый проект начинают «с нуля» используется показательный термин «организационное слабоумие» [6].

3. Опыт сторонних разработчиков заимствуется при закупке комплектующих для разрабатываемого изделия. Зачастую выгоднее купить готовый компонент у производителя, чем разрабаты-

вать его своими силами. При этом можно не только сэкономить ресурсы, но и выиграть в качестве (производитель специализируется на данных компонентах).

4. Приобретать можно не только готовые комплектующие, но и инструментарий для разработок. Сюда относятся как физические устройства, так и технологии, модели, программное обеспечение и пр. В инструментах воплощен опыт их разработчиков.

5. Автоматизация разработки (различного рода САПР) – еще один пример использования опыта сторонних разработчиков, которые вложили свои знания в автоматы. Особая привлекательность автоматов в снижении человеческого фактора, субъективных решений и ошибок.

6. Бесплатно можно перенимать опыт через службы технической поддержки, на интернет-форумах разработчиков, конференциях и пр. Очень полезно знакомство с т. н. открытыми проектами, например, проектами с открытым исходным кодом программ. Существует множество сравнительно недорогих и информативных обучающих курсов. Широко распространена практика привлечения консалтинговых компаний и т.д.

Приведенный краткий обзор демонстрирует широкие возможности привлечения к разработке накопленных человечеством знаний и опыта. Однако само по себе привлечение большого числа разработчиков и технологий далеко не всегда приводит к желаемому результату. Пропорционально их количеству возникает путаница, нестыковки, различного рода конфликты, которые могут застопорить разработку. Возникает проблема рациональной организации взаимодействия разработчиков.

Оказывается, что эффективность коллективного труда в значительной мере определяется не только взаимодействием между разработчиками, но и внутренней организацией проектируемой технической системы, ее архитектурой. Чем больше и сложнее система, тем большее значение имеет ее архитектура. Этим и другим подобным вопросам в пособии уделяется значительное внимание.

1. КЛЮЧЕВЫЕ ПРИЕМЫ РАЗРАБОТКИ СЛОЖНЫХ СИСТЕМ

1.1. Разделение труда, кооперация предприятий

Сложный продукт имеет, как правило, много этапов обработки. Во времена ремесленнического производства все эти этапы производил ремесленник, возможно прибегая к помощи подмастерья. Получавшийся продукт можно сравнивать с произведением искусства, воплощавшим в себя личные навыки и мастерство изготовителя. Подобные близкие к искусству навыки трудно передать другим людям, поскольку, во-первых, далеко не многие к ним способны, во-вторых, процесс обучения длился достаточно долго. За всю жизнь ремесленник мог обучить сравнительно небольшое количество учеников. Эффективность передачи опыта была незначительной.

Чем сложнее продукт, чем больше у него этапов обработки, чем они разнообразнее, тем сложнее одному человеку освоить его изготовление. Сами по себе этапы обработки могут быть несложными, но их количество и разнообразие делают затруднительным изготовление продукта одним человеком.

Кроме того, в силу индивидуальных особенностей человек более предрасположен к одним видам работ, а меньше к другим. Индивидуальные затруднения с некоторым из видов обработки не позволяют изготовить продукт целиком. Очевидное решение проблемы – привлечь к работе еще одного мастера, который более умело выполняет проблемный вид обработки. Таким образом, в изготовлении продукта будут участвовать два мастера, каждый будет специализироваться на том виде работ, к которому он лучше предрасположен. В результате получается продукт, который каждому из мастеров в отдельности было бы затруднительно изготовить или вообще не под силу. В данном примере можно говорить о синергии совместного труда мастеров.

Развитие указанной идеи привело к появлению мануфактур – производств, в которых существенным образом использовалось *разделение труда* рабочих. Эффективность такой организации труда была столь велика, что в короткое время мануфактуры практически полностью вытеснили с рынка своих конкурентов – ремесленников.

Разделение труда и кооперация позволили изготавливать более сложные изделия, которые не под силу изготовить одному человеку. Требования к квалификации большинства рабочих снизились по сравнению с мастером-одиночкой, стоимость труда уменьшилась, стабильность производства повысилась (меньшая зависимость от человеческого фактора).

Сниженные требований к основной массе рабочих позволили расширять производства, увеличивать объем продукции, получать за счет этого большую прибыль. Возросшая текучка кадров заставила искать способы передачи навыков и опыта в массовом порядке. Существенную роль стала играть технологическая, обучающая и пр. документация, появились специализированные обучающие подразделения и даже независимые учебные заведения и т.д.

Хотя требования к рабочим в общем снизились, значимость высококвалифицированных и талантливых людей возросла. Собственники и руководители предприятий поняли, что талантливых людей не так много, их выгоднее использовать для творческих, наиболее проблемных задач, а не для рутины. Для мастеров создавались условия, чтобы те изобретали технологии, генерировали рационализаторские идеи, разрабатывали шаблоны и пр. Для рутинной работы привлекались менее квалифицированные рабочие.

Рассмотренные примеры разделения труда представляют собой кооперацию разработчиков (изготовителей) в пределах предприятия. Если разделение труда эффективно в пределах предприятия, то почему оно не должно принести выгоду за его пределами? И действительно, в силу исторических, географических и пр. условий те или иные производства оказываются более эффективными в каких-то технологиях. По этой причине кооперация нескольких узкоспециализированных предприятий может оказаться выгод-

нее, чем единственное громоздкое производство, производящее все этапы обработки продукта.

Примером кооперации могут быть предприятия металлургии и машиностроения. Горно-обогатительные комбинаты добывают и обрабатывают железную руду, металлургические – выплавляют чугун, из него разные виды стали, которую формуют в виде профилей, труб, листов и пр. Металлическое сырье используется соответствующими предприятиями для изготовления деталей, например, крепежа: болтов, гаек, шайб и т.д. Детали и заготовки используются для производства и сборки изделий для конечного пользователя: машин, станков, мостов и пр.

1.2. Промежуточный продукт, расширение рынка, усложнение продукта

В приведенном примере цепочки изготовления продуктов машиностроения появляется новый вид продукта – *полуфабрикат.* Продукт, предназначенный для последующей обработки, а не для конечного пользователя. Полуфабрикат, как правило, имеет бо́льшую универсальность, чем конечный продукт: болты могут применяться где угодно, а для моста сложно придумать другое применение, кроме основного.

Бо́льшая универсальность привлекательна более широким и стабильным рынком сбыта. С другой стороны, широкий рынок сбыта привлекает производителей и порождает конкуренцию. Поскольку полуфабрикаты менее сложны, чем конечные продукты (конечный продукт включает в себя и полуфабрикаты), то проще наладить их производство, от чего конкуренция усиливается.

Расширением рынка объясняется также тенденция усложнения производимых продуктов. Очевидно, что более сложные продукты могут предоставлять для потребителя более широкий спектр функций, бо́льшую *функциональность.* Новые функции увеличивают потребительский спрос и открывают новые ниши рынка.

Сказанное в полной мере относится не только к конечным

продуктам, но и к полуфабрикатам. Ярким примером являются процессоры. Их универсальность, с одной стороны, обеспечивает широчайший рынок сбыта. С другой стороны, разработка и производство процессоров настолько сложны, что реализовать их могут только единичные фирмы и единичные страны. Оба эти обстоятельства обеспечивают высокую прибыль фирмам-производителям.

Чтобы высокотехнологичный полуфабрикат находил широкое применение, нужно упростить его использование. Другими словами, для использования полуфабрикатов желательны несложные доступные технологии.

Вышеупомянутое направление прогресса привело к следующему положению дел. Крупные фирмы (корпорации), стремясь завоевать новые ниши рынка, затрачивают огромные средства на разработку высокотехнологичных полуфабрикатов, предлагая потребителю новые и новые функции/возможности. Для расширения рынков сбыта эти полуфабрикаты стремятся делать универсальными, а их использование – простым. Массовость производства позволяет снизить стоимость высокотехнологичных продуктов, что в свою очередь способствует расширению рынка.

Спрос на все более и более сложные товары естественным образом заставляет разработчиков усложнять производимые продукты. Можно утверждать, что в любое время были продукты, разработанные на пределе возможностей существующих на тот момент технологий. По мере совершенствования технологий усложнялись и разрабатываемые продукты.

Усложнение продукта приводит к росту затрат на его разработку и производство. Причем рост этот существенно нелинеен. Давно замечено, что с некоторого момента увеличение затрат на разработку не приводит к сколь нибудь значимому улучшению потребительских качеств продукта, в частности, к росту его функциональности. Такое свойство иногда называют порогом сложности. Понятие порога сложности относительно. Оно привязано к определенному уровню технологий. То, что было пределом возможного для одних технологий, может не представлять особой проблемы

для более совершенных. Появление новых технологий не раз приводило к скачкообразному росту потребительских свойств продуктов. Впрочем, достаточно быстро разработки приближались к новому порогу сложности, соответствующему новым технологиям.

1.3. Декомпозиция сложных систем

Чем сложнее становился продукт, тем большую роль в его разработке играла *декомпозиция*. Согласно данному методу сложная задача (разрабатываемый продукт) разбивается на ряд более простых подзадач (подсистем продукта). Для каждой задачи выдвигаются локальные (более простые, чем в исходной задаче) требования, которые по-отдельности решаются проще, чем вся задача в целом. Если декомпозиция произведена правильно, то решение локальных задач дает также и решение исходной.

Ограниченные человеческие возможности не позволяют воспринимать сложную задачу целиком со всеми ее подробностями. Прием декомпозиции позволяет рассматривать задачу на разных *уровнях абстракции*. На нижнем уровне решаются частные подзадачи без учета (без детального учета) вышестоящих проблем и проблем в соседних подзадачах. На верхнем уровне решаются вопросы увязки подзадач без учета многих деталей их реализации.

При использовании метода декомпозиции возникают и дополнительные накладные расходы на увязку/стыковку подзадач, но, опять же, при правильной декомпозиции эти расходы должны быть ниже, чем решение исходной задачи. Более того, без декомпозиции исходная задача может и вовсе не иметь решения.

Применение метода декомпозиции позволяет не только упростить решение сложной задачи. Попутно обеспечивается и разделение труда разработчиков, распараллеливание работ, специализация групп разработчиков на подсистемах определенного типа.

В отличие от первых мануфактур, о которых говорилось ранее, разделение труда в данном случае не обязательно связано с используемыми технологиями. Разные подсистемы могут разрабатываться с помощью одинаковых технологий, а специализируются они на отдельных функциях системы.

Если на мануфактурах разным стадиям технологической обработки подвергался один и тот же продукт, то декомпозированная система собирается в единое целое из нескольких подсистем, которые в свою очередь могли проходить несколько стадий обработки. Использование приема декомпозиции при разработке продукта положило основу для появления *составных систем*.

Подсистемы, по сути, являются полуфабрикатами, поэтому обладают всеми преимуществами последних. По аналогии с полуфабрикатами, рассматривавшимися ранее, они более универсальны и, следовательно, имеют более широкий рынок сбыта. Например, один и тот же двигатель может использоваться на разных марках автомобилей и не только на автомобилях.

Широкий рынок сбыта способствует тому, что группы разработчиков, специализирующиеся на определенных подсистемах, со временем выделяются в отдельные фирмы. Если изначально подсистемы изделия разрабатывались в рамках одного предприятия и были уникальными, то со временем подсистемы находят себе применение в других изделиях и даже в других отраслях, становятся более универсальными. Таким образом возникли фирмы, специализирующиеся на навигационных системах, двигательных установках и пр.

1.4. Агрегирование

Со временем рынок все более и более насыщается универсальными полуфабрикатами, которые скрывают в себе значительный опыт и сложные технологии. В то же время их использование несложно и доступно большинству небольших фирм и даже индивидуальным разработчикам. Развиваются не только полуфабрикаты, но и инфраструктура торговли, законодательства стран и пр. Полуфабрикаты становятся доступными практически в любой точке планеты.

Указанная тенденция приводит к ситуации, когда разработчик системы не изобретает ее составляющие, а подбирает из имеющихся на рынке, увязывает (комплексирует) готовые решения. По сути, он *агрегирует* систему из готовых компонент.

При проектировании систем целесообразно применять высокотехнологичные полуфабрикаты (например, процессоры), которые с одной стороны, концентрируют в себе колоссальный опыт человечества, с другой стороны, доступны, сравнительно недороги и просты в использовании. Универсальность полуфабрикатов позволяет их применение даже в весьма специфических областях.

Непосредственное использование готовых полуфабрикатов не единственный путь использования накопленных знаний. Собственные разработки также целесообразно реализовать в виде полуфабрикатов, предназначенных для многократного использования в разных проектах. Особенный эффект достигается, если в виде универсального полуфабриката удастся реализовать самые сложные элементы разработки так, чтобы их повторное использование было простым. Другими словами, однажды потратив средства на разработку сложных элементов (полуфабрикатов), можно будет многократно повторно их использовать.

В силу сказанного, важнейшей задачей при разработке системы является выявление в ней высокотехнологичных и сложных элементов (узлов, подсистем), которые станут кандидатами на сложные, но универсальные полуфабрикаты, подобные вышеописанным. Если нужные полуфабрикаты не окажутся доступными на рынке, то их разработку можно заказать у фирм, специализирующихся на соответствующих технологиях или разработать самостоятельно. В последнем случае возможно временное привлечение для разработки высококлассных (и дорого оплачиваемых) специалистов. Разовое привлечение позволит сэкономить средства с их постоянной оплаты.

Выявление в разрабатываемой системе универсальных и высокотехнологичных узлов происходит, как правило (и даже в первую очередь), с оглядкой на имеющиеся на рынке полуфабрикаты. Со временем все более и более усиливается тенденция, когда не конкретная разрабатываемая система диктует то, какими будут полуфабрикаты, а наоборот, имеющиеся на рынке полуфабрикаты определяют то, какой будет система (здесь имеется в виду не функциональность системы, а ее реализация, внутренняя организация).

1.5. Проблемы стыковки подсистем. Инфраструктура системы

Чем больше усиливается тенденция агрегирования систем из существующих на рынке полуфабрикатов, тем острее проявляется проблема их взаимоувязки, особенно, если система собирается из полуфабрикатов различного назначения, изготовленных различными изготовителями. Исторически сложившееся разнообразие способов взаимодействия с устройствами оставляет мало шансов на то, что они будут совместимы между собой, особенно, если они изначально не предназначались для совместной работы. Проблема усиливается с ростом числа используемых полуфабрикатов.

Решений проблемы может быть несколько. Например, головная фирма разрабатывает облик системы, подбирает готовые подсистемы. Как правило, подсистем, взаимодействующих друг с другом желаемым образом, оказывается немного. В таких случаях ищутся контрагенты, специализирующиеся на нужных подсистемах. Им дается техническое задание на доработку существующих подсистем таким образом, чтобы они стыковались в изделии с соседними компонентами нужным образом. Некоторые подсистемы разрабатываются заново.

Очевидно, что доработать существующую систему дороже, чем купить готовую, еще больше стоимость новой разработки. По этой причине бывает выгодно разработать дополнительные устройства-*адаптеры,* которые увязывают готовые компоненты в единую систему или же разработать платформу, которая объединит разнородные компоненты.

На первый взгляд такая задача кажется простой, поскольку ее функции: адаптация, коммутация и пр. не столь сложны. Однако здесь кроется множество неявных проблем и, как показывает практика, проблемы увязки и стыковки различных компонент оказываются наиболее сложными и трудозатратными. Соответственно платформа, объединяющая подсистемы часто оказывается самым дорогостоящим компонентом системы.

Одна из причин проблемы стыка устройств – формальная. До-

кументация на приобретаемый компонент очень часто не раскрывает всех технических тонкостей его интерфейса. Аналогично, техническое задание контрагентам на разработку (доработку) устройства не всегда отражает все технические детали, может содержать двусмысленные трактовки и банальные ошибки. Последнее обстоятельство во многом объективно. Невозможно на ранней стадии разработки (на стадии формирования технического задания) предугадать все нюансы. Ошибки и нюансы выясняются позже – на стадии стыковки и совместной отработки устройств в составе системы. Однако контрагенты неохотно соглашаются дорабатывать свои подсистемы для исправления ошибок технического задания. Для этого требуются дополнительное финансирование и сроки.

Другая проблема совместной отработки устройств системы заключается в сложности выявления причин нестыковки. Не редки ситуации, когда устройства по-отдельности работают нормально, а совместно не работают или же работают нештатно. Трудно понять в каком из устройств кроется причина проблемы. Причина может крыться не в самих устройствах, а в коммуникациях, адаптерах. Ярким примером может быть электромагнитная несовместимость устройств.

В подобных ситуациях приходится анализировать не отдельно взятые подсистемы, а всю систему целиком, нарушая идею декомпозиции. Более того, произведенная декомпозиция системы отрицательно сказывается на решении проблем стыковки, поскольку она способствует тому, что разные части системы разрабатываются разными группами специалистов, с разными традициями, по разным технологиям.

Для анализа совместной работы устройств нередко приходится привлекать их разработчиков, что организовать весьма не просто, а иногда и не возможно (если, например, компонент зарубежный).

Разумеется, проблема стыка подсистем не нова. С ней сталкиваются практически все фирмы-разработчики сложных систем. Раз это так, то существуют и методы решения рассматриваемой проблемы.

Одним из таких методов является *унификация интерфейсов* взаимосвязи между устройствами. По всему миру тратятся значительные средства на разработку стандартов взаимосвязей. Это и механические связи, и электрические, и электромеханические, электромагнитные, оптические, цифровые и пр.

Разработка *стандарта* весьма трудоемкий процесс, который предусматривает не только предварительный анализ большого числа разработок, но и последующую отработку стандарта на многих проектах. Кроме того, формальные юридические процедуры оформления стандарта обязывают его рецензирование большим числом ведущих фирм, специализирующихся на проблематике конкретного стандарта.

Таким образом, стандарт накапливает в себе значительный опыт многих разработчиков, раскрывает практически все нюансы, не содержит двусмысленностей и ошибок.

В силу указанных причин разработчики сложных систем стараются применять стандарты внутри своей системы, что порождает спрос на компоненты со стандартными интерфейсами. Соответственно разработчики полуфабрикатов, стремясь завоевать рынок, стараются реализовать свой продукт в соответствии со стандартами.

Использование стандартов на уровне системы существенно упрощает проблемы увязки подсистем. При формировании технического задания контрагентам вместо объемного текста просто дается ссылка на соответствующий стандарт.

К сожалению, исторически складывается так, что многие стандарты дублируются. Там где стандарт мог бы быть единым, существует несколько вариантов в чем-то превосходящих, а в чем-то уступающих друг другу. Но даже в такой ситуации целесообразнее использовать один из существующих стандартов, чем изобретать свой. Тем более, что на рынке можно приобрести готовые адаптеры, мосты и пр. устройства согласования различных интерфейсов.

Стандарты и правила взаимодействия компонентов в рамках системы, а также их аппаратная реализация представляют собой некоторый *каркас* или *инфраструктуру,* которая наращивается конкретными устройствами.

Правильный выбор инфраструктуры имеет чрезвычайно важное значение для будущих модификаций разрабатываемой системы. Если поменять одно или несколько устройств можно с незначительными затратами, то изменение инфраструктуры влечет за собой практически полную переделку всей системы, что в ряде случаев не представляется возможным.

Показательным является пример использования номинала напряжения 110В в бытовых электросетях США. В свое время данный номинал был принят из гуманных, на первый взгляд, соображений безопасности. Однако безопасность по сравнению с номиналом 220В повышается символически, зато потери в электросетях возрастают в четыре раза. Кроме того, производителям электроприборов, стремящимся выйти на внешний рынок, приходится ориентироваться сразу на два стандарта: европейский и американский. Неудачный номинал напряжения в электросетях превратился в большую проблему, которая не может быть решена из-за наличия огромного числа абонентов.

Яркими примерами систем, в которых ключевую роль играет инфраструктура, являются системы массового обслуживания и распределенные системы: электросети, системы связи, телекоммуникаций и пр. Можно утверждать, что изначально у таких систем проектировались не столько оконечные устройства, сколько сама инфраструктура. Тщательное проектирование инфраструктуры привело к тому, что она пережила несколько поколений абонентских устройств.

1.6. Системы с открытой архитектурой

Со временем популярность распределенных систем все более и более возрастает. Появляются новые системы, отличающиеся областью применения, масштабом, технологиями. Это и сравнительно простые системы сигнализации, видеонаблюдения, пожаротушения и более сложные системы автоматизации производств, торговли, банковские системы, системы связи, навигации, системы государственного и военного назначения и т.д. Самым внушительным примером открытых систем можно назвать Интернет.

Если первые распределенные системы обслуживания, например системы связи, имели дело с однотипными абонентами, то сейчас можно найти множество систем, чьи абоненты существенно разнотипны. В качестве примера можно взять систему автоматизации сборочного цеха. Цех оборудован разнообразными автоматами: станками и механизмами, приборами контроля, пультами оператора и пр. Система автоматизации должна предусматривать взаимодействие всего разнообразия устройств. Набор устройств на каждом из предприятий специфичен. Более того, отдельные устройства и система в целом должны адаптироваться под часто меняющийся ассортимент продукции. В случае нештатных ситуаций, например, при локальных неполадках оборудования, система должна предоставлять возможность оперативного переконфигурирования. Также должна предусматриваться возможность наращивания системы, в том числе и таким оборудованием, которое появится на рынке позднее самой системы.

Очевидно, что разработка подобной масштабной системы «с нуля» требует огромных затрат. В рыночных условиях трудно найти такое производство, которое бы на них согласилось. Другое дело, если подобные системы уже внедрены на других предприятиях и проявили себя положительно. В таком случае снижаются не только риски, но и стоимость, поскольку адаптировать подобную систему дешевле, чем разработать ее заново. Другими словами шанс завоевать рынок имеется у тех фирм, которые обладают значительным опытом и наработками, умеют эффективно накапливают знания.

В силу очевидных причин данную нишу рынка завоевали системы, обладающие следующими свойствами: *распределенностью* и *регулярностью* структуры, возможностью *наращивания, конфигурирования, адаптации.* Эти свойства закладываются в систему на этапе проектирования, на их разработку затрачивается немало ресурсов, привлекаются лучшие специалисты. Все это позволяет разработать стандартные правила и типовые устройства, с помощью которых систему могут конфигурировать инженеры среднего уровня с минимальными затратами. Причем делается это на этапе

эксплуатации, часто без отключений системы. Можно сказать, что распределенные системы обладают высокой степенью *гибкости*. Значительные ресурсы, затраченные единовременно на разработку инфраструктуры, многократно окупаются впоследствии при повторном использовании решений.

Часто говорят, что системы, обладающие перечисленными свойствами, имеют открытую архитектуру. Ее упрощенная схема представлена на рис. 1.

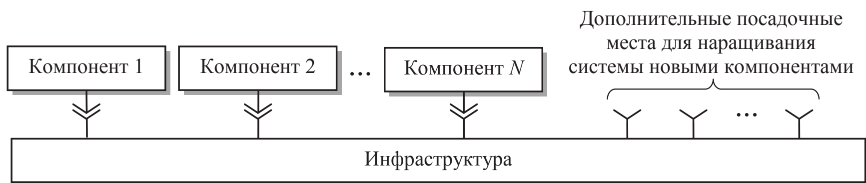


Рис. 1. Условная схема системы с открытой архитектурой

Основой системы является каркас или инфраструктура, которая предоставляет посадочные места для компонент. Компоненты взаимодействуют друг с другом только посредством инфраструктуры. Посадочные места предусматривают простую замену компонент на этапе эксплуатации (возможно при временном отключении системы). В каркасе предусмотрены дополнительные посадочные места для будущих модернизаций и расширений системы. Дополнительные посадочные места могут и отсутствовать, но их добавление не должно быть проблематичным.

Если система разрабатывается «с нуля», когда мало наработок и нет прототипов, то производится декомпозиция системы на подсистемы, которые в будущем превратятся в компоненты или наборы взаимодействующих компонент. Если же имеется достаточное число наработок и готовых компонент, то система агрегируется из них. Таким образом, изначально предполагается использование рассмотренных выше важнейших для сложных систем методов разработки: декомпозиции и агрегирования со всеми сопутствующими им преимуществами.

Рассмотрим основные свойства и приемы разработки систем с открытой архитектурой.

1. Системы с открытой архитектурой естественным образом упорядочивают труд разработчиков. Разработчики (группы разработчиков, фирмы) специализируются на конкретных компонентах, чем обеспечивается разделение труда. Кроме того, разработка системы распараллеливается: многие компоненты могут разрабатываться независимо и одновременно (ускоряется процесс разработки).

2. Системы с открытой архитектурой обладают высокой степенью гибкости. В таких системах изначально предусматриваются возможности конфигурирования и наращивания компонент. Система легко модернизируется, путем замены устаревших компонент. Модернизация может производиться в процессе эксплуатации постепенно – по одной или нескольким компонентам. Возможность наращивания количества компонент системы во время эксплуатации позволяет постепенно увеличивать масштаб системы. Часто система запускается в минимальной конфигурации, после чего наполняется новыми компонентами. При этом может увеличиваться не только масштаб системы, но и ее функциональность – новые компоненты могут привносить в систему новые функции даже такие, которые не предусматривались при ее создании. Первые разработчики Интернета вряд ли могли себе представить то, какие функции на него будут возложены со временем.

3. Особую роль в системах с открытой архитектурой играет инфраструктура или каркас. Именно она в большой мере обеспечивает гибкость системы. Для этого инфраструктура должна обладать достаточными резервами пропускной способности и не накладывать на систему лишних ограничений. Особой гибкостью обладают те системы, в которых обеспечивается связь между любыми ее компонентами. Наиболее эффективны инфраструктуры, основанные на общепринятых стандартах взаимосвязей между компонентами. Во-первых, стандарты впитывают в себя значительный опыт разработчиков подобных систем, более того, стандартную инфраструктуру можно приобрести либо целиком, либо частично. Во-

вторых, под стандартную инфраструктуру можно найти готовые компоненты, а у компонент, разрабатываемых заново, появляется шанс быть использованными не только в данной системе, но и в других системах, основанных на таком же интерфейсе (расширение рынка сбыта – дополнительный стимул).

4. Большинство разработчиков с целью экономии ресурсов стараются ограничить количество используемых в инфраструктуре стандартов. Чаще всего используется единый стандарт. Тем не менее, предусматриваются меры для адаптации к другим стандартам в случае необходимости, например, для увязки с другими системами или нестандартными компонентами. В подобных случаях часто разрабатывают специальные устройства: адаптеры, мосты и пр. Значительное количество таких устройств можно найти на рынке в готовом виде.

5. Унификация и стандартизация приносит эффект не только в инфраструктуре, но и в отдельно взятых компонентах системы. Унифицированный компонент имеет шанс быть использованным повторно в аналогичных системах, в том числе и в разработках сторонних фирм. Компонент является полуфабрикатом и, как об этом уже говорилось, часто имеет большую универсальность и более широкую область применения, чем система целиком, а значит и рынок сбыта. Даже если разрабатываемый компонент и не выйдет на рынок в виде отдельного продукта-полуфабриката, стремление сделать его таковым можно считать хорошим тоном проектирования особенно, если затраты на его унификацию незначительны. Во-первых, такой полуфабрикат может быть использован в других разработках фирмы, во-вторых, «выделяемость» компонент в отдельные продукты говорит о хорошей декомпозиции системы.

6. Гибкость и модернизационный потенциал системы обеспечивается не только возможностями наращивания и конфигурирования ее структуры. Важную роль играют резервы производительности, памяти, пропускной способности и пр. ресурсов. Существенную гибкость может обеспечить прием параметризации, когда предусматриваются простые способы изменения свойств продукта, путем задания нужных значений ряда параме-

тров. Перечисленные приемы эффективны не только для инфраструктуры, но и для компонент системы.

7. Важную роль в смысле гибкости системы играют не только стандартные правила взаимодействия компонент через инфраструктуру, но и фактическое наполнение передаваемой/принимаемой компонентами информации, другими словами *интерфейс* компонент (на рис. 1 интерфейс условно показан в виде разъемов, посадочных мест). В ряде случаев удается так обобщить интерфейс компонента, что значительные переделки внутреннего содержания последнего никак не отражаются на его интерфейсе, а значит, и на других компонентах, на всей системе в целом. Компонент воспринимается системой, как черный ящик. Он виден только через свой интерфейс, а его внутреннее исполнение скрыто (т.н. прием *инкапсуляции*). Если интерфейс не поменялся, то неважно поменялось ли содержимое компонента. Подобные интерфейсы компонент принято называть *стабильными*.

9. Взаимодействие между компонентами следует всячески ослаблять. Чем меньше предположений и увязок будет между компонентами, тем легче их удалять/заменять/добавлять. Другими словами легче конфигурировать систему.

10. Открытая архитектура систем целенаправленно не оговаривает внутреннее исполнение компонент, но каждая организация стремиться унифицировать технологии и технические решения, применяемые в разработке разных компонент.

11. Стремление унифицировать все и вся, предусмотреть все возможные варианты использования, обеспечить систему большими резервами и пр. часто приносит противоположный результат. Система переутяжеляется: увеличиваются габариты, энергопотребление, сроки и стоимость, как разработки, так и конечного продукта. Часто оказывается, что заложенные в системе дополнительные возможности так и не оказываются востребованными, а непредвиденные проблемы все равно возникают в неожиданных местах, поэтому при проектировании всегда следует сохранять разумную меру.

Системы с открытой архитектурой впитывают практически все

приемы накопления знаний и повторного использования удачных решений. Их анализ позволяет перенять ценнейший опыт проектирования и эксплуатации сложных систем.

1.7. Общесистемные ресурсы

На схеме системы с открытой архитектурой (рис. 1) можно видеть, что инфраструктура отделена от других компонент (подсистем). В тоже время инфраструктура – это одна из подсистем изделия. Отличие заключается в ее предназначении: инфраструктура предоставляет сервис для других подсистем. Подобные подсистемы иногда называют *общесистемным ресурсом.* Многие сервисы не представляют интереса для конечного пользователя, а выполняют *служебные,* еще говорят, *системные* функции.

Одно из основных назначений сервисов – скрыть (инкапсулировать) подробности и технологии управления ресурсами и предоставить компонентам-пользователям упрощенный доступ к этим ресурсам. В частности, инфраструктура скрывает технологии передачи информации между компонентами.

Скрытие технологий – непосредственное проявление разделения труда, а использование разными компонентами системы единого сервиса – повторные использования технического решения. По указанным причинам общесистемные ресурсы получили широкое распространение. Примерами общесистемных ресурсов могут быть: телекоммуникации, системы энергообеспечения, охлаждения, корпусная часть и пр.

Поскольку на общесистемный ресурс опираются остальные компоненты, его принято изображать в виде слоя или уровня под компонентами, как показано на рис. 1.

1.8. Многоуровневые структуры

Общесистемный ресурс зачастую скрывает не одну, а несколько технологий. С целью разделения труда эти технологии часто удается реализовать в виде подуровней. Ярким примером такой структуры является эталонная модель взаимодействия открытых

систем OSI (Open Systems Interconnection, стандарт ISO-7498)[1]. Модель часто в том или ином (например, усеченном) виде используется при реализации инфраструктуры открытых систем (рис. 1).

Функции OSI разделены на семь подуровней таким образом, что все вышележащие уровни пользуются услугами нижележащих через стандартизованные интерфейсы (см. рис. 2).

Прикладной уровень (application layer)
Уровень представления данных (presentation layer)
Сеансовый уровень (session layer)
Транспортный уровень (transport layer)
Сетевой уровень (network layer)
Канальный уровень (data link layer)
Физический уровень (physical layer)

Рис. 2. Подуровни модели OSI

Самый нижний (физический) уровень определяет требования к механическим свойствам кабелей и разъёмов, электрические характеристики сигналов, топологию сети, способ кодирования и передачи отдельных битов данных через физическую среду. Второй снизу (канальный) уровень формирует кадры из последовательности битов, поступающих от физического уровня, обеспечивает организацию, поддержку и разрыв связи между физическими узлами сети. Сетевой уровень реализует функции маршрутизации пакетов, обработки ошибок, мультиплексирования и т. д. Приведенные примеры демонстрируют существенное разделение технологий между уровнями модели.

Многоуровневая структура позволяет модифицировать любой

[1] В полном виде стандарт OSI не нашел широкого применения. Наибольшее распространение получил стек протоколов TCP/IP, который стал стандартом de facto. В данном контексте важнее то, что TCP/IP имеет подобную многослойную структуру.

из уровней, не затрагивая остальные, что обеспечивает гибкость системы. Даже, если поменяется интерфейс одного из уровней, то изменения затронут только вышележащий слой, а не всю систему.

1.9. Основная проблема проектирования

Каждый уровень многоуровневой системы (см., например, модель OSI на рис. 2) скрывает подробности и технологии, предоставляя вышележащему уровню упрощенный доступ к ресурсам. Скрытие подробностей и упрощение доступа является по своей сути *абстрагированием.* С каждым новым уровнем скрывается все больше и больше деталей реализации. Таким образом получается, что с повышением уровня возрастает и уровень абстракции проблемы.

С уровнями абстракции тесно связан т. н. *провал (зазор) абстракции,* который называют основной проблемой проектирования [6].

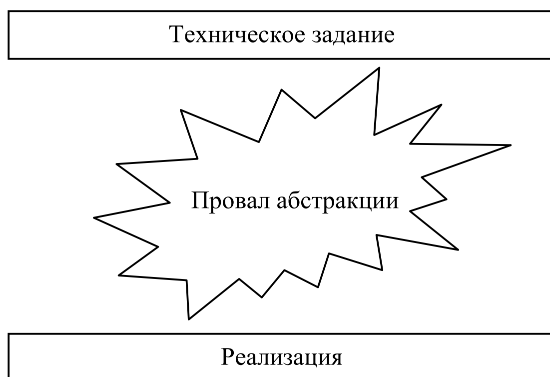


Рис. 3. Иллюстрация основной проблемы проектирования

У заказчика имеется представление о том, что должна представлять собой система, какими потребительскими характеристиками должна обладать, какой функциональностью и пр. Свои требования заказчик обычно формулирует в виде *технического*

задания. Но заказчик плохо представляет себе то, каким образом требования к системе превратятся в ее реализацию, т.е. в конкретную конструкцию, аппаратуру, кабели, разъемы, сигналы, биты, байты и пр.

С другой стороны, разработчик хорошо разбирается в аппаратуре, технологиях, инструментариях разработки, но часто хуже представляет себе область применения конечного продукта.

В процессе проектирования преодолевается разрыв между техническим заданием на продукт и его реализацией. Эффективность технологий разработки можно оценивать по скорости и затратам на преодоление этого разрыва. Преодоление разрыва снизу вверх называют абстракцией, а сверху вниз – уточнением задания (рис. 4).

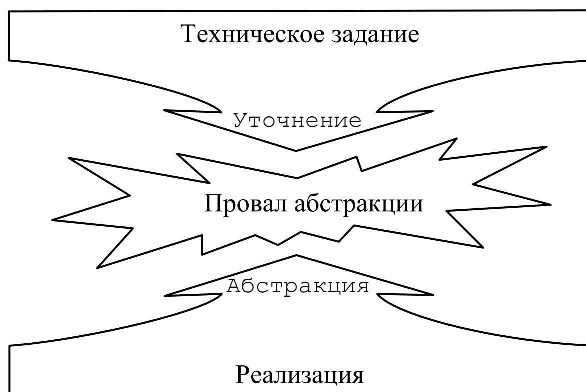


Рис. 4. Преодоление основной проблемы проектирования

Разработчик, решая некоторую задачу, интуитивно чувствует, что полученное частное решение может быть использовано повторно, например, в других проектах. Он оформляет решение в виде «заготовки», в которой скрываются детали реализации. При необходимости, такой заготовкой (полуфабрикатом) можно будет воспользоваться в будущем.

При накоплении значительного количества заготовок разработчик начинает мыслить в терминах заготовок, а не в терминах

низкоуровневой реализации, т.е. мыслить в абстракциях более высокого уровня. Заготовки являются непосредственным примером повторного использования решений и накопления опыта.

Рассмотренный пример демонстрирует преодоление провала абстракции снизу-вверх путем абстрагирования. Рассмотренные в предыдущем пункте многоуровневые системы и характерные для них общесистемные слои также являются примером преодоления провала абстракции снизу-вверх путем абстрагирования.

С другой стороны, заказчик, видя те или иные проблемы разработки, помогает принять проблемные или неоднозначные решения, в которых разработчики не достаточно компетентны. По сути, он уточняет задание, постепенно вникая в реальные возможности разработки. Таким образом, провал абстракции преодолевается сверху-вниз.

Опытные проектировщики настоятельно рекомендуют привлекать заказчика к процессу разработки, объясняя ему преимущества и недостатки принимаемых решений и используемых технологий. В этом случае заказчик будет реально оценивать реализуемость своих пожеланий, а также помогать разработке, осознанно и грамотно принимая «генеральские» решения.

Провал абстракции и соответствующие способы его преодоления не зря называют основной проблемой проектирования. В дальнейшем понятия и приемы абстрагирования будут регулярно использоваться в данном документе.

2. ЦИФРОВЫЕ СИСТЕМЫ

В последние десятилетия повсеместное распространение получили цифровые технологии. Сказанное в полной мере относится и к системам с открытой архитектурой. Такое положение дел – не простая дань моде, а следствие тех преимуществ, которые предоставляют цифровые технологии. Анализ причин столь значимого успеха цифровых технологий позволяет перенять ценнейший опыт разработок сложных систем, чем не воспользоваться было бы недопустимо.

Пожалуй, наиболее значимую роль в успехе цифровых технологий сыграли: абстрагирование, обобщение, гибкость и автоматизация.

Компьютерная техника позволяет освободить разработчиков программного обеспечения от огромного пласта проблем, связанных с аппаратной реализацией вычислительного процесса. Наверное большинство программистов, в том числе и высококлассных, смутно себе представляют тонкости организации полупроводниковых устройств в кристалле процессора и других микроэлектронных элементов, но это не является существенным недостатком. Скорее наоборот, данное обстоятельство демонстрирует высокий уровень разделения труда по технологиям, который позволяет программистам сосредоточиться на решении своих прикладных задач. Цифровая аппаратура скрывает в себе проблемы микроэлектроники таким образом, что пользоваться ею можно, используя простые технологические приемы, доступные людям, не владеющим специальными знаниями физических процессов, происходящих в аппаратуре. Другими словами, процесс разработки программ абстрагирован от этих проблем.

Не меньшую роль в повсеместном распространении цифровой техники сыграло обобщение технологий практически на любую прикладную задачу. Давно стало очевидным, что подавляющее количество задач может быть сформулировано в виде их цифровых аналогов. В числовом виде может быть представлена практически

любая информация, независимо от того, какой прикладной области она принадлежит. Подобным образом и методы обработки информации могут быть представлены в виде соответствующих вычислительных алгоритмов.

Возможность перепрошивки цифровых систем новыми программами придает таким системам особую гибкость. В настоящее время технологии перепрошивки доведены до такого уровня, что провести эту операцию могут не только инженеры, но и бытовые пользователи. В считанные минуты можно модернизировать функциональность системы в той ее части, которая реализована на программном уровне. При этом система может не сниматься с эксплуатации, возможно только временное ее отключение. Учитывая такую возможность, разработчики систем стараются бóльшую функциональность реализовать на программном, а не аппаратном уровне, обеспечивая тем самым большую гибкость.

Цифровая техника способствовала высочайшему уровню автоматизации труда человека. Автоматы способны выполнять рутинные, многократно повторяющиеся действия. При этом им несвойственны характерные для людей недостатки: невнимательность, усталость, плохое самочувствие или настроение. Можно сказать, что автоматы «абстрагируют» человека от рутинной работы, оставляя за ним более интеллектуальную и творческую деятельность. Также важно отметить, что автоматы воплощают в себе знания своих разработчиков и, т.о. являются одним из способов накопления опыта.

Автоматы существуют очень давно. Возможно первыми автоматами были капканы, придуманные людьми много тысяч лет назад. Однако особую роль автоматы стали играть только в последнее столетие. Причина тому – новые технологии. На смену механическим пришли электрические, затем электронные ламповые, полупроводниковые, микроэлектронные, субмикронные и пр. устройства. Каждая последующая технология отличается от предыдущей размерами используемых элементов, точнее сказать, увеличивается *информационная плотность* устройств. Т.е. в одних и тех же габаритах можно реализовать больше функций или более сложные

функции. Современные цифровые системы даже бытового уровня способны выполнять миллиарды операций в секунду и хранить терабайты информации. О таких возможностях вряд ли мог помышлять инженер всего лишь несколько десятилетий назад. Сказанное подчеркивает современные возможности автоматов.

Указанные свойства придали цифровым системам такую технологическую и экономическую эффективность, которая обеспечила им существенные преимущества на рынке, что неминуемо привлекло инвесторов. Трудно себе представить какие колоссальные ресурсы вложило человечество в развитие цифровых технологий. И весь этот опыт, благодаря высокому уровню абстрагирования от сложных технологий и развитой инфраструктуре рынка, стал доступен широким слоям потребителей: от крупных предприятий до небольших фирм, владеющих простыми и дешевыми технологиями, и даже до бытовых пользователей. Цифровые системы представляют собой ярчайший пример повторного использования опыта, накопленного человечеством.

Стремление охватить, как можно больше ниш рынка, способствовало выпуску цифровых систем в виде полуфабрикатов различной степени готовности: отдельные элементы, модули, компьютеры, компьютерные системы. Степень готовности полуфабриката соответствует уровням абстрагирования во многослойных системах (см. п. 1.8). Каждый слой скрывает от вышележащих слоев определенные технологии. Чем выше слой, тем больше технологий он скрывает, и тем проще технологии требуются для использования полуфабрикатов данного слоя. В то же время, чем выше слой, тем менее он универсален.

Например, для того, чтобы воспользоваться отдельными электронными элементами, необходимо владеть технологиями проектирования печатных плат, их изготовления, пайки, отладки, контроля и пр. Спроектировать таким способом изделия можно практически для любой области применения. Если же покупаются не элементы, а готовые модули, то при их использовании можно ограничиться более простыми технологиями: изготовлением жгутов, посадочных мест, корпусов, сборкой и т.д. Однако область

применения таких систем более ограничена, поскольку спектр имеющихся в продаже модулей ограничен только теми, которые наиболее востребованы на рынке.

Особый интерес представляют собой изделия, предназначенные для конечного пользователя, в частности, – обывателя. Как правило, массовость продаж подобных изделий наибольшая, за счет чего достигается значительная прибыль.

2.1. Программные технологии

Нижние уровни цифровых систем – аппаратные. Начиная с некоторого уровня (уровня абстракции) используются только технологии программирования. Программное обеспечение, в свою очередь, имеет много подуровней: операционная система, драйверы, библиотеки, системы управления базами данных, платформы, различного рода САПРы и т.д.

На сегодняшний день программные проекты могут служить примером самых сложных систем, из когда-либо разрабатывавшихся человечеством. Подтверждением тому служит количество привлекаемых разработчиков и объем бизнеса таких мировых софтверных гигантов как Microsoft, IBM, Oracle, Apple, Adobe и пр. Доля программного обеспечения в общей стоимости проектов в течение последних десятилетий неуклонно росла, в то время как доля аппаратных средств уменьшалась. В настоящее время доля программного обеспечения оценивается примерно в 80% стоимости, тогда как ранее, безусловно, доминировала аппаратная составляющая [5].

Непрерывно совершенствующиеся технологии разработки программ привнесли значительный вклад в мировой опыт проектирования сложных систем. Несмотря на специфические нюансы все приемы проектирования программного обеспечения (ПО) в основной своей сути сводятся к рассмотренным выше идеям: разделение труда, повторное использование наработок, абстрагирование и т.д. Чтобы убедиться в этом рассмотрим ряд примеров, никоим образом не претендуя на полноту и хронологическую точность изложения.

2.2. Низкоуровневое программирование

Первые программы разрабатывались в *машинном коде* – последовательности инструкций процессора, закодированных в виде нулей и единиц. Такое представление программы очень трудно воспринимается человеком, в результате чего объемы и сложность создаваемых программ были незначительны (килобайты).

Существенным прорывом было изобретение языка *Ассемблера*. Ассемблер представляет собой мнемоническую запись инструкций процессора в текстовом виде, который легче воспринимается человеком. В русле рассмотренного ранее материала его можно рассматривать как уровень абстракции, используемый при разработке программ.

Исходный код программы на языке Ассемблера, транслируется в машинный код другой программой, также называемой *Ассемблером*. Процесс трансляции происходит автоматически. Другими словами, транслирующая программа или *транслятор* (в данном случае называемый Ассемблером) является автоматом, в который заложен опыт ее разработчиков (привлечение стороннего опыта). Кроме трансляции Ассемблер выдает подсказки разработчику в виде сообщений об обнаруженных и потенциальных ошибках, что также автоматизирует процесс разработки.

Еще одним инструментом разработки сложных программ было их разбиение на *подпрограммы* или *процедуры*. Неоднократно повторяющиеся действия выделялись в отдельную подпрограмму, которая вызывалась из основной программы, а после своего завершения возвращалась в точку вызова. Основная программа может содержать множество вызовов подпрограмм. Для сборки программы из отдельных файлов разрабатываются специальные программы-*компоновщики* или т. н. *редакторы связей*.

Следующим шагом было появление *операционных систем* (ОС). За время своего развития ОС прошли путь от простых библиотек с процедурами ввода/вывода, до сложных многозадачных систем. ОС по своей сути представляет собой каркас, обеспечивающий совместный доступ разных программ к общим ресурсам

компьютера. Современная ОС предполагает, что на компьютере может выполняться не единственная прикладная программа, а множество, причем набор и назначение этих программ практически никак не ограничиваются. Набор программ и сама ОС могут конфигурироваться, подстраиваться под конкретное использование компьютера. Можно сказать, что ОС представляет собой каркас, инфраструктуру или платформу открытой системы. Отметим многократное повторное использование однажды разработанной определенной фирмой ОС (своего рода полуфабриката) в разных приложениях разными компаниями.

Разбиение программы на составные части – пример декомпозиции задачи. Разные части могут разрабатываться разными специалистами, что обеспечивает разделение труда. Выявление повторяющихся действий и их выделение в отдельные подпрограммы служит примером преодоления «Провала абстракции» снизу-вверх, т.е. обобщением или абстрагированием. Однажды разработанные подпрограммы могут впоследствии многократно использоваться (повторное использование удачных решений), а накопление множества подпрограмм позволяет в дальнейшем производить сборку программы методом агрегирования из готовых решений, т.е. на более высоком уровне абстракции. В процессе разработки активно используются автоматизированные инструменты: трансляторы, компоновщики, отладчики и пр. Зарождаются каркасы или инфраструктуры открытых систем в виде ОС.

2.3. Языки высокого уровня

Можно видеть, что уже на заре своего развития приемы программирования использовали основные принципы разработки сложных систем. Дальнейшее развитие технологий, по сути, использует те же принципы, но делает их существенно более эффективными. Рассмотрим вкратце основные этапы последующего развития технологий программирования.

Важной вехой для технологий разработки программ стало появление в 50-70х годах прошлого века т. н. *языков высокого уровня:*

FORTRAN, COBOL, ALGOL, C, PASCAL и пр. В отличие от Ассемблера эти языки абстрагированы от набора инструкций конкретной модели процессора (языки, обладающие подобными свойствами, называют *аппаратно-независимыми*). Данное обстоятельство позволяет программе или подпрограмме, однажды написанной на языке высокого уровня, быть повторно использованной на процессорах, отличающихся наборами инструкций, архитектурой и пр. Таким образом, резко увеличились возможности повторного использования удачных решений, расширения области их применения. Массово начали появляться и распространяться всевозможные библиотеки подпрограмм (IMSL, LAPACK, BLAS, ГРАФОР, CNL, NAG, Дубна, ITAG, PORT и пр.), которые позволяют эффективно накапливать готовые решения, а также использовать сторонний опыт. Библиотеки существенно повышают уровень абстракции разработки.

Для *портирования* (переноса) программы на процессор другой модели необходимо взять исходный код, из него создать новый проект для соответствующего компилятора и откомпилировать. Нередко часть файлов все же приходится изменять. Грамотное проектирование позволяет существенно уменьшить количество изменений исходного кода и локализовать их. Тем не менее, операция портирования автоматизирована не полностью и требует квалифицированного ручного труда.

Программы, написанные на языках высокого уровня, как правило, уступают в производительности, поскольку унификация не позволяет воспользоваться всеми ресурсами конкретных моделей процессоров. Несмотря на подобные накладные расходы, дивиденды, приносимые за счет унификации и обобщения, оказываются гораздо более весомыми.

Для повышения эффективности многие языки высокого уровня предусматривают расширения, например, ассемблерные вставки в код программы, написанной на языке высокого уровня. Поскольку такие вставки делают код непереносимым на другие процессоры, их, как правило, локализуют в отдельных файлах, чтобы другие части программы оставались универсальными.

2.4. Объектно-ориентированные языки

Рассмотренные выше, т. н. *процедурные* языки высокого уровня навязывают разработчику метод декомпозиции программы на подпрограммы или процедуры. Такое разбиение на модули было естественным следствием того, что процессор воспринимает программу в виде последовательности инструкций. Собственно эта последовательность и разбивалась на подпоследовательности, т.е. на процедуры.

Декомпозиция программы на процедуры оказалась однобокой в том смысле, что не учитывает вторую важную составляющую сущность программы – данные. Во многих приложениях именно структура используемых данных играет ключевую роль. В этом смысле характерно высказывание одного из основоположников программной индустрии Фрэда Брукса в его знаменитом труде [15]: «Покажите мне ваши алгоритмы, не показывая структуры данных, и я останусь в заблуждении. Покажите мне структуры данных – и алгоритмы, скорее всего, не понадобятся – они будут очевидны».

С 80х годов прошлого столетия особенную популярность получил другой подход к декомпозиции (следовательно, и к проектированию) программ, основанный на понятии *объектов* – сущностей, объединяющих в себе как данные, так и методы для работы с этими данными (т.н. *объектно-ориентированное программирование* – ООП). В настоящее время, подавляющее количество языков программирования поддерживают концепцию ООП. Многие процедурные языки со временем приобрели свойства объектно-ориентированных: C++ (бывший C), Object Pascal (бывший Pascal), Ada, Modula и др. Появились языки, изначально ориентированные на объекты: Smalltalk, Java, C#, Python, Ruby и пр.[1]

Обычно к основным механизмам ООП причисляют: *инкапсуляцию, наследование* и *полиморфизм,* которые обеспечивают пре-

[1] Первым объектно-ориентированным языком считается Simula. В нем реализованы почти все идеи ООП. Однако эти идеи не были оценены по достоинству в то время (конец 60х годов прошлого столетия).

имущества данного подхода, повышающие уровень абстракции и гибкости программ. Не вдаваясь в подробности, акцентируем внимание на новом способе декомпозиции программы. ООП навязывает в качестве *единицы модульности* программы не процедуру, а объект или *класс*. Понятие класса более общо, чем процедура (по крайней мере, класс включает в себя процедуры), но, что более важно, оно лучше моделирует реальный мир, в котором присутствует множество взаимодействующих объектов. Такое представление привычно человеку и делает программу более понятной. Понятные и простые технологии позволяют создавать более сложные системы.

Понятие *абстрактных базовых классов* позволяет проектировать программу «крупными мазками», не вдаваясь в детали реализации. Причем спроектированные решения будут не просто эскизами, черновиками или вспомогательными документами, а реальными работающими частями кода создаваемой программы. Механизмы наследования и полиморфизма позволяют впоследствии конкретизировать абстрактные классы. Делается это поступательно. Сначала многие конкретные объекты представляют собой «заглушки» или «пустышки», но с ними программа может запускаться и тестироваться. Постепенно, по мере отладки, заглушки заменяются реальными объектами. Функциональность программы последовательно наращивается до нужного уровня. Такой способ разработки, называемый *инкрементным,* оказался очень эффективным и получил широкое распространение.

Механизм подмены объектов их заглушками на начальной стадии разработки (см. выше) оказывается очень полезным при модернизации программ. «Незаметная» подмена объектов их новыми версиями дает возможность проектировать очень гибкие программные продукты.

Мода на ООП привела к появлению огромного числа библиотек классов. Некоторые из таких библиотек являются неотъемлемой частью языков программирования и поставляются совместно с транслятором (т.н. *библиотеки времени исполнения* или *runtime*-библиотеки): BCL, JFC, STL, tkinter и пр. Остальные библиотеки

являются расширением стандартных средств языка: MFC, VCL, Qt, Boost, ACE, POCO, wxWidgets, JFace, SciPy и др.

Можно утверждать, что ООП устоялся и прочно занял свое место в инструментариях разработки программного обеспечения. Однако эволюция его не завершена. Со временем появились нововведения: перегрузка операций, параметризованные типы и функции, делегаты, замыкания, лямбда-выражения и пр., которые повысили выразительность и мощь объектно-ориентированных языков[1].

2.5. Графические языки, CASE-технологии

Языки высокого уровня воспринимаются человеком значительно легче, чем Ассемблер и, тем более, машинный код. Однако синтаксис языков содержит множество деталей, которые отвлекают от основной идеи программы. Чтобы упростить восприятие алгоритмов используют т. н. *псевдокод,* т. е. упрощенный язык программирования, в котором скрыты малозначимые детали. Псевдокод не предназначен для трансляции и последующего выполнения на компьютере, а используется для демонстрации/обсуждения отдельных решений на ранних стадиях проектирования, в учебниках и т. п. Стандартов псевдокода не существует и каждый автор вправе изобретать свой язык, лишь бы он более наглядно демонстрировал его мысли.

Псевдокод является примером абстрагирования, который приносит определенную пользу для упомянутых задач. Но текст программы даже на языке высокого уровня – не лучшая для восприятия человеком форма представления информации. Большинство людей легче воспринимают изображение, чем текст. С целью представить программу в более наглядном схематическом виде были разработаны различные *графические языки* программирования (ДРАКОН, VisSim, G и др.). Наиболее употребительными в на-

[1] Незначительные улучшения языка программирования, повышающие его выразительность, называют иногда синтаксическим сахаром.

шей стране были т. н. *блок-схемы* (ГОСТ 19.701-90, ISO5807-85). В последнее время набрал популярность язык UML (Unified Modeling Language), предназначенный для объектно-ориентированного моделирования и анализа программного обеспечения, бизнес-процессов, организационных структур и пр.[1]

Следом за графическими языками появились системы, автоматизирующие процесс разработки программ на основе визуального представления – т. н. CASE-технологии (Computer-Aided Software Engineering). Такие системы предоставляют графические редакторы, средства анализа и документирования, трансляторы с языка схем в один или несколько языков высокого уровня и обратно (Rational Software Architect, Umbrello, ERwin, Simulink, LabVIEW, ГРАФИТ-ФЛОКС и др.).

Рассмотренные технологии повышают уровень абстракции и автоматизации разработки. В определенных, как правило, узких областях эффективность разработок повысилась существенно (Simulink, LabVIEW). Что же касается графических языков (UML) и CASE-систем, претендующих на самый широкий спектр задач, то их успехи оказались не столь значительны.

Один из недостатков UML заключается в его нестрогости, приближенности описания, что не позволяет из UML-диаграмм генерировать код программы с учетом всех аспектов, особенно системных: безопасности, отказоустойчивости, производительности и пр. В то же время приближенность UML позволяет обсуждать ключевые решения на ранних стадиях проектирования, абстрагировавшись от множества деталей реализации. По указанной причине UML де-факто стал стандартом описания архитектуры программ и широко используется при документировании.

[1] Тот факт, что объектно-ориентированный подход годится не только для моделирования программных продуктов, но и для социальных, организационных, физических и др. приложений, подтверждает правильность заложенных в него идей.

2.6. Идиомы. Паттерны проектирования

Выше внимание неоднократно акцентировалось на значении повторного использования удачных решений. Для повторного использования предназначены процедуры, библиотеки, классы и т. д. Но удачные решения могут быть использованы не только напрямую, как часть программы. Передача опыта в виде советов, консультаций, обучающих курсов, учебников также являются повторным использованием решений, хотя и косвенным.

Одним из лучших способов передачи опыта, является разбор конкретных практических примеров. Подобными примерами всегда изобиловала обучающая литература. Вскоре выяснилось, что многие приемы программирования можно обобщить на широкий спектр задач. Поначалу такие приемы появлялись применительно к конкретному языку программирования и назывались *идиомами* [7].

Очевидно, что многие идиомы применимы не только для конкретных языков, но и для программирования вообще, т. е. являются «высокоуровневыми» идиомами. Применительно к объектно-ориентированному проектированию такие идиомы стали называть *паттернами* (pattern) или *образцами*[1] проектирования. Ключевой вехой в данном направлении стала работа [4], опубликованная в 1994 г. В книге [4] приводится 23 паттерна, представляющие собой примеры решения проблемы проектирования в некоторых, часто возникающих ситуациях. Каждый паттерн рассматривается по одинаковой схеме.

Особое внимание уделяется названию паттерна – оно должно лаконично и точно отражать назначение паттерна. Хорошо проду-

[1] Часто в данном контексте слово pattern переводят с английского, как шаблон. Такой перевод представляется не совсем удачным, поскольку слово шаблон ассоциируется с автоматической генерацией кода, точно соответствующей данному шаблону. Однако сами авторы книги говорят, что паттерны – это всего лишь примеры, приближенные образцы решений типовых проблем. В конкретных приложениях используется только идея образца, а не точная его копия.

манное имя сразу описывает проблему, ее решение и последствия, т. е. инкапсулирует эту информацию. Использование словаря паттернов в обсуждениях и в документации позволяет вести разработку на более высоком уровне абстракции.

Для каждого паттерна подробно описывается соответствующая проблематика. Приводятся конкретные проблемы проектирования, которые позволяет упростить данный паттерн. Описывается контекст проблемы.

Далее рассматривается само решение. Описываются элементы (объекты-участники) дизайна, их роли и отношения. Описание дается в обобщенном виде с использованием UML. Кроме того даются примеры реализации паттерна на языках C++ и/или Smalltalk.

Напоследок обсуждаются результаты применения паттерна. Область применения, преимущества и недостатки, разного рода компромиссы.

Для примера на рис. 5 приведен паттерн «Адаптер» или «Обертка». Данный паттерн преобразует интерфейс одного класса (Adaptee) к интерфейсу другого (Service), который ожидают клиенты (Client).

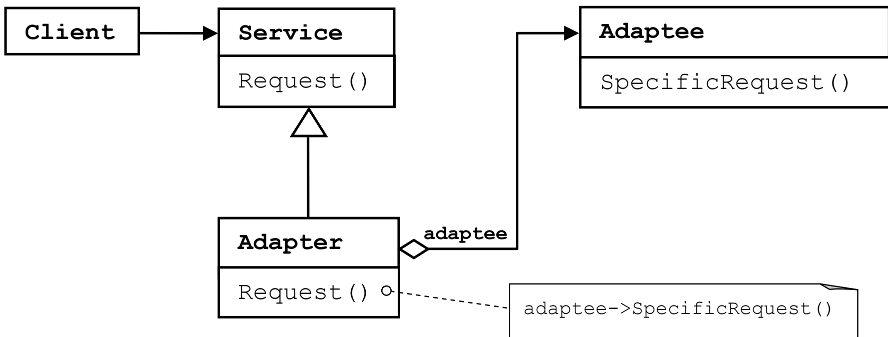


Рис. 5. Паттерн «Адаптер»

В паттерне используется класс-посредник Adapter, который реализует интерфейс Service (указано стрелкой **Δ**) и содержит ссылку adaptee на объект типа Adaptee (показано стрелкой ◇). При

обращении Client к методу Request посредника, последний перенаправляет обращение к adaptee, вызвав соответствующий метод SpecificRequest.

Как можно видеть, паттерн «Адаптер» представляет собой статическую структуру объектов и взаимоотношений между ними. Из подобных статических структур состоят и другие паттерны в отличие от алгоритмов, представляемых в виде временной последовательности действий. Данную структуру можно рассматривать как строительный блок или одну из архитектурных конструкций для построения приложения.

Паттерн «Адаптер», как и другие паттерны, решает локальную проблему, один из аспектов программы. Паттерны не претендуют на масштаб всего приложения, но они и не столь примитивны, как, например, списки или таблицы, которые можно реализовать один раз в виде классов библиотеки и потом повторно использовать без изменений. Паттерны – это примеры, образцы, которые следует осознанно адаптировать под конкретную задачу.

Вслед за [4] начали появляться и другие паттерны, например, паттерны серверных приложений на платформе Java EE, GRASP (General Responsibility Assignment Software Patterns – общие образцы распределения обязанностей), паттерны параллельного программирования [17] и пр. Примечательно появление *антипаттернов,* т. е. примеров часто повторяемых ошибок проектирования (пример: антипаттерн «Большой комок грязи» – система с трудно распознаваемой структурой).

Паттерны оказались эффективным инструментом абстрагирования разработки и обмена опытом, независящим от языков и систем программирования/проектирования.

2.7. Роль архитектуры

Как говорилось, паттерны не претендуют на масштаб всего приложения, а решают только локальные проблемы. Чтобы спроектировать серьезное приложение, нужно увязать между собой множество паттернов. По сути, проектирование с использовани-

ем паттернов представляет собой агрегацию из «кирпичиков», которыми в данном случае являются не классы или процедуры, как раньше, а паттерны, т.е. элементы более высокого уровня абстракции (паттерн – это группа взаимосвязанных объектов, а не отдельные объекты). Более высокий уровень абстракции упрощает понимание системы и, как следствие, облегчает ее проектирование. Это означает, что можно создавать более сложные системы, т.е. повысить порог их сложности. Однако при приближении к новому порогу возникают старые проблемы, связанные со сложностью.

Напрашивается вопрос: можно ли еще отодвинуть порог сложности, оставаясь на том же уровне абстракций, т.е. используя те же «кирпичики»?

Один из ответов кроется в том, насколько понятными окажутся взаимосвязи между «кирпичиками» в приложении. При удачном решении взаимосвязи просты и понятны, несмотря на их количество. При неудачном – связи запутанны настолько, что их почти невозможно воспринять (т.н. антипаттерн «Спагетти»). Очевидно, что простые взаимосвязи в системе позволяют создавать системы большего масштаба. В качестве примера можно привести управление армией. Преимущество имеет та армия, у которой более четкая структура и дисциплина. Армия без четкой иерархии и дисциплины превращается в плохо управляемую толпу. По аналогии сложная система без упорядочивания превращается в плохо контролируемый хаос.

Как же добиться того, чтобы связи между компонентами системы были просты и понятны?

На данный момент однозначного ответа на поставленный вопрос не существует, хотя разработчики программного обеспечения давно осознали его важность и тратят значительные усилия в данном направлении. В настоящее время лишь выработаны некоторые рекомендации общего плана, но четких технологий пока нет. Например, одна из рекомендаций состоит в том, чтобы выявить (предугадать) в создаваемой системе регулярные связи, обеспечивающие однотипное подсоединение к системе большинства компонентов. Однако слово «выявить» без четких методик относится больше к искусству, чем к технике.

Простота и наглядность связей в системе сами по себе не являются критериями ее качества. Это скорее способ решения проблемы. Ни заказчик, ни пользователи обычно не выдвигают требований по внутренней организации системы, а говорят об ее потребительских, эксплуатационных и пр. характеристиках. Внутренняя организация системы – это вотчина разработчика. Однако внутренняя организация сложной системы оказывается настолько значимой, что во многом предопределяет ее успех и потребительские свойства.

Правила организации системы связывают с ее *архитектурой*. Существует множество определений данного понятия. Например, на ресурсе *http://www.sei.cmu.edu/architecture/definitions.html* приводится более сотни определений архитектуры. Приведем в качестве примера определение согласно стандарту IEEE 1471: «Архитектура – это фундаментальная организация системы, реализованная в ее компонентах, отношений этих компонент друг с другом и с внешней средой и принципах, определяющих структуру и развитие системы».

Из множества определений архитектуры можно выделить следующие наиболее часто встречающиеся положения[1]:

1. Архитектура определяет структуру системы, а именно устанавливает правила декомпозиции на компоненты (агрегирования из компонент), связи между ними. В частности, правила могут предусматривать возможность переконфигурации системы, наращивания ее новыми компонентами.
2. Архитектура рассматривает только значимые решения в системе. Значимыми считаются те решения, которые определяют ключевые свойства системы, такие как масштабируемость, надежность и пр., а также решения, затратные в смысле разработки и изменений. Значимые решения д.б. устойчивыми (неизменными) во времени, т. к. их изменение чревато масштабными переделками.
3. Архитектура увязывает различные, зачастую противоречивые,

[1] Питер Илес. Что такое архитектура программного обеспечения? http://www.ibm.com/developerworks/ru/library/eeles/

требования к системе, в частности, требования заказчика, конечного пользователя, разработчика, производителя и пр.

4. Архитектура влияет на кооперацию разработчиков, поскольку декомпозиция системы на компоненты во многом определяет разделение труда. Часто складывается обратная ситуация, когда архитектура повторяет сложившуюся (например, стихийно) кооперацию разработчиков. Такой подход нередко приводит к плохим результатам.
5. Архитектура программного обеспечения в основном определяет не *прикладную функциональность* системы (функциональность, которую использует конечный потребитель), а *системную,* т.е. служебные функции.
6. Архитектура может соответствовать одному из архитектурных стилей. Архитектурные стили очень похожи на паттерны или образцы, рассматривавшиеся ранее, но в отличие от последних имеют не локальный масштаб, а масштаб всего приложения. Образцами могут быть известные удачные проекты, например, WWW, Java, Eclipse и др. Некоторые архитектурные стили получили собственные имена: «Классная доска», «Клиент-сервер», «Сервис-ориентированная архитектура» и пр.

Форма описания архитектуры на данный момент не стандартизована. В большинстве случаев архитектурная документация оформляется по внутренним правилам той или иной организации. Тем не менее, попытки формализовать описание архитектур не прекращаются. Различными фирмами разработано несколько языков описания архитектур: AADL, Wright, Acme, xADL и др. На стандарт языка описания архитектур де-факто претендует UML.

2.8. Каркасы, фреймворки

Архитектура носит не только описательный характер в виде проектной документации, различного рода диаграмм и пр. В конечном итоге архитектура воплощается в программном продукте. В одних приложениях она размыта по всем компонентам программы и не вычленена в отдельные модули. В других – архитектурные

конструкции реализованы в отдельных модулях и четко отделены от той части программы, которая реализует прикладную функциональность. Наиболее ярким примером второго варианта являются т.н. *каркасы.*

Каркас представляет собой неизменяющуюся (мало изменяющуюся) часть программы, которая реализует системную функциональность приложения. В каркасе предусмотрены «посадочные места», для подстыковки пользовательских модулей. Такие посадочные места называют еще *точками расширения.* В отдельных модулях производится *сборка* программы – наполнение каркаса пользовательскими модулями, т. е. конфигурирование. Здесь же обычно производится и настройка других параметров приложения.

Поскольку каркасы реализуют архитектурные решения, а те по своему замыслу должны быть стабильными (мало подверженными изменениям), то каркасы также должны быть стабильными. Стабильность каркасов делает их претендентами для многократного повторного использования в типовых приложениях, конфигурируемых под нужды конкретных пользователей. Типовые приложения на основе единого каркаса называют *линейкой продуктов.* Часто каркасы добавляют в библиотеку для повторного использования в других проектах.

Важное свойство каркаса заключается в том, что он может запускаться пустым, т.е. без пользовательских модулей. Данное свойство позволяет создавать программу поэтапно, постепенно начиняя каркас все новыми и новыми модулями, наращивая, таким образом, функциональность программы (инкрементная разработка). На начальных этапах, пока разрабатываемая система невелика, отлаживаются ключевые системные функции: безопасность, коммуникации и пр.

Отличительная черта каркаса от обычной библиотеки – т.н. *инверсия управления.* Когда используется обычная библиотека, то пользователь в своем коде вызывает библиотечные методы. При использовании каркаса наоборот – запущенный каркас сам вызывает методы пользователя, ранее зарегистрированные в каркасе.

Данный прием известен еще как принцип Голливуда: «Не звони нам, мы сами позвоним тебе».

При внимательном рассмотрении можно увидеть, что программа на основе каркаса представляет собой ничто иное, как систему с открытой архитектурой (см. п. 1.6), где каркас – ее инфраструктура. Следовательно, все преимущества открытых систем остаются в силе.

Разработка с использованием каркаса может выглядеть следующим образом. Разработчик берет из библиотеки готовый каркас, из той же или из других библиотек берет стандартные модули для начинки каркаса и осуществляет сборку программы в нужной конфигурации (агрегирование). Недостающие модули разрабатываются заново, но их, как правило, немного.

Указанная процедура проста и легко автоматизируется. Как следствие, на рынке возникло множество систем разработки, ориентированных на каркасную архитектуру создаваемых приложений. Часто такие системы возникали на базе популярных библиотек, которые развивались, расширяли свою функциональность, дополнялись специфическими средствами разработки. Со временем во многих системах программирования появилась возможность автоматической генерации каркасов приложений для специфических областей применения. Такие системы разработки стали называть *фреймворками*. Подобный путь прошли Delphi, Qt и пр. Некоторые системы изначально разрабатывались, как фреймворки (Joomla!, 1C и др).

Одними из первых фреймворков были системы программирования графического интерфейса пользователя. В свое время ярким событием стало появление систем *визуального программирования,* например, Delphi. Разработчик в буквальном смысле рисует на экране компьютера нужный интерфейс программы из готовых «кирпичиков» – компонентов пользовательского интерфейса, предоставляемых библиотекой VCL. Среда Delphi по созданному интерфейсу генерирует каркас программы автоматически. Разработчику остается реализовать отдельные специфические, как правило, небольшие части кода.

На примере Delphi можно проследить следующий прием привлечения сторонних разработчиков. Среда поставляется со встроенным набором типовых визуальных компонент библиотеки VCL, кроме того, пользователям предоставляются инструменты для разработки своих собственных компонент и добавления их к общей палитре компонент. Таким образом, к проекту было бесплатно привлечено большое количество третьих фирм и отдельных разработчиков, которые существенно расширили функциональность системы, популяризировали и усилили ее рыночную привлекательность (библиотеки RxLib, JEDI, GLScene и др.).

2.9. Рефакторинг

Со временем проектировщики программного обеспечения все больше и больше осознают значимость грамотной архитектуры. Показательным примером этого может быть повсеместное распространение т. н. *рефакторинга* [10]. Рефакторинг представляет собой переделку работающей программы с целью улучшения ее внутренней организации, архитектуры. При этом прикладная функциональность программы, как правило, не меняется. Заказчик, не видя изменений во внешнем поведении программы, редко соглашается на финансирование подобного рода доработок. В таких случаях разработчики проводят рефакторинг за свой счет, осознавая, что поддержка и модификации программы с плохой архитектурой обойдутся гораздо дороже.

Значение рефакторинга подчеркивает и тот факт, что многие языки и системы программирования стали включать в себя инструменты для рефакторинга. При этом рефакторинг рассматривается не как реакция на некоторый исключительный провал в проектировании, а как нормальный технологический этап разработки.

2.10. Проблема увязки конкурирующих технологий

Рассмотренный выше материал демонстрирует важные вехи в совершенствовании технологий программирования. Для простоты изложения многие аспекты не раскрывались. В частности, не раскрывались проблемы связанные с выбором и увязкой множе-

ства конкурирующих инструментов и технологий, в значительной мере дублирующих друг друга. Сказанное касается аппаратных средств, операционных систем, языков программирования, компиляторов, библиотек, фреймворков, протоколов обмена и пр.

Разнообразие и избыточность инструментария, с одной стороны, усложняет процесс разработки из-за необходимости увязки разных технологий, с другой стороны, является следствием конкуренции, в которой инструментарий совершенствуется.

Попытки унифицировать набор инструкций процессора, разработать единый язык высокого уровня, стандартизовать программный интерфейс операционных систем и пр. проводились неоднократно. Подобные усилия приносили определенные плоды в некоторых областях, но в целом ситуация мало изменилась. Нередко конкурирующие фирмы-разработчики умышленно дублируют технологии, чтобы сегментировать рынок и завоевать его часть. Так или иначе, многообразие близких по функциональности стандартов и инструментов следует воспринимать, как объективную реальность.

В качестве примера можно привести длительные споры разработчиков о том, какой из языков программирования лучше. За ожесточенность и безрезультатность попыток убедить оппонента такие споры получили жаргонное название «священные войны».

В истории развития компьютерной индустрии ощутимо выделяются две вехи, с одной стороны, послужившие массовому распространению компьютеров, с другой стороны, повлекшие за собой хаос в технологиях разработки. Первая из них – появление персональных компьютеров в начале 1980-х гг (массовый продукт для конечного пользователя – прибыльный рынок, см. выше). Вторая – всемирная сеть WWW. Сейчас можно говорить о новой волне, связанной с распространением мобильных устройств.

Сеть Интернет открыла чрезвычайные возможности для создания масштабных систем. Такие системы состоят из взаимодействующих программных и аппаратных компонент, разбросанных по предприятию, по отрасли, по стране, по всему миру. В то же время, компоненты написаны на разных языках программирования,

развернуты на различных платформах, обмениваются данными по каналам с разными физическими и программными протоколами, регулярно меняется их структура, функциональность и т. д.

Увязка между собой всего комплекса подобных проблем является одной из основных проблем компьютерной индустрии последних 20 лет. Ею занимаются крупнейшие мировые корпорации, затрачиваются миллиардные инвестиции. Анализ данной проблемы позволяет перенять богатый опыт проектирования сложных систем. Некоторые аспекты проблемы рассматриваются ниже, начиная с самых ранних этапов.

2.11. Увязка языков программирования. Декомпозиция программ на исполнимые модули

Как было показано выше, независимость исходного кода программы от марки процессора обеспечивается применением языка высокого уровня и соответствующего транслятора. Однако наличие большого числа языков создает новые проблемы. В данном пункте приводятся примеры увязки нескольких языков программирования в одной программе.

Простейшим является случай, когда программа создается на одном языке. Для небольших программ это идеальный вариант. Однако в больших проектах появляются объективные причины для использования нескольких языков, например, при кооперации разработчиков, использующих разные системы программирования. Другим примером может быть использование специализированных (следовательно, более эффективных) языков для соответствующих подзадач в рамках одной программы.

Одна из идей основана на том, что любой программный код, написанный на языке высокого уровня, в конечном итоге должен быть транслирован в машинные коды. После этого уже не важно, на каком из языков была изначально написана программа. Данное обстоятельство подсказывает очевидный способ увязки разных языков программирования на уровне машинного кода. Файлы с

подпрограммами на разных языках высокого уровня компилируются соответствующими компиляторами в т. н. *объектные модули,* состоящие из машинных кодов и некоторой дополнительной информации для сборки будущей программы (имеются стандарты объектных модулей COFF, ELF, PE и др.). Далее программа-компоновщик компонует из объектных модулей исполняемую программу. Такой способ компоновки (до запуска программы) называют *статическим связыванием.*

Проблемным моментом технологии является несоответствие механизмов передачи аргументов при вызовах процедур, реализуемых разными компиляторами. Решается проблема с помощью специфических (часто нестандартных) приемов, но и это не гарантирует их полной совместимости. Чаще всего совместимыми оказываются компиляторы, разработанные одной фирмой (например, Microsoft: C/C++, Visual Basic, Fortran).

Со временем бо́льшую популярность завоевали *динамически подгружаемые библиотеки* (DLL – Dynamic Loaded Library). В отличие от статического, *динамическое связывание* производится во время выполнения программы (при ее запуске или по мере необходимости). Установление связей в данном случае производит не программа-компоновщик, а операционная система. Программа уже состоит не из единственного исполнимого файла, в котором содержаться все нужные подпрограммы, а из целого набора DLL-файлов, дополняющих вызывающую программу. Можно говорить о декомпозиции программы на уровне исполняемых модулей.

В данном случае передача аргументов процедурам стандартизируется требованиями ОС. Компиляторы должны подстраиваться под эти требования, например, вводя нестандартные для языка модификаторы процедур, что несколько затрудняет переносимость кода. Это затруднение покрывается дополнительными преимуществами динамического связывания.

Некоторые DLL являются сугубо специфическими для конкретной задачи, а некоторые могут использоваться и другими приложениями. В последнем случае можно предоставить соответствующие DLL в общее пользование, т.е. сделать их общим ресурсом

компьютера (один и тот же код библиотеки, загруженный в ОЗУ, может одновременно использоваться несколькими программами). Удобным становится также и модификация установленных программ путем подмены старых DLL их новыми версиями без перекомпоновки. Подмена позволяет полностью изменить реализацию модуля, чего нельзя сказать о его спецификации – интерфейс поменять таким образом невозможно.

Немалое значение имеет то обстоятельство, что библиотеки могут поставляться на рынок в бинарном виде без открытия исходного кода на языке высокого уровня, чем нередко пользуются компании для защиты своих авторских прав и ноу-хау.

Итак, механизм динамических библиотек позволяет увязывать в одной программе модули, написанные на разных языках разными разработчиками. Из DLL удобно формировать общие ресурсы, их легко модернизировать даже во время эксплуатации. Преимущества предопределили их широкое использование, в частности, ОС Windows была реализована на базе DLL.

Недостатком динамических библиотек является то, что откомпилированные модули пригодны только для конкретных типов процессоров с установленной на них конкретной ОС. Другая проблема, когда различные приложения требуют для своей работы разные версии одной и той же DLL (не совместимые или не полностью совместимые версии), получила название «DLL-кошмара».

2.12. Многозадачность

Динамические библиотеки отчасти решали проблему в пределах одной программы. Бóльшие возможности декомпозиции и агрегирования открываются при организации взаимодействия нескольких программ *(процессов)* в пределах одного процессора. Механизмы обмена данными и синхронизации (т. н. IPC – Inter Process Communication) обеспечиваются *многозадачной операционной системой* (сообщения, общие области памяти, семафоры и пр.). Поскольку обмен данными осуществляется посредством операционной системы на уровне машинных кодов, то каждая из

взаимодействующих программ может быть разработана на любом из языков программирования.

Некоторые языки программирования, например Ada, предоставляют механизмы для взаимодействия между процессами, абстрагированные от используемой ОС. Однако данные механизмы обеспечивают в основном взаимодействие между процессами, написанными на одном языке.

Для языков, которые не предоставляют механизмы взаимодействия между процессами, разрабатываются библиотеки-надстройки над ОС или т. н. *фасады* (соответствуют паттерну «Фасад»). Фасады определяют упрощенный и обобщенный программный интерфейс (API – Application Program Interface) взаимодействия процессов. При портировании приложения под другую ОС интерфейсы фасадов остаются неизменными – меняется только реализация библиотеки (точнее небольшие ее части). Прикладная часть программы не изменяется, поскольку использует неизменный интерфейс фасадов. Примером такого подхода является библиотека ACE [11, 12]. Данная библиотека, написанная на C++ и портированная под десятки ОС, получила широкое распространение в промышленности, особенно для встраиваемых систем.

Также начали появляться технологии, которые позволяли не только посылать/принимать сообщения между приложениями, но и вызывать методы (подпрограммы) одного приложения из другого. Одной из таких технологий была модель COM (Component Object Model). COM обеспечивает взаимодействие программных компонент на компьютере под управлением Windows в объектном стиле. Компоненты-серверы регистрируют в системе свои сервисы в виде классов. Компоненты-клиенты обращаются с запросами к зарегистрированным в системе сервисам.

Примечательной особенностью технологии COM является введения языка интерфейсов MIDL (Microsoft Interface Definition Language). MIDL не предназначен для программирования исполняемого кода. На нем специфицируются интерфейсы COM-серверов, которые затем регистрируются в системе и которыми пользуются

клиенты. Можно сказать, что язык интерфейсов, типа MIDL, является абстракцией более высокого уровня, используемый для взаимодействия программ.

2.13. Сетевые ОС

Рассмотренное приложение, состоящее из взаимодействующих процессов (программ, компонент, задач) в пределах процессора, имеет больше возможностей, чем приложение из единственного процесса. Еще больший масштаб и потенциал имеют приложения, состоящие из компонент на нескольких (или на множестве) географически удаленных компьютеров, соединенных между собой сетью.

Со временем большинство ОС стали *сетевыми,* т.е. начали поддерживать сетевое оборудование, протоколы, авторизацию, доступ к удаленным ресурсам (к принтерам, дискам) и пр. Такие ОС позволяют, в частности, обеспечить обмен сообщениями между процессами, запущенными как на одном, так и на нескольких процессорах. Появляется возможность распределять компоненты (процессы) программы по вычислителям сети в зависимости от загруженности последних. Гибкость данного подхода позволяет легко *масштабировать* подобные программы, т. е. наращивать пропускную способность и/или функциональность. При правильном проектировании перенос процесса с одного вычислителя на другой не требует доработки программы – достаточно изменить файлы конфигурации, чем обычно занимается *системный интегратор.* Программы, состоящие из взаимодействующих процессов, функционирующих на географически удаленных вычислителях, называют *распределенными приложениями.*

2.14. Удаленный вызов процедур

Программирование распределенных приложений представляло собой нетривиальную задачу, требующую высокой квалификации и значительных трудозатрат. Разработчикам приходилось использовать *сокеты* (низкоуровневый программный интерфейс

сетевого обмена) и самостоятельно реализовывать весь стек прикладных протоколов.

Проблема чрезвычайно усложнялась, если приходилось связывать программы, написанные на разных языках и работающие на разных аппаратных и программных платформах, объединенных разными коммуникациями *(гетерогенные системы)*. А это приходилось делать все чаще, из-за увеличения масштабов систем, их интеграции. Кроме того, все острее давала о себе знать проблема переносимости программ с одной платформы на другую.

Решение подобных проблем классическое – это введение *дополнительного уровня косвенности или промежуточного слоя* между прикладной программой и операционной системой, который будет скрывать различия нижележащих слоев[1].

Таким слоем вначале был механизм *удаленного вызова процедур* (RPC – Remote Procedure Call). Данный механизм позволяет вызывать процедуры на удаленном компьютере так, будто бы вызывается локальная процедура. При этом «незаметно» для пользователя осуществлялась упаковка параметров процедуры в сообщение *(сериализация, маршалинг),* передача сообщения на удаленный компьютер, распаковка сообщения *(десериализация, демаршалинг),* вызов локальной для удаленного компьютера процедуры, упаковка-передача-распаковка ответа вызывающему компьютеру.

Механизм удаленного вызова скрывает от пользователя способ упаковки/распаковки сообщений, способ и маршрут передачи данных, местоположение, платформу и способ реализации вызываемой удаленной процедуры (например, язык, на котором она написана). Говорят, что RPC делает перечисленные аспекты *прозрачными.* Переконфигурация сети, замена удаленных платформ и реализаций удаленных процедур отражается только в настройках RPC и никак не сказывается на прикладной программе.

На основе RPC строятся приложения с т. н. *клиент-серверной*

[1] В среде программистов распространена шутка, которая имеет значительную долю правды: «Любая проблема в программировании решается введением дополнительного уровня косвенности, кроме проблем слишком большого числа уровней».

архитектурой, получившей широчайшую популярность и распространение (подробнее об этой архитектуре будет упомянуто ниже).

2.15. Программное обеспечение промежуточного слоя

Первые реализации RPC были в основном одноплатформенными. Как правило, это были Unix-системы (Sun ONC, Apollo NCS, DCE), ориентированные на язык С и сетевые протоколы TCP/UDP. Первые реализации также были процедурно-ориентированные (не объектно-ориентированные). Последующие этапы развития RPC представляют собой попытки преодолеть указанные недостатки.

Так упоминавшаяся ранее объектно-ориентированная (точнее компонентно-ориентированная) технология COM была расширена до DCOM (Distributed COM). DCOM предоставляет возможность коммуникации распределенных компонент через сеть, но ограничивается только Windows-платформами.

Подобное компонентное решение RMI (Remote Method Invocation) и его расширение EJB (Enterprise Java Beans) разработано для платформы Java. Существенное преимущество по сравнению с DCOM (ориентированной только на Windows) заключается в том, что платформа Java может быть развернута поверх большинства современных ОС. Однако взаимодействующие компоненты должны быть реализованы по технологии Java, что является нежелательным ограничением.

Еще одним конкурирующим продуктом является спецификация CORBA (Common Object Request Broker Architecture – общая архитектура объектных запросов), разрабатывавшаяся консорциумом OMG (Object Management Group – рабочая группа по разработке объектно-ориентированных технологий и стандартов – группа представляющая интересы нескольких сотен известных компьютерных фирм). Данная спецификация представляет собой попытку разработать полностью унифицированный компонентно-ориентированный механизм взаимодействия программ в гетерогенной среде. На технологию полагались большие

надежды, вкладывались значительные инвестиции, привлекались ведущие специалисты. Однако по ряду причин, в первую очередь организационных [16], CORBA не получила предполагавшегося распространения. Основные претензии предъявляются к ее чрезмерной сложности и не предрасположенности для всемирной сети Интернет (в частности, не обеспечивающей достаточного уровня безопасности).

Рассмотренное программное обеспечение, обеспечивающее взаимодействие программ в гетерогенной среде, располагающееся поверх ОС и под прикладными программами, получило название *промежуточного слоя (middleware)*. Примеры промежуточного слоя (DCOM, RMI, CORBA) имеют много общего не только в постановке задачи, но и в способах ее решения. Эти решения представляют непосредственный интерес в смысле заимствования опыта решения сложных задач.

Во-первых, программное обеспечение промежуточного слоя выполнено именно в виде слоя. Это классический пример многоуровневой архитектуры, в которой промежуточный слой обеспечивает разделение труда разработчиков (разработчики верхнего уровня не вникают в детали реализации нижележащих слоев и наоборот). Другими словами middleware, разделяет технологии. Он же и повышает уровень абстракции задачи, скрывая от прикладных задач детали взаимодействия компонент в гетерогенной системе. Скрытие деталей нижележащих слоев позволяет менять их реализацию, не затрагивая прикладной уровень, т. е. обеспечивает гибкость системы.

Во-вторых, компоненты гетерогенной системы для своего взаимодействия должны, так или иначе, использовать единые правила, в частности, язык интерфейсов IDL (стандартизация). Язык интерфейсов должен иметь отражение на все языки программирования, т. е. должен быть некоторым общим знаменателем, в который входят совместимые элементарные и структурированные типы данных, правила вызова функций и пр.

В-третьих, в промежуточном слое в том или ином виде присутствуют общие службы (общесистемный ресурс) для регистрации

компонента в системе, публикации его интерфейса, поиска в сети компонентов с нужными интерфейсами, их активации, установления связи, управления доступом и пр.

В-четвертых, адреса компонентов должны отображать их прикладное назначение (их роль) и быть полностью абстрагированным от топологии и местоположения в сети. Поиск физического местоположения может осуществляться автоматически. Это дает возможность «незаметно» перемещать компоненты в сети (например, для обеспечения производительности), т. е. позволяют переконфигурировать сеть.

В-пятых, имена компонент (адреса) имеют иерархическую древовидную структуру (наподобие папок с файлами или адресов URL), позволяющую эффективно упорядочивать множество имен в глобальном сетевом пространстве, а также избегать конфликта имен.

В-шестых, взаимодействие с удаленным объектом (компонентом) реализуется с помощью локального посредника (брокера, заместителя, proxy, агента, заглушки). Такое решение соответствует паттерну «Заместитель», хорошо зарекомендовавшему себя в подобных задачах.

Трудоемкость разработки ПО промежуточного слоя демонстрирует тот факт, что крупнейшие ИТ-корпорации на протяжении десятка лет пытались (и пытаются) решить ее, но результат оставлял (и оставляет) ожидать лучшего. Трудоемкость велика, во-первых, из-за необходимости увязки множества стандартов и технологий. Во-вторых, из-за необходимости поддержки огромного массива созданного по устаревшим технологиям программного обеспечения, которое необходимо поддерживать (говорят, что приложения сопротивляются изменениям инфраструктуры и стандартов). В-третьих, из-за умышленной сегментации рынка конкурирующими компаниями разработчиков.

Приведенный пример сложности увязки разных технологий объясняет, почему опытные проектировщики настоятельно рекомендуют использовать в разрабатываемой системе минимальное количество стандартов и не отклоняться от принятой архитекту-

ры. Решение в рамках принятой архитектуры более предпочтительно даже в тех случаях, когда оно уступает по каким-то параметрам, например, по производительности.

По той же причине не рекомендуется начинять создаваемую систему функциональностью, в которой нет особой нужды, но которая, как это кажется, может понадобиться в будущем. В процессе разработки все равно возникнут непредвиденные проблемы, но чаще всего совсем не те, которые изначально предполагались.

Хотя вводить излишние стандарты и функции в систему и не рекомендуется, но архитектура должна предусматривать расширения для них, чтобы смягчить последствия их внедрения в случае настоятельной необходимости.

2.16. Кроссплатформенность

Из рассмотренных технологий промежуточного слоя: DCOM, CORBA, Java RMI только последняя продолжает развиваться. Остальные за редким исключением сконцентрированы на поддержке существующих приложений. В чем секрет такой стабильности Java? Скорее всего, ответ кроется в удачном решении проблемы *кроссплатформенности,* которая имеет особое значение для гетерогенных систем.

Традиционно кроссплатформенность обеспечивалась с помощью языков высокого уровня. Преобразование исходного кода программы в машинный для конкретного процессора обеспечивает соответствующий компилятор. В таких случаях говорят, что кроссплатформенность обеспечивается *на уровне компиляции.*

Большинство современных языков высокого уровня можно назвать кроссплатформенными, т. е. для них имеются компиляторы под разные платформы. Однако наиболее распространенным в этом смысле оказался язык C/C++. Трудно найти современный процессор, для которого не было бы компилятора C. Даже микроконтроллеры, у которых всего лишь несколько килобайт оперативной памяти, и те снабжаются компилятором C, иначе они будут неконкурентоспособными. Кроме того, подавляющее большинство операционных систем написаны на C/C++. Семейство

Unix-подобных ОС включает компилятор данного языка в свой состав. Не удивительно, что именно язык C/C++ чаще всего использовался для написания кросс-платформенных библиотек: ACE, boost, GTK+, Qt, wxWidgets, STL, OpenGL, OpenAL, BerkeleyDB, ImageMagick, TinyXML, OpenSSL, POCO, asio и множества других.

Но данный подход имеет и свои недостатки, которые ограничивают применение приложений в гетерогенных системах. Среди этих недостатков – компиляция программы (трансляция всей программы в машинные коды до запуска), производимая в ручном или полуавтоматическом режиме при *портировании* (переносе) программы с одной платформы на другую.

Больший уровень автоматизации обеспечивают *интерпретаторы,* которые последовательно транслируют отдельные инструкции программы и тут же их исполняют. Примеры интерпретируемых языков: Perl, PHP, Python, Ruby, Matlab, Mathematica, Maple, Mathcad. Интерпретатор представляет собой среду для исполнения интерпретируемых программ, расположенную поверх некоторой платформы (ОС). Если имеются интерпретаторы языка для разных платформ, то говорят о *кроссплатформенных средах исполнения*.

Современные интерпретируемые языки обладают сложными конструкциями высокого уровня абстракции (классы, шаблоны и пр.), которые требуют значительных затрат для трансляции (интерпретации). Сравнительно медленный процесс интерпретации (и соответственно исполнения) в ряде приложений оказывается неприемлемым.

Есть и другие способы обеспечения кроссплатформенности, например, эмуляция одной платформы поверх другой (DOS поверх Windows, Windows поверх Linux и т.п.). Хотя такие способы более громоздки и менее эффективны, в ряде случаев они могут оказаться наиболее подходящими.

2.17. Виртуальная машина

Проблем с переносимостью программ не существовало, если бы процессоры имели одинаковый универсальный набор инструкций. Однако, как уже говорилось, добиться этого на аппаратном

уровне не удалось из-за конкуренции фирм-производителей и прочих причин. Но такой процессор *(виртуальную машину)* вполне можно реализовать на программном уровне.

Имея в распоряжении виртуальную машину можно производить трансляцию в два этапа: сначала компилировать программу с языка высокого уровня в промежуточный машинный код виртуальной машины (т.н. *байт-код*[1]), после чего интерпретировать с байт-кода в машинные коды конкретного вычислителя. Разбивка процесса трансляции на два этапа приносит значительные преимущества.

Первый этап трансляции самый сложный в реализации и самый затратный по времени выполнения (из-за сложных абстракций исходного языка, см. выше). Второй этап существенно проще, поскольку байт-код существенно менее абстрактен, более компактен и приближен к машинному коду, поэтому интерпретаторы байт-кода (их же называют виртуальными машинами) гораздо проще и производительнее. Простота интерпретаторов байт-кода позволяет сравнительно легко и быстро разработать виртуальную машину для появившейся новой платформы.

Интересно, что компилятор первого этапа (из исходного кода в байт-код) написан на самом же компилируемом языке высокого уровня. Компилируется новая версия компилятора с помощью компилятора предыдущей версии (первая простая версия пишется на другом языке, например, на C). Такой рекурсивный метод разработки называется *bootstrapping*.

Таким образом, сложный компилятор первого этапа трансляции имеет единственную реализацию исходного кода[2], а не их множество (можно сказать: разрабатывается для единственной платформы). Кроссплатформенность обеспечивается написанием сравнительно небольших и несложных интерпретаторов байт-кода для разных платформ.

---

1 Байт-код называется так потому, что длина каждого кода операции – один байт. Код команды может иметь большую длину, т.к. кроме кода операции включает дополнительные параметры: регистры, адреса памяти.

2 Имеется ввиду реализация конкретной версии компилятора, например, текущей версии.

В гетерогенной системе применение байт-кода позволяет реализовать очень гибкие технологии. Например, откомпилированные в байт-код прикладные программы хранятся на сервере. При необходимости эти программы передаются по сети клиентам, где они выполняются (интерпретируются) на соответствующей виртуальной машине. Если на клиенте не установлена виртуальная машина, то она предварительно загружается по сети с сервера и устанавливается автоматически. Если добавляемый узел реализован на новой платформе, то достаточно зарегистрировать на сервере соответствующую виртуальную машину. Таким образом, удается полностью автоматизировать процесс адаптации новых узлов в гетерогенной системе.

Использование байт-кода позволяет элегантно решить не только проблему кроссплатформенности, но и проблему «многоязыковости». Достаточно, чтобы языки высокого уровня компилировались в байт-код. Полученные модули с байт-кодом компонуются в программу обычным способом независимо от того, на каких языках они были написаны изначально.

Упрощается также и задача разработчиков компиляторов языков высокого уровня. Теперь им не нужно разрабатывать компиляторы для каждой новой платформы, достаточно вести один проект для компиляции в байт-код. Кроссплатформенность обеспечивается виртуальными машинами.

Виртуальная машина или абстрактный процессор являются красивым и очень эффективным примером промежуточного (декомпозирующего) слоя, который позволяет не только увязывать между собой (и наращивать) разные технологии нижележащих подсистем, но и вышележащих.

Любопытно, что увязка производится на низком уровне абстракции – на уровне байт-кода, а не на языке высокого уровня (так же, как и в случае увязки модулей, откомпилированных в нативный машинный код, например DLL). Объяснение тому видится следующим. Байт-код, с одной стороны, существенно более прост, с другой стороны, менее нагляден и более громоздок из-за своих

низкоуровневых конструкций. Простота байт-кода позволяет легче автоматизировать работу с ним, в частности, разработать интерпретатор. В свою очередь для интерпретатора (автомата), наглядность и громоздкость программы особой роли не играют. По-сути, абстрагирование, декомпозиция и пр. приемы важны только для разработчика, т. е. человека с его ограниченными возможностями одновременного восприятия большого объема информации.

Основной недостаток виртуальной машины – снижение производительности за счет интерпретации кода во время выполнения программы. Это те накладные расходы, которыми приходится расплачиваться за множество преимуществ. В большинстве случаев эти расходы пренебрежимо малы. Кроме того, разработаны способы, существенно ускоряющие выполнение байт-кода. Например, некоторые виртуальные машины не интерпретируют байт-код, а компилируют его «на лету» (JIT-компиляция, Just In Time) в машинные коды процессора. При следующем запуске программа уже не будет компилироваться, а будет запущена ее ранее откомпилированная версия, сохраненная в кэше. Известны также примеры аппаратной реализации виртуальных машин (Jazzele, ObjectCore, picoJava и пр.). Тем не менее, есть приложения, где медленное выполнение байт-кода не допустимо. Чаще всего – это встраиваемые системы реального времени.

2.18. Технологии Java

Наиболее ярким применением виртуальной машины, можно сказать технологическим прорывом в области гетерогенных систем в конце1990-х, стали технологии Java. В настоящее время сообщество Java-разработчиков считается крупнейшим в мире (около 10 млн человек), платформы Java установлены более чем на миллиарде ПК, и более чем на 3 млрд мобильных телефонов и пр.

Java-технологии включают в себя множество компонентов, однако ключевой из них, обеспечивающий кроссплатформенность

и работу в гетерогенных сетях, – это виртуальная машина Java[1] (JVM – Java Virtual Machine), над которой надстраиваются другие инструменты.

Компания-разработчик Sun Microsystems (а сейчас – компания Oracle, которая поглотила Sun Microsystems), стремясь завоевать разные ниши рынка, специфицируют свой продукт в разных конфигурациях в соответствии с типовыми запросами пользователей.

Для обычных пользователей предоставляется среда исполнения Java (JRE – Java Runtime Environment), состоящая из виртуальной машины JVM и библиотеки стандартных классов Java. Среда исполнения необходима для выполнения Java-программ. JRE бесплатна и может загружаться через Интернет автоматически при необходимости.

Опытные разработчики могут бесплатно использовать инструментарий разработки JDK (Java Development Kit), включающий в себя кроме среды исполнения JRE, компилятор javac из объектно-ориентированного языка Java в байт-код, документацию, примеры и разного рода утилиты: отладчик и пр. Запуск утилит JDK производится с командной строки.

Для автоматизации процесса разработки существуют надстройки над JDK в виде интегрированных сред разработки с графическим интерфейсом: NetBeans IDE, Sun Java Studio Creator, Eclipse, Borland JBuilder и др.

Java-платформы обычно доступны в трех основных вариантах: Java Platform Standard Edition (Java SE), Java Platform Enterprise Edition (Java EE), Java Platform Micro Edition (Java ME). Java SE предназначена для приложений индивидуального пользования или для небольших предприятий. Java EE предоставляет более мощные средства для создания серверных платформ и коммерче-

[1] Впервые виртуальная машина и байт-код были использованы в языке Smalltalk, как и другие идеи, которые впоследствии завоевали популярность: шаблоны проектирования, рефакторинг, CRC-карты, экстремальное программирование, динамическая типизация, сборка мусора, JIT-компиляция и пр. Несмотря на то, что Smalltalk так и остался экспериментальным языком, его идеи внесли огромный вклад в развитие программной индустрии.

ских приложений для крупных предприятий. Java ME – подмножество платформы Java для устройств, ограниченных в ресурсах, например: сотовых телефонов, карманных компьютеров и пр.

Имеются и другие вариации Java-платформ, например, Java Card – платформа для устройств с крайне ограниченными вычислительными ресурсами (смарт-карт и т.п.). В виртуальной машине Java Card (JCVM – Java Card Virtual Machine) исключены потоки, операции с плавающей запятой и пр. Однако программы, ориентированные под JCVM, могут запускаться и на стандартной JVM, т.к. используемые в них инструкции являются подмножеством стандартного байт-кода.

Рынок Java-разработчиков расширяется благодаря возможности взаимодействия многих языков высокого уровня: Java, Groovy, Scala, JRuby (Ruby поверх JVM), Jython (Python поверх JVM), Clojure и др., которое обеспечивается виртуальной машиной Java.

Java построена на открытой культуре с высокой конкурентностью фирм. Для участия в процессе разработки заинтересованных фирм предусмотрен формальный процесс JCP (Java Community Process), согласно которому желающие стороны подают заявки с предложениями о нововведениях. После ряда формальных процедур, в случае, если последние пройдены успешно, выпускается спецификация нововведения. Сторонним разработчикам предоставляется возможность реализации нововведения согласно выработанной спецификации. Нередко им предлагается также и эталонная реализация. В результате такого процесса большинство Java-инструментов имеет множество конкурирующих реализаций, разработанных разными фирмами. Например, виртуальные машины Java реализованы десятками фирм. Таким образом, решается вопрос независимости пользователей Java от ее производителя.

Большинство Java-инструментов поставляются бесплатно, а также с открытым исходным кодом. Например, полностью открыты исходные коды JDK, включая Java-компилятор javac, сервер приложений GlassFish, интегрированные среды разработки NetBeans, Eclipse и т.д.

Большое внимание уделяется консалтингу. В частности, для платформы Java EE разработаны паттерны проектирования, описывающие архитектуру серверной платформы для задач средних и крупных предприятий.

Разумная стратегия развития/распространения и грамотные технические решения, заложенные в основе технологий, позволили Java удерживать лидирующие позиции в программной индустрии на протяжении длительного времени. В настоящее время Java не потеряла своей актуальности, хотя и выглядит несколько устаревшей по сравнению с новыми технологиями.

2.19 .NET Framework

Корпорация Microsoft, видя растущую популярность и эффективность Java-технологий, не могла оставаться в стороне, боясь утратить значительную часть рынка. Сначала Microsoft реализовала свою Java-машину MSJVM под Windows, начала реализовывать расширение J++ языка Java. Потом были судебные иски к Microsoft со стороны Sun по поводу нарушений лицензирования. В частности, Microsoft обвиняли в том, что J++ лишен свойств кроссплатформенности (например, из-за интеграции с DCOM). В конечном итоге Microsoft решила реализовать альтернативную платформу, которая получила название .NET Framework. С 2000-го года данная платформа объявлена, как основное направление разработок Microsoft.

Платформа .NET предназначалась для унификации всех программных продуктов Microsoft, которые к тому времени разрабатывались по разным технологиям. Планировалось кардинально изменить инфраструктуру приложений, провести своего рода рефакторинг. Структурные изменения подобного масштаба – непростая задача даже для такой крупной корпорации, как Microsoft. Решение было рискованным, но фирма сумела справиться с поставленной задачей.

Фундаментальные идеи .NET практически полностью повторяют идеи Java. Используются: аналог байт-кода – промежуточный

язык IL[1] (Intermediate Language); аналог JRE – среда исполнения CLR (Common Language Runtime); аналог объектно-ориентированного языка программирования Java – C# и т.д.

Платформы .NET и Java используют подобные технологии и являются очевидными конкурентами, поэтому их часто сравнивают друг с другом. В целом они близки, но подходы к их развитию у Microsoft и у Oracle (ранее Sun) существенно различаются.

Проектировщики Java занимают консервативную позицию по добавлению новых возможностей, не желая привязать свои технологии к сиюминутным течениям, которые в долгосрочной перспективе могут стать обузой. Внимание акцентируется на простоте и переносимости.

Разработчики .NET (в частности, языка C#), наоборот, спешат реализовать модные тенденции программирования в ущерб простоте, что усложняет переносимость.

В настоящий момент Java имеет несравнимо больше адаптаций под различные платформы и, соответственно доминирует в мобильном секторе, а также широко представлена на рынке веб-приложений (Java EE). В свою очередь, платформа .NET в основном используется для настольных приложений под Windows[2] и для веб-приложений (ASP.NET). Пожалуй, наиболее жесткое противостояние происходит в области веб-приложений.

Рассмотренный в данном пункте пример еще раз демонстрирует, как конкуренция порождает дублирование технологий, создавая дополнительные сложности пользователям. Конечно, конкуренция является стимулом для совершенствования разра-

---

[1] Байт-код оказался столь удачной абстракцией, что его в явном или неявном виде используют практически все современные интерпретируемые языки программирования: Perl, Ruby, Python, PHP. Многие компилируемые языки также могут компилироваться в байт-код.

[2] Хотя существуют реализации .NET для ОС, отличных от Windows (Mono, DotGNU), пока эти реализации не нашли широкого применения, кроме того, имеются юридические проблемы их распространения. Пользователи .NET оказываются тесно привязанными к платформе Windows. Такая ситуация, когда производитель стороннего ПО может диктовать покупателю практически любые условия на поддержку внедренного проекта, называется vendor-locking.

батываемого продукта, но вместе с пользой привносится много новых проблем. В рассмотренном случае появилась новая проблема – необходимость увязки приложений на платформе .NET с Java и CORBA.

2.20. Протокол SOAP

Итак, платформы на базе виртуальных машин стали ключевыми технологиями, обеспечивающими кроссплатформенность в гетерогенных системах. Однако проблему промежуточного программного обеспечения (middleware) они не решили. Не нашлось такой платформы, которая бы всех устроила в качестве единой базы. Думается, что некоторых влиятельных игроков на рынке не устраивала в этом смысле платформа чужой разработки. Вместе с тем индустрия (в частности, электронная коммерция) нуждалась в промежуточном обеспечении все сильнее и сильнее.

Microsoft, видя бесперспективность своей технологии DCOM, не собиралась оставлять конкурентам рынок электронной коммерции. Вместо развития DCOM Microsoft перевела противостояние в иную плоскость, выпустив в 1999 г спецификацию протокола SOAP (Simple Object Access Protocol). Если раньше технологии DCOM, CORBA и RMI конкурировали между собой, стремясь вытеснить одна другую, то SOAP представляет собой как бы связующий язык для их взаимодействия (рис. 6).

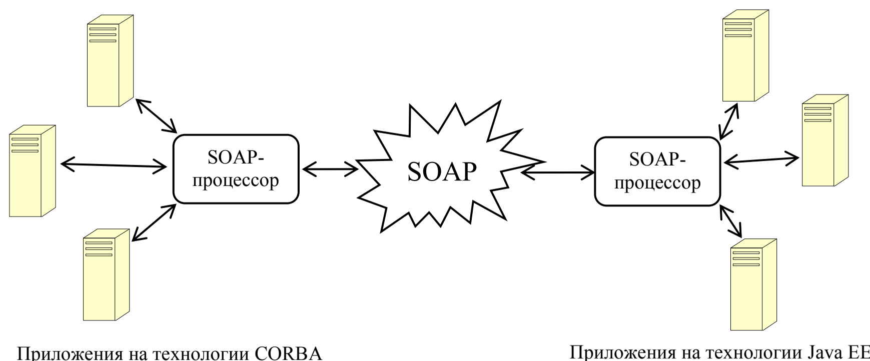


Рис.6. Взаимодействие систем посредством протокола SOAP

По своей сути протокол SOAP специфицирует формат сообщений между узлами распределенной гетерогенной системы. Большинство преимуществ SOAP вытекает из его ориентированности на Интернет. Это означает, что SOAP будет работать поверх хорошо зарекомендовавшей себя, глобально распространенной инфраструктуры. Кроме того, любой современный компьютер независимо от используемой платформы может иметь доступ во Всемирную паутину, поддерживает интернет-протоколы, располагает браузерами и пр. Такой компьютер уже обладает всем необходимым для использования SOAP.

SOAP использует текстовый формат сообщений с целью более простой совместимости разных платформ. Эта же идея с успехом используется в Интернет, в частности, в HTML (Hyper Text Markup Language – язык разметки гипертекста). Кроме того, SOAP реализован на базе расширяемого языка разметки XML (Extensible Markup Language), который приобрел большую популярность в конце 1990-х, а главное – XML рекомендован Консорциумом Всемирной паутины W3C (World Wide Web Consortium).

XML позволяет представлять данные в структурированном виде, например, в виде списков, деревьев (вложенных папок) и пр. Аналогичные структуры данных широко используются в современных языках программирования, поэтому легко трансформируются (отображаются) в XML и обратно. Ниже приводится пример простого SOAP-запроса стоимости некоторого продукта с идентификатором 75:

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd ="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi ="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Body>
    <GetPrice xmlns="http://www.somesite.com/">
      <ItemID xsi:type = "xsd:int">75 </ItemID>
      <Currency xsi:type = "xsd:string">USD</Currency>
    </GetPrice>
  </soap:Body>
</soap:Envelope>
```

Как показано в примере, SOAP-сообщение представляет собой стандартное обрамление (конверт, обвертку) с тегами Envelope/ Body, внутри которых передаются пользовательские данные (теги GetPrice, ItemID, Currency), тип и структура которых определяется конкретным приложением.

До появления SOAP распределенные приложения обменивались между собой данными через Интернет, упаковывая их в HTML-формы или в параметры URL (Uniform Resource Locator – унифицированный указатель информационного ресурса), например:

```
http://www.google.com/search?q=Baas+thesis&btnG=Google+Search
```

Поскольку HTML/URL не предоставляют механизмов для четкой структуризации и типизации данных, то семантику и значение данных приходилось извлекать, расшифровывая HTML/URL текст нестандартным способом. Если, например, сервер поменяет разметку HTML-страницы, то это непредсказуемым образом может сказаться на клиентах, расшифровывающих данную страницу. Предоставляя недостающие механизмы, протокол SOAP решает указанную проблему.

Еще одним преимуществом SOAP является его расширяемость, которая следует из расширяемости XML. Данное свойство позволяет посредникам (компьютерам через которые передается запрос) обрабатывать часть SOAP-сообщения или добавлять к нему дополнительные данные, что позволяет производить последовательную и асинхронную обработку сообщения несколькими серверами. Не меньшее значение имеет возможность наращивать в будущем протокол SOAP новыми конструкциями. Расширяемость также позволяет создавать шаблоны пользовательских сообщений с четкой структурой, типами данных и накладываемыми на них ограничениями (используя механизм т.н. XML-схем).

SOAP сообщения могут передаваться поверх любого протокола прикладного уровня: HTTP, FTP, SMTP и др. На практике чаще всего используется привязка к самому распространенному в Интернете транспортному протоколу HTTP. При этом не требуется

открытия дополнительных портов компьютера или предприятия (что когда-то считалось преимуществом).

Приведенный неполный перечень преимуществ SOAP способствовал тому, что данный протокол очень быстро был одобрен крупнейшими софтверными фирмами (в том числе и непримиримыми конкурентами) и рекомендован консорциумом W3C для использования в Интернет.

Протокол SOAP является типичным декомпозирующим слоем (уровнем косвенности), увязывающим несовместимые технологии.

SOAP имеет и свои недостатки. В частности, текстовый формат сообщений существенно увеличивает поток передаваемой информации (трафик) по сравнению с данными в двоичном представлении. По некоторым оценкам трафик увеличивается приблизительно на порядок.

2.21. Веб-службы

Протокол SOAP, обладая высоким уровнем абстракции и другими преимуществами, претендует на универсальный стандарт обмена между распределенными приложениями в гетерогенной сети. Соблюдая правила SOAP, зная сетевой адрес (URL) приложения и его конкретный формат запроса/ответа (интерфейс), клиенты могут обращаться к этому приложению, независимо от того, где они находятся, на какой платформе функционируют, какие технологии используют. Для случаев, когда клиентам неизвестны адреса и интерфейсы удаленных приложений, разработаны спецификации WSDL и UDDI.

Первый из них WSDL (Web Services Description Language – язык описания веб-сервисов) является подобием IDL (Interface Definition Language) в CORBA и COM и описывает формат сообщений, принимаемых/отправляемых сервисом, а также операции, которые могут выполняться с этими сообщениями. Документ WSDL схож с описанием библиотеки функций или классов в языках программи-

рования. Кроме описания функциональности в документе WSDL указываются т.н. привязки (к транспортному протоколу), описывающие способ доставки сообщений (например, поверх HTTP).

Созданный WSDL-документ необходимо опубликовать в некотором общедоступном реестре, где пользователи смогут осуществлять поиск интересующих сервисов и осуществлять их интеграцию в свои системы. Для этого предназначен инструмент UDDI (Universal Description Discovery & Integration). Взаимодействие приложений происходит следующим образом (см. рис. 7). Поставщик сервиса публикует его спецификацию в реестре. Клиент ищет в реестре интересующий сервис, найдя который, обращается непосредственно к поставщику с запросом. В целом правила взаимодействия очень похожи на соответствующие механизмы CORBA, EJB и др.

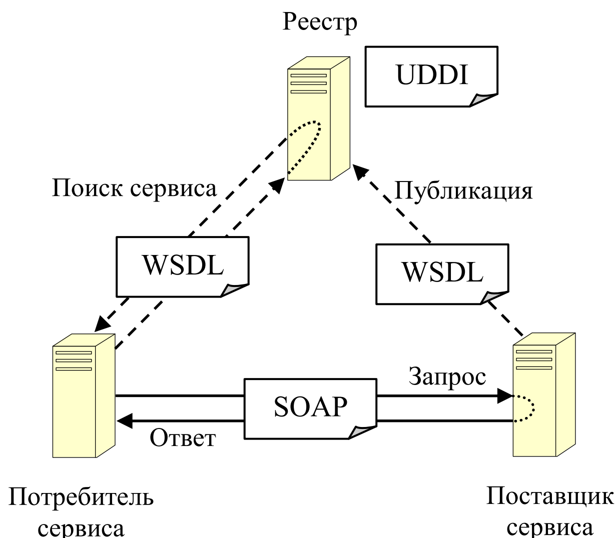


Рис.7. Взаимодействие компонент веб-службы

Используемые веб-сервисами спецификации представляют собой многослойную структуру (рис. 8).

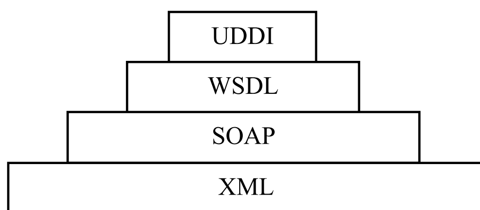


Рис.8. Многослойная организация спецификаций веб-службы

С появлением веб-служб инфраструктура Интернет стала все больше и больше использоваться для взаимодействия (автоматического) распределенных приложений, а не только для предоставления информации пользователям (людям) в «ручном» режиме.

2.22. Сервис-ориентированные архитектуры

Веб-службы являются одной из реализаций сервис-ориентированной архитектуры (SOA – Service Oriented Architecture). Данная архитектура предусматривает взаимодействие распределенных компонент по рассмотренной выше схеме. SOA открывает возможности построения распределенных приложений, состоящих из набора слабо связанных (loosely coupled) распределенных компонентов, обнаруживаемых в сети динамически (в отличие от монолитных распределенных приложений). В результате слабой связанности распределенные приложения лучше масштабируются, наращиваются, переконфигурируются и т.д.

Слабая связанность обеспечивается также за счет ряда ограничений данного архитектурного стиля. В частности, сервисы не могут хранить состояние и предысторию диалога с клиентом – вся необходимая информация, включая данные о предыстории, должны передаваться в сообщениях-запросах и, соответственно, храниться на стороне клиента (например, с помощью т.н. Cookies). Подобные ограничения существенно упрощают реализацию сервисов и горизонтальную масштабируемость приложений, но нередко не устраивают пользователей.

Недостающую функциональность веб-сервисов призваны обеспечить их надстройки, например, WS-BPEL (Web Services Business Process Execution Language) – язык на основе XML для формального описания бизнес-процессов и протоколов их взаимодействия. BPEL расширяет модель взаимодействия веб-служб, описывая последовательность взаимодействия нескольких сервисов, поддерживая транзакции[1] и пр. С использованием BPEL производят т.н. *оркестровку* – автоматическую координацию и управление сложными распределенными приложениями на базе сервисов. Разрабатываются и другие инструменты для расширения функциональности веб-сервисов, например, язык моделирования бизнес-процессов WS-CDL (Web Services Choreography Description Language), семейство спецификаций WSRF (Web Services Resource Framework) консорциума OASIS (Organization for the Advancement of Structured Information Standards): WS-Notification, WS-Addressing, WS-Transfer и пр. с собирательным названием WS-*.

Радужные представления начала 2000-х о том, что вот-вот будет создан единый всемирный реестр многократно повторно используемых веб-сервисов так и не реализовались. Бурная деятельность в этом направлении ведущих мировых корпораций привела к появлению большого числа спецификаций, что существенно усложнило технологию и оттолкнуло многих разработчиков. На фоне усложнения веб-служб вновь стал популярным традиционный для WWW архитектурный стиль REST (REpresentational State Transfer), напрямую использующий простой и широко распространенный протокол HTTP (REST также реализует сервис-ориентированную архитектуру, но более ориентированную на информацию, чем на операции). Тем не менее, веб-службы завоевали значительную нишу рынка, особенно в реализации сложных бизнес-процессов, хотя технологии веб-сервисов и представляются незрелыми.

По какой бы технологии не реализовывались SOA, давно стало очевидным преимущество самой сервис-ориентированной ар-

[1] Транза́кция – группа последовательных операций, которая может быть выполнена либо целиком и успешно, либо не выполнена вообще (если некоторые операции не выполнились, то для остальных производится «откат»).

хитектуры, навязывающей сервисы, как единицы модульности слабо связанной системы. Можно утверждать, что именно сервис-ориентированная архитектура WWW обеспечила его гибкость, масштабируемость и, как результат, успешность. Следует отметить, что сервисы успешно используются не только в информационных системах, но и в других технических системах, а также в коммерческих, социальных, биологических и пр. Сервисы можно рассматривать как особый вид общесистемных ресурсов.

Не успели еще устоятся технологии веб-служб, а индустрия уже вовсю предлагает новые виды сервисов – т.н. облачные вычисления (cloud computing): SaaS (Software as a Service – программное обеспечение как услуга), PaaS (Platform as a Service – платформа как услуга), IaaS (Infrastructure as a Service – инфраструктура как услуга), DaaS (Data as a Service – данные как услуга) и пр.

2.23. Серебряная пуля

Несмотря на появление новых инструментов и технологий, разработка программного обеспечения была и остается плохо предсказуемым и непрозрачным процессом. Тому есть несколько причин.

На первый взгляд, нижележащие слои (аппаратура, ОС, драйверы, компиляторы, фреймворки и пр.) абстрагируют программиста от множества проблем и рутины. Однако все преимущества абстрагирования исчезают, как только проявляется некоторая проблема в нижележащих слоях (что в эпоху рыночной конкуренции и спешки с выпуском новых версий продуктов становится не таким уж редким событием). В таких случаях невольно приходится изучать технологии нижележащих слоев, вникать в детали их реализации и взаимодействия, но даже это не всегда доступно, поскольку часто нижележащие слои представляют собой «черный ящик». Очевидно, что никто не может гарантировать отсутствие подобных проблем, не менее трудно спрогнозировать затраты на их решение. В подобных ситуациях особенно ценны специалисты, в совершенстве владеющие множеством технологий. Однако

встречаются они очень редко и, к сожалению, их чаще всего используют для поиска и устранения чужих ошибок. Для большей гарантии результата разработчики не редко используют пусть устаревший, но проверенный стек технологий.

Вторая причина заключается в том, что компоненты современных сложных систем осуществляют семантическое (смысловое) взаимодействие друг с другом на программном уровне. Проблемы доставки данных и пр. реализуются нижележащими слоями более-менее стандартно, но семантика сообщений, их смысл всегда уникальны. Другими словами на плечи программиста ложатся нестандартные проблемы увязки компонент большой системы. Эта задача – одна из самых сложных, поэтому плохо прогнозируема.

Индустрия также вставляет свои палки в колеса программистов, плодя новые и новые технологии (нередко тупиковые), поставляя их на рынок в незрелом виде, не стесняясь при этом рекламировать их, как очередной прорыв человечества.

Еще одна проблема заключается в том, что руководству и заказчикам трудно оценивать затраты на программирование, не видя физически осязаемого результата в виде некоторого прибора, устройства и т.п. Им нередко представляется удивительным, что коллектив трудился целый год, а результат их труда помещается на флешке или компакт-диске. Особенно эта проблема распространена в странах, где руководство предприятий формируется по указанию из вышестоящих инстанций. Руководство, не ощущая проблем программирования, не может самостоятельно планировать разработку и раздражается, когда узнает, что программисты и сами не могут указать четких сроков выпуска продукта (по совершенно объективным причинам). В результате, назначаемые директивно сроки нередко срываются, что служит причиной многих конфликтов.

Приведенный неполный перечень проблем специфичен именно для разработчиков программного обеспечения и делает их труд весьма незавидным. Многих программистов объединяет радужная мечта, что индустрия вот-вот разработает некоторый замечательный инструмент или технологию, которые в одночасье решат

большинство их проблем. Такое волшебное средство на жаргоне называют *серебряной пулей*[1].

Однако проходят десятилетия, а серебряная пуля так и не появляется. Разумеется, появляются новые средства, упрощающие работу программистов, делающие ее более эффективной, но таких средств, которые смогли бы на порядок повысить эффективность разработки программ (именно по этому параметру принято отличать серебряную пулю) в программировании не появляется. Такое положение дел Фредерик Брукс предвидел еще в 1986 году [15]. По его мнению, после появления языков высокого уровня технологии программирования стали настолько абстрагированы от компьютера и приближены к прикладным задачам, что сложность проекта определяется уже не столько тонкостями программной реализации, сколько самой прикладной проблемой. При этом не отрицается, что технологии совершенствуются и за десять лет могут на порядок повысить производительность разработок.

С данным мнением можно соглашаться или нет, но даже если серебряная пуля будет изобретена, можно утверждать, что жизнь разработчиков программного обеспечения не упростится. Просто перед ними встанут новые, более сложные проблемы.

[1] Волшебное средство – единственный вид пуль эффективный против нечистой силы (ведьм, вампиров, монстров).

3. ОРГАНИЗАЦИЯ ПРОЦЕССА РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

3.1. Эволюция процессов разработки программного обеспечения

Выше в общих чертах рассмотрены отдельные технологические приемы и инструментарий разработки программного обеспечения. Очевидно, что успех проекта определяют не только используемые инструменты, но и другие факторы, одним из которых является организация процесса разработки программного обеспечения.

Классической считается *каскадная модель* разработки или *модель водопада* [14], схема которой представлена на рис. 9.

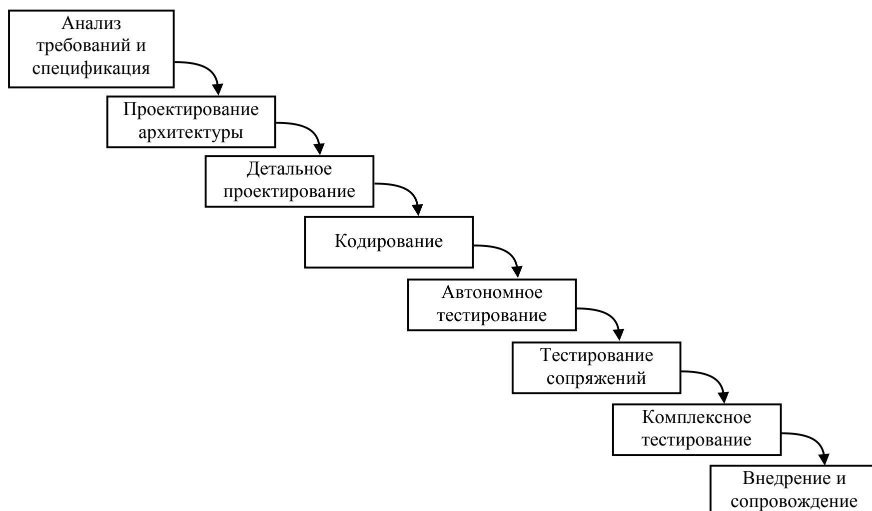


Рис.9. Каскадная модель процесса разработки программного обеспечения

Следуя каскадной модели, разработчик переходит от одной стадии к другой строго последовательно. На этапе анализа фор-

мируется список требований к программному продукту. Далее на этапе проектирования создаются документы, подробно описывающие способ и план реализации указанных требований. Разработанная проектная документация используется при реализации программного кода. Полученные модули продукта тестируются автономно (т.н. Unit-тестирование). Потом производится их интеграция: предварительно проверенные компоненты объединяются во все более сложные комбинации (тестирование сопряжений), которые тестируются до тех пор, пока не будет собрана вся система и не произведена ее комплексная проверка. Затем происходит сдача продукта заказчику, его внедрение и поддержка – мелкие доработки и устранение ошибок в процессе эксплуатации.

Разумеется, каскадная модель представляет собой идеализированный процесс разработки, который может служить лишь некоторым ориентиром. Тем не менее, его появление в 70х годах прошлого столетия стало значимым событием в программной индустрии и позволило существенно снизить риски и сделать разработку более прозрачной и прогнозируемой, что особенно ценно для крупномасштабных проектов.

Опытный разработчик без труда заметит недостатки каскадной модели, основной из которых, – отсутствие обратной связи между этапами разработки. Проверка принимаемых технических решений становится возможной только на поздних стадиях разработки каскадной модели, когда появится работающая система и, соответственно, возможность ее тестирования. Устранение выявленных недостатков, особенно ошибок и неточностей, допущенных на ранних стадиях проектирования (в требованиях, в спецификациях, в архитектуре), приводят к значительным расходам по их устранению вплоть до срыва проекта.

Такое положение дел не могло оставаться без изменений. Вскоре появились приемы, устраняющие недостатки каскадной модели. Один из таких приемов – *прототипирование (макетирование).* На ранней стадии создается действующий прототип создаваемой программы (ее упрощенный, черновой вариант), на котором проверяются принимаемые решения. Особенно полезны прототипы на этапе спецификации интерактивных систем, когда заказчик и

разработчик могут найти общее понимание проблем на базе прототипа пользовательского интерфейса. Во многих случаях взаимодействие с заказчиком (с пользователями) через прототипы дают разработчику ценнейшую информацию, которую получить другими способами очень трудно.

Прототипы могут использоваться не только на стадии анализа требований. В частности, использование прототипов на этапе проектирования архитектуры позволяет оценить возможности масштабирования системы, ее производительность, пропускную способность и пр. Т.е. те параметры, знание которых существенно снижает риски проекта, а их оценка другими способами весьма проблематична.

Прототипы могут использоваться только для проверки тех или иных свойств разрабатываемой системы, тогда они называются *вре́менными*. Другие прототипы могут лечь в основу будущей системы или ее отдельных компонент и при постепенном наращивании функциональности превратятся в окончательный продукт. Такие прототипы называют *эволюционирующими*, а соответствующий процесс разработки – *инкрементным*. На практике рассмотренные подходы могут комбинироваться, например, как показано на рис. 10.

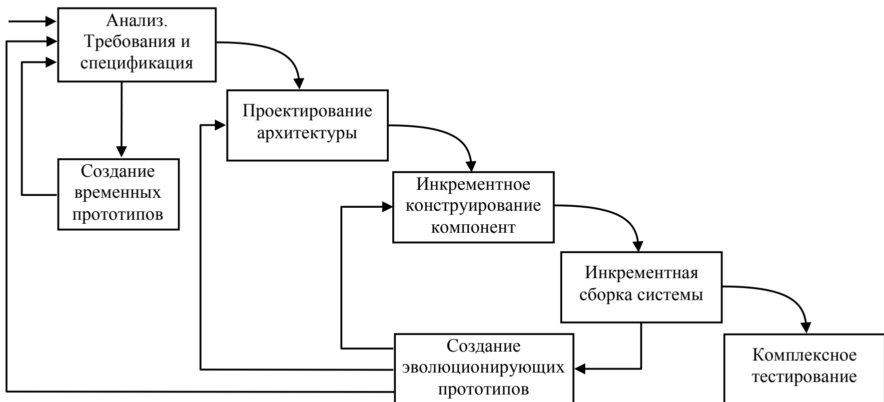


Рис.10. Жизненный цикл инкрементной разработки с временными прототипами

Кроме выявления ошибок на ранних стадиях, инкрементная разработка растягивает реализацию и тестирование на больший интервал времени, чем также снижаются риски. Важнейшим этапом считается запуск первой версии работающей программы, что позволяет приступить к отладке. Разработчики стараются осуществить запуск как можно раньше пусть даже в минимальной конфигурации. Зачастую минимальная конфигурация включает в себя только инфраструктуру или ее часть.

Существует множество вариаций процесса разработки программ. Наверное, у каждой фирмы он свой и, кроме того, изменяется от проекта к проекту. Это не снижает значимости унифицированных формализованных процессов разработки. Такие процессы разрабатывались и продолжают разрабатываться. Примерами могут быть: унифицированный процесс разработки программного обеспечения USDP (The Unified Software Development Process) [13], методология разработки программного обеспечения RUP (Rational Unified Process), созданная компанией Rational Software и др. Процесс проектирования формализуется не только для программного обеспечения, но и для других видов продуктов. Пример – свод знаний по управлению проектами PMBoK (Project Management Body of Knowledge) Института управления проектами PMI (Project Management Institute).

Громоздкие формализованные процессы проектирования обладают преимуществом в крупномасштабных проектах, где необходимо налаживать совместную работу множества организаций. Однако было замечено, что в проектах среднего и малого масштабов систематически более стабильные результаты проектирования демонстрировали небольшие группы разработчиков, использующих гибкие облегченные методики. Таких методик множество: т. н. экстремальное программирование, DSDM (Dynamic Systems Development Method), Scrum и др. Представители перечисленных направлений в 2001 году выпустили «Манифест гибкой методологии разработки программного обеспечения» (более кратко – Agile), в котором сформулировали следующие ключевые идеи:

– личности и их взаимодействия важнее, чем процессы и инструменты;

- работающее программное обеспечение важнее, чем полная документация;
- сотрудничество с заказчиком важнее, чем контрактные обязательства;
- реакция на изменения важнее, чем следование плану.

Манифест Agile выдвигает также 12 принципов разработки:

- удовлетворение клиента за счёт ранней и бесперебойной поставки ценного программного обеспечения;
- приветствие изменений требований даже в конце разработки (это может повысить конкурентоспособность полученного продукта);
- частая поставка рабочего программного обеспечения (каждый месяц или неделю или ещё чаще);
- тесное, ежедневное общение заказчика с разработчиками на протяжении всего проекта;
- проектом занимаются мотивированные личности, которые обеспечены нужными условиями работы, поддержкой и доверием;
- рекомендуемый метод передачи информации – личный разговор (лицом к лицу);
- работающее программное обеспечение – лучший измеритель прогресса;
- спонсоры, разработчики и пользователи должны иметь возможность поддерживать постоянный темп на неопределённый срок;
- постоянное внимание улучшению технического мастерства и удобному дизайну;
- простота – искусство не делать лишней работы;
- лучшие технические требования, дизайн и архитектура получаются у самоорганизованной команды;
- постоянная адаптация к изменяющимся обстоятельствам.

Можно заметить, что принципы гибкой разработки Agile находятся в определенном противопоставлении формализованным моделям типа водопада. Упор делается на личностные качества и заинтересованность разработчиков, их непосредственное взаимо-

действие, не обремененное формальностями и лишней документацией. Как правило, разработчики (оптимально около 10 человек) находятся в одном офисе, где также располагается и заказчик. Гибкие процессы разработки очень эффективны в ситуациях, когда крайне ограничены сроки и ресурсы (отсюда термин – экстремальное программирование).

Гибкие модели разработки имеют и свои недостатки. Это недисциплинированность и значительная зависимость от человеческого фактора. Кроме того, данный подход не стимулирует долгосрочного планирования, значительных затрат на разработку архитектуры, отдача от которых может произойти не так быстро. Считается, что Agile мотивирует разработчиков решать возникающие задачи самым быстрым и простым способом (принцип – «работает и ладно»), что в будущем может повлечь за собой частые и значительные переделки продукта, которые снижают его надежность.

Очевидно, что «золотая середина» для каждого проекта или организации находится где-то между двумя рассмотренными крайностями – громоздким формализованным процессом и экстремальным программированием. Небольшие команды разработчиков должны понимать значимость долгосрочного планирования и вкладывать ресурсы в архитектуру, а крупные корпорации стимулировать работу небольших самоорганизующихся коллективов, упрощать формальные преграды их существования и развития.

Можно наблюдать, что последняя тенденция завоевывает все большую популярность. Западные корпорации-гиганты, которые в свое время были неповоротливыми и монолитными, целенаправленно разбивают большие коллективы на небольшие самоорганизующиеся мобильные группы, способные эффективно решать оперативные задачи[1]. При этом из лучших технических

[1] Мобильным группам разрешается кооперировать свою работу не только с группами из той же корпорации, но наравне и с внешними организациями. Такой подход расширяет конкурентную среду за пределы предприятия и стимулирует команды в смысле качественного и эффективного исполнения своих задач.

специалистов формируются подразделения для стратегического планирования и определения основных направлений развития, на основе которых формируются задачи для мобильных подразделений. Таким образом, несмотря на раздробленность, сохраняется стратегическое единство организации. Кроме того, значительные финансовые возможности корпораций позволяют им концентрировать капиталовложения в ключевых направлениях и за счет этого добиваться рыночного преимущества над менее крупными компаниями-конкурентами.

3.2. Автоматизация разработки

Многие современные программные продукты насчитывают миллионы строчек кода, десятки тысяч файлов. Совершенствуются десятилетиями при участии сотен разработчиков. Код программ одновременно редактируется людьми, нередко находящимися в разных помещениях, в разных организациях, в разных странах. Вносимые изменения могут содержать ошибки или быть противоречивыми. Любое изменение потенциально может сделать стабильную прежде систему неработоспособной. Разобраться в этой путанице позволяют т.н. *системы контроля версий.*

Системы контроля версий хранят историю изменений документов (в частности, файлов исходного кода) в виде периодических фиксаций их содержимого, а также дополнительной информации о том, кто вносил изменения, когда и с какой целью. Автоматическое ведение реестра изменений само по себе полезно и позволяет в случае необходимости вспомнить и проанализировать интересующие этапы разработки, найти тех, кто внес те или иные изменения.

Гораздо важнее оказывается возможность «отката» проекта к любому ранее зафиксированному состоянию. Такая возможность особенно важна в том случае, когда внесенные изменения сделали разрабатываемую систему нестабильной или неработоспособной.

Кроме перечисленных возможностей, современные системы управления версиями предоставляют механизмы параллельной

разработки проекта. Например, когда к системе добавляется новая функция, то можно выделить группе ее разработчиков новую ветвь проекта, в которую они будут вносить изменения. В то же время остальные разработчики будут развивать основную ветвь независимо. После того, как новая функция будет реализована и отлажена, ее можно добавить в основной проект путем, т.н. слияния ветвей. Процесс слияния производится полуавтоматически. Система автоматически определяет все изменения содержимого файлов, и предлагает пользователю вариант объединения изменений. Противоречивые изменения, например, изменения одного и того же файла в разных ветвях, предлагается разрешить вручную. На разных этапах разработки количество параллельных веток проекта может существенно варьироваться, от десятков в начале разработки, до одной-двух ветвей на этапе сопровождения.

Примечательно, что объемы хранилищ истории увеличиваются не столь катастрофично, поскольку изменения файлов запоминаются в виде приращений, т. е. запоминается не вся новая версия файла, а только то его место, которое изменилось (например, строчка с исправленной ошибкой). Особенно эффективно в этом смысле хранение текстовых файлов, а не двоичных.

Изначально системы контроля версий были *централизованными* (CVS, Subversion и пр.), т.е. хранилища истории располагались на сервере, а пользователи копировали из него временной срез (фиксацию, commit) проекта на локальные компьютеры, модифицировали файлы и фиксировали изменения в хранилище сервера.

Со временем все более популярными становятся т.н. *распределенные* системы контроля версий (Git, Mercurial, Bazaar и др.). Такие системы на каждом локальном компьютере хранят не отдельный временной срез проекта, а всю его историю целиком. Другими словами, каждое локальное хранилище содержит полный объем информации и является равноправным по отношению к остальным хранилищам. Такие системы менее уязвимы и в определенном смысле более удобны. При этом вполне допустимо хранить одно из таких хранилищ на сервере и условно называть его центральным (так чаще всего и делают). Не все ветви разработки

обязаны вноситься в центральное хранилище. Например, экспериментальные ветви м. б. закрыты и оставлены на локальных компьютерах, чтобы не засорять основной проект.

Как правило, системы контроля версий управляются с командной строки, но для них имеется множество графических оболочек (Tortoise, QGit, SmartSVN, Giggle и др.). Многие системы контроля версий интегрированы в среды разработки (NetBeans IDE, Eclipse, Microsoft Visual Studio, Qt Creator и пр.). Пользуются популярностью сервисы, предоставляющие возможность расположения хранилищ в Интернете (GitHub, Bitbucket, SourceForge, Google Code, Launchpad и др.). Множество известных проектов пользуются подобными услугами (Linux, Android, Ruby on Rails, Qt, Joomla!, PHP, Mozilla, OpenOffice, Netbeans, OpenSolaris, Facebook, Yahoo, Twitter, Perl и пр.). Исходные коды большинства проектов открыты. Любой пользователь Интернета имеет возможность получить не только полную версию текстов программ со всей историей, но и стать полноправным участником разработки.

Системы контроля версий нередко интегрируются с системами управления проектами (Redmine, Trac, Bugzilla и пр.). Такие системы позволяют планировать и отслеживать разработку, назначая задачи, приоритеты, ответственных исполнителей и т.п.

Рассмотренные и многие другие инструменты автоматизации разработок (средства автоматической сборки проектов, прогонки тестовых примеров и пр.) реализуют, на первый взгляд, тривиальные возможности, однако подобные мелочи играют колоссальную роль в больших проектах.

3.3. Когда базар строит собор[1]

Ниже приводится два примера разной организации разработки проектов с участием большого числа разработчиков. Первый пример основан на статье «Восход и закат CORBA» Мичи Хеннинга

[1] Фраза взята из одноименной статьи сборника [4]. В свою очередь название статьи навеяно известным эссе «Собор и Базар» (The Cathedral and the Bazaar) Эрика Рэймонда на тему открытого процесса разработки ядра Linux.

[16], в которой раскрываются глубинные причины провала проекта CORBA – одного из наиболее финансируемых и коммерчески востребованных проектов своего времени. Во втором примере рассматривается процесс разработки интегрированной настольной системы KDE – одного из самых крупных мировых проектов с открытым программным кодом. Пример основан на статье Тиля Адама и Мирко Бёма из книги [4].

Пример процесса разработки спецификаций CORBA

Как упоминалось, разработка чрезвычайно востребованного для электронной коммерции стандарта промежуточного программного обеспечения производилась совместными усилиями нескольких сотен ведущих компьютерных фирм в рамках консорциума OMG. Ожидалось, что заинтересованность потребителей, участие в разработке ведущих фирм и значительные инвестиции обеспечат технологии CORBA безоблачное будущее. Однако произошло нечто противоположное – технология не смогла оправдать возложенных на нее надежд и заняла весьма узкую нишу на рынке (в основном – это встраиваемые системы).

Хеннинг отмечает внешние причины упадка CORBA, но основная причина – в ее техническом несовершенстве и, в первую очередь, – в ее сложности. Как же могло произойти так, что крупнейший в мире софтверный консорциум создал технически несовершенную технологию? Причину этого Хеннинг видит в организации совместной работы специалистов.

OMG публикует спецификации на основе голосования. Демократичность и справедливость процесса вместо ожидаемых дивидендов на практике сослужила плохую службу. Некоторые отрицательные проявления данного процесса перечисляются ниже.

1. Многие из участников разработки не являются экспертами в вопросах, за которые они голосуют, что постоянно приводит к принятию спецификаций, обладающих серьезными техническими дефектами.
2. Потребители, участвующие в разработке наравне с поставщиками, часто выдвигают требования, которые на практике труд-

но реализуемы. Нередко предлагаемая ими функциональность не столь принципиальна, но ее реализация противоречит принятой архитектуре системы и, поэтому оказывается очень громоздкой и запутанной.

3. Поставщики соглашаются на предложения потребителей даже в тех случаях, когда им известны технические дефекты. Делается это для получения благосклонности потребителей (и, возможно, контрактов), поскольку поставщики конкурируют между собой за потребителей.
4. У поставщика имеется конфликт интересов по отношению к стандартизации. С одной стороны, стандартизация привлекательна, поскольку упрощает торговлю технологией. С другой стороны, поставщики избегают слишком строгой стандартизации, чтобы не раскрывать свои решения, которые дают преимущества над конкурентами. Поставщики, стремясь удержать рынок сбыта, всячески пытаются привязать потребителя именно к своему продукту. Для этого многие компоненты, которые следовало бы стандартизовать, поставщики не раскрывают, пытаются блокировать их стандартизацию, или сделать эту стандартизацию слишком «туманной» и бесполезной. Похожее противодействие стандартизации возникает также в случаях, когда предлагаемая спецификация требует изменений в существующих продуктах поставщиков и т. п.
5. При рассмотрении конкурентных вариантов спецификации, предлагаемых разными участниками, OMG пытается объединить все возможности в единую спецификацию вместо того, чтобы выбрать один из вариантов. В итоге спецификация превращается в «кухонную раковину», в которую сливаются все предложения, приходившие кому-либо и когда-либо в голову. Эта практика является основной причиной сложности CORBA. Различные решения, совершенно разумные по отдельности, могут противоречить одно другому и приводить к семантическим конфликтам.
6. В OMG для принятия спецификации не требуется наличие эталонной реализации. Нередко спецификации, рожденные толь-

ко в обсуждениях и на бумаге, содержат в себе «подводные камни», которые выявляются только на этапе реализации. Подобные спецификации могут оказаться либо труднореализуемыми, либо экономически непривлекательными.

Перечисленные проблемы характерны для т. н. *разработки комитетом,* когда в принятии ключевых решений равноправно участвует множество сторон без единого видения проблемы или с противоречивыми интересами. В подобных ситуациях проекту зачастую характерны излишняя сложность, неполнота, логические противоречия и отсутствие целостной структуры[1].

Как отмечает Хеннинг, для совместной разработки ключевое значение имеют *сотрудничество и доверие.* Без них: «Наивно созывать конкурирующих поставщиков и потребителей в консорциум и ожидать, что они совместно смогут создать высококачественный продукт – коммерческие реалии неизбежно приводят к тому, что в умах участников консорциума сотрудничество и доверие находятся на самом последнем месте».

Благодаря сотрудничеству и доверию эффективность совместной разработки у проектов с открытым программным кодом (open source) существенно выше. Тем не менее, при производстве технологий индустрия предпочитает полагаться на крупные консорциумы. Эффективность консорциумов можно несколько улучшить, если соблюдать ряд формальных требований, например, не принимать стандартов без предварительной их апробации на нескольких реальных проектах разумной сложности и пр. Однако формальными требованиями невозможно заставить конкурирующих разработчиков доверять друг другу.

Примечательно, что после своего ухода из консорциума OMG, где он был членом комитета по архитектуре CORBA, Мичи Хеннинг стал руководителем исследовательских работ в перспективном проекте с открытым кодом Ice (Internet Communications Engine). Ice известно, как эффективное и масштабируемое ПО промежуточного слоя, намного меньше и проще, чем CORBA.

---

1 Такое часто происходит, когда система разрабатывается без единого системного архитектора.

Пример организации процесса разработки KDE

KDE (K Desktop Environment) – один из крупнейших мировых проектов с открытым программным кодом. Цель KDE – создание интегрированной настольной графической среды для операционных систем (изначально Unix, GNU/Linux, потом Windows, Mac OS X и встроенных платформ). KDE включает в себя множество взаимодействующих программ: веб-браузер, файл-менеджер, пакет офисных приложений, интегрированную среду разработки, графический и видео редакторы, утилиты для работы с мультимедиа, органайзер и пр. Компоненты KDE располагаются поверх единой инфраструктуры, включающей в себя слой взаимодействия с оборудованием Solid, слой обработки мультимедиа Phonon, слой динамичного масштабируемого пользовательского интерфейса Plasma и др. Очевидно, что функциональность KDE существенно шире, чем функциональность рассматривавшейся спецификации CORBA. По крайней мере, CORBA использовалась в KDE в качестве одного из компонентов инфраструктуры, но со временем была заменена более легковесным протоколом D-Bus.

Представляется удивительным, но вся перечисленная функциональность создана практически на инициативной основе без централизованного финансирования и управления. В проекте на равноправной основе участвует большое количество разработчиков разного возраста, с разным цветом кожи, из разных, порой враждующих стран, с разных континентов. В проекте нет единого главного архитектора, нет специальной группы стратегического планирования – все инициативы исходят от самих же разработчиков. Действительно получается, что «базар строит собор». Но как удается находить компромиссные решения при таком количестве разных и талантливых участников, каждый из которых склонен считать себя архитектором (как известно, талантливые люди часто непримиримы во мнениях)?

Разумеется, разногласия во мнениях существуют, обсуждения и споры ведутся непрерывно, а зачастую и ожесточенно. Но есть

нечто общее, что позволяет участникам принимать компромиссные решения. Это стремление создать технически совершенный, элегантный и передовой продукт. Сделать что-то действительно полезное вместо никому не нужных курсовых работ и рефератов. Ради достижения этой цели участники готовы жертвовать личным временем, идти на уступки и т. п. Как ни странно, это делается не ради почестей, положительных отзывов пользователей или публикаций в СМИ, а для какого-то внутреннего удовлетворения.

Когда выше говорилось, что у KDE нет централизованного управления, то имелось ввиду техническое управление, но не координационное. В самом деле, организация работ в KDE считается чуть ли не главным достижением данного проекта. Структура рождалась в продолжение многих итераций и представляет собой сообщество разработчиков, своеобразную соревновательную экосистему с минимально возможным бюрократическим аппаратом (изначально родившимся для решения неизбежных вопросов лицензирования и пр.). Основная цель, как ее видят организаторы, заключается в том, чтобы обеспечить разработчиков возможностью творить технически совершенный, элегантный и передовой продукт, как об этом упоминалось выше. Для достижения указанной цели в KDE используют ряд принципов, некоторые из которых перечисляются ниже.

Сообщества свободного ПО, в том числе и KDE, в первую очередь являются социальными, а не техническими. Самым ценным и редким ресурсом считаются люди. Соответственно, любые предпринимаемые шаги делаются с оглядкой на этот тезис. Например, вопрос о сплоченности команды считается не менее значимым, чем технические вопросы.

Одно из важнейших правил (если не самое важное) заключается в том, что направление развития продукта определяет не руководящая команда, а инициативные разработчики. Они назначают приоритеты, руководствуясь своей внутренней мотивацией. Несмотря на кажущийся субъективизм, принимаемые решения оказываются очень эффективными и практически всегда реализуются

успешно. Бюрократическая часть коллектива занимается только поддержкой технических разработчиков, сбором пожертвований, организацией конференции и пр.

Особой ценностью KDE считается независимость разработчиков при принятии технических решений. Многие успешные проекты с открытым программным кодом распадались, когда попадали под заметное влияние спонсоров. Как правило, из подобных проектов формируется коммерческая компания, спонсор берет под свой контроль основных участников, а сам проект перестает стремиться к новизне и переходит в режим сопровождения. Разработчик коммерческой компании всегда чувствует над собой прессинг сроков выполнения задач, все время вынужден оправдывать капиталовложения. Жесткие сроки завершения очередной версии продукта практически всегда негативно сказываются на его качестве. Необходимость регулярно идти на компромисс между сроками и качеством угнетающе действуют на большинство разработчиков. Чтобы не попасть под такую зависимость, в KDE отказываются от значительных пожертвований спонсоров (что у многих вызывает, мягко выражаясь, недоумение).

С другой стороны, если не назначать никаких сроков, то разработка превращается в долгострой, который не менее негативно сказывается на его участниках. Успешное завершение очередного этапа разработки (выпуска новой версии продукта) очень важны в психологическом плане. Кроме того, разумные сроки разработки поддерживают разработчиков в тонусе, не дают расслабиться. Понимая это, сообщество KDE добровольно назначило для себя срок выпуска очередной версии продукта – раз в пол года. Однако, временные рамки не столь строги, как в коммерции. Например, если причина срыва сроков – принципиальные архитектурные вопросы, то для их решения сроки могут быть отодвинуты без особых последствий.

Как и у любого проекта с открытым программным кодом, ключевой метод обеспечения качества программного продукта заключается в многократном анализе и тестировании открытого

исходного кода программ многими специалистами. Участник открытого проекта всегда осознает, что небрежно написанная им часть программы будет неизбежно переписана другим участником. В результате такого естественного (Дарвинского) отбора выживают только лучшие идеи и их реализации.

Координация работ производится в основном через Интернет, но необходимы также и непосредственные встречи разработчиков. Для этого KDE организует ежегодную всемирную конференцию, куда приглашаются все разработчики и желающие. Кроме того, отдельные подгруппы организуют локальные встречи, например, раз в месяц или чаще.

Странное с деловой точки зрения, буйное и плохо управляемое сообщество KDE, развивает очень сложный продукт, который под силу не каждой достаточно крупной коммерческой компании. При этом финансовые затраты KDE несоизмеримо меньше, а качество продукта выше.

3.4. Какая организация процесса разработки лучше

Рассмотренные два примера организации разработок в проектах CORBA и KDE схожи в том, что привлекают большое количество участников для решения весьма сложных задач. Почти во всем остальном данные проекты являются противоположностями друг друга. Можно считать их двумя крайностями среди бесчисленного множества вариантов организации разработок. Например, очень эффективен вариант открытого проекта с т. н. *благожелательной диктатурой,* когда альтернативные идеи и варианты их реализации предлагаются множеством разработчиков, но окончательный вариант определяется одним человеком или небольшой группой единомышленников.

Из рассмотренных усеченных примеров можно сделать неправильный вывод, что проекты с открытым кодом являются панацеей на все случаи. Это далеко не так. Хотя качество программного кода и малозатратность разработок являются отличительной чер-

той проектов с открытым программным кодом, в целом они занимают скромную нишу на рынке. Качество и малозатратность – не достаточные условия успеха.

При возникновении новых идей часто создаются небольшие коммерческие предприятия или открытые проекты, которые реализуют эти идеи и выходят с ними на рынок. Если идеи оказываются эффективными, то появляются их альтернативные реализации других фирм, которые стараются завоевать часть рынка. Множество реализаций порождает путаницу, осложняет выбор пользователям. Однако со временем небольшие команды, создавшие удачные реализации, поглощаются крупными корпорациями. В частности, такой путь прошли множество открытых проектов. Далее происходит интеграция различных реализаций, которая часто сопровождается монополизацией. Открытые продукты коммерциализируются, становятся *проприетарными* (частными, запатентованными), что, к сожалению, негативно сказывается на их качестве, развитии и стоимости. Разумеется, есть и исключения (например, KDE), но они не меняют картину в целом. Крупные корпорации все более монополизируют рынок и привязывают пользователя к своим технологиям.

Очевидно, что крупные монополии обладают дополнительными рычагами влияния: финансовыми, инфраструктурными и пр. Часто именно эти рычаги, а не техническое совершенство определяют успех проектов. Тем не менее, корпорации, видя эффективность открытых проектов и гибких процессов проектирования, пытаются каким-то образом адаптировать их к своим разработкам. Тем более, следует обратить внимание на опыт открытых проектов и гибких методик небольшим фирмам, пытающимся разработать продукт в условиях недостатка ресурсов.

4. ПРИМЕРЫ ОРГАНИЗАЦИИ СИСТЕМ

4.1. Эволюция информационных систем предприятия

Выше в общих чертах рассмотрены технологии, инструменты и организация разработки распределенных систем, которые хотя и навязывают определенную архитектуру, тем не менее, оставляют разработчику широкую свободу выбора. Любопытные изменения претерпели тенденции построения наиболее распространенных из распределенных архитектур, а именно, – клиент-серверные. В данном пункте акцент будет делаться на системах предприятий или корпоративных системах.

На заре компьютерной эры, когда вычислительные машины занимали целые здания, взаимодействующих прикладных программ практически не существовало. Перфокарты с программами ставились в очередь и, по мере исполнения предыдущей, из очереди автоматически загружалась следующая программа, которой вычислительная машина предоставлялась в монопольное использование (т.н. пакетный режим). По сути, вычислительная машина являлась общим ресурсом, который последовательно предоставлялся прикладным программам.

Чуть позже появилась возможность «одновременной» работы нескольких пользователей с вычислительной машиной через терминалы или консоли (рис. 11). В действительности программы продолжали выполняться по-очереди, но в это время несколько пользователей могли набирать тексты программ и пр. (операционная система VM – Virtual Machine или ее советский аналог СВМ – Среда Виртуальных Машин). Это не меняет сути – вычислительная машина продолжала оставаться общим ресурсом, предоставляемым по-очереди нескольким независимым программам.

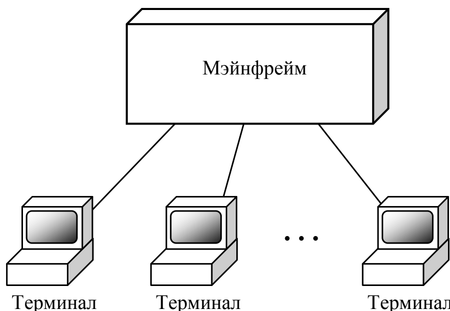


Рис. 11. Организация рабочих мест для больших вычислительных машин

Ситуация кардинально изменилась с появлением персональных компьютеров. Их количество стало неуклонно расти, в результате все больше проявлялась необходимость обмена информацией между компьютерами. Одним из первых видов межкомпьютерного взаимодействия были одноранговые или пиринговые сети (от англ. peer-to-peer – равный к равному). Каждый компьютер такой сети (узел, пир) равноправен остальным. Он предоставляет для всеобщего использования свои ресурсы и в то же время может потреблять ресурсы других узлов, т. е. является одновременно клиентом и сервером. Одноранговые локальные сети часто используют в офисах и небольших организациях для общего использования файлов, принтеров и пр. В глобальном масштабе наиболее распространены файлообменные пиринговые сети: ED2K, FidoNet, BitTorrent и др. На каждом компьютере одноранговой сети устанавливается полной комплект программ для администрирования и управления сетью.

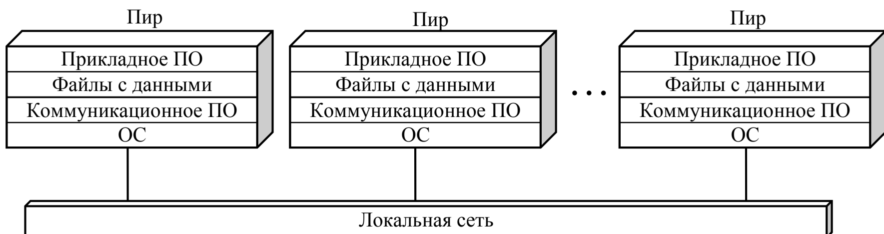


Рис. 12. Пиринговая сеть

Управлять информацией в виде файлов, разбросанных по узлам одноранговой сети, достаточно трудно, учитывая то обстоятельство, что любой пользователь компьютера может произвольно изменять файлы и директории (по крайней мере, локальные). Ситуация существенно упрощается, если общие файлы поместить на одном компьютере (файл-сервере, см. рис. 13). При централизованном расположении данных гораздо проще следить за ними, ограничивать доступ и т.п.

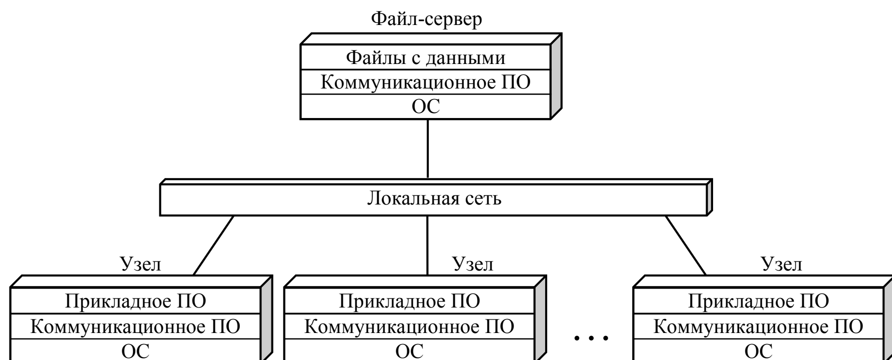


Рис. 13. Централизованное хранение информации на файл-сервере

Группировка файлов по директориям, хоть и является мощным инструментом упорядочивания данных, все же недостаточна для множества приложений. Она не позволяет строго структурировать внутреннее содержание файлов, а также устанавливать взаимосвязи между данными в разных файлах. Указанную проблему решили базы данных (БД). Например, в реляционных базах данные представляются в виде строго структурированных взаимосвязанных таблиц. Для формирования таблиц, поиска/изменения/добавления данных разработаны специальные программы – системы управления базами данных (СУБД), а также специальные языки для доступа к СУБД из внешних программ такие, как язык структурированных запросов SQL (Structured Query Language).

Изначально СУБД предназначались для работы с данными на локальном компьютере (Access, Paradox, FoxPro). Эти же СУБД

можно использовать и в сети. При этом данные располагаются на файл-сервере, а СУБД – на компьютерах пользователей (рис. 14). Файлы с данными целиком копируются с файл-сервера на локальный компьютер, а после модификации записываются обратно на файл-сервер. Одновременный доступ к данным обеспечивается с помощью обычных блокировок файлов на чтение/запись, предоставляемых ОС.

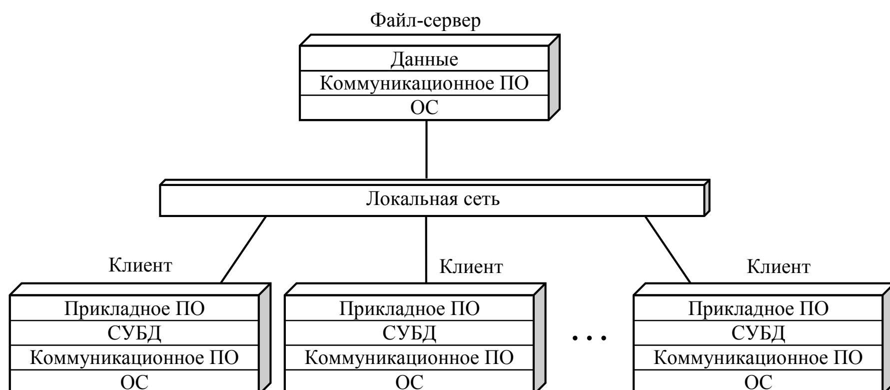


Рис. 14. Доступ к централизованным данным посредством локальных СУБД

Хотя файл-серверные СУБД и снижают требования к аппаратному и программному обеспечению сервера, но существенно нагружают сеть, т. к. передают все данные целиком, а не только те, которые интересуют пользователя. Такая организация баз данных для крупных систем считается устаревшей. На смену пришла т. н. *клиент-серверная*[1] организация, когда СУБД располагается на том же сервере, что и сама база данных, и монопольно владеет

[1] Клиент-серверным взаимодействием связано между собой большое число (если не большинство) программных компонент. По своей сути «клиент» и «сервер» – это роли, которые закреплены за компонентами. Сервер предоставляет свои услуги во всеобщее пользование. Клиенты обращаются к серверу за услугами, тот обрабатывает их в порядке очереди. С этой точки зрения сервер не обязательно должен находиться на другом компьютере и т.п. Однако термин «клиент-сервер» закрепился именно за сетевыми системами.

ею (примеры таких СУБД: Oracle, Interbase, DB2, MySQL, MS SQL Server и др., см. рис. 15). Централизованное управление данными существенно снижает загрузку сети, облегчает обеспечение надежности, безопасности и пр.

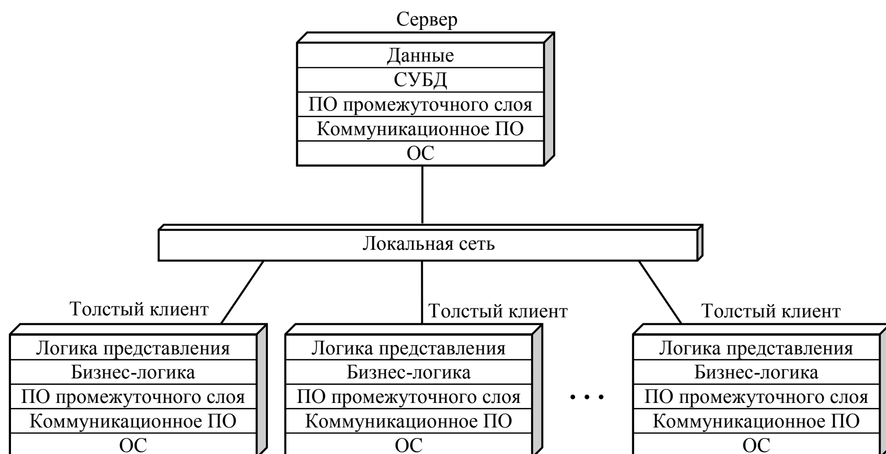


Рис. 15. Клиент-серверная организация системы

Совмещение большого количества данных (БД) и алгоритмов, их обрабатывающих (СУБД), на сервере не случайно. Эффективная обработка большого количества информации затруднительна, если алгоритмы расположены «далеко» от данных. Причиной тому медленная, неустойчивая (и гетерогенная!) связь между узлами сети. Совмещение БД и СУБД на сервере решает данную проблему. При обмене СУБД с внешним миром (с удаленными клиентами) поток информации (в виде SQL-запросов и ответов) существенно ниже и требования к его устойчивости менее жесткие.

СУБД реализует только типовые операции с данными: поиск, добавление, обновление и т. п. Для каждого конкретного приложения существуют свои специфические алгоритмы обработки данных, называемые *бизнес-логикой:* правила расчета зарплаты, сбор специфической статистики в виде отчетов и пр. Эта логика изначально реализовалась на стороне клиентов. Там же реализо-

вался интерфейс пользователя или *логика представления*. Такие клиенты называются *толстыми*.

Реализация бизнес-логики на стороне клиента не всегда эффективна. Например, для подсчета специфической статистики приходилось передавать с сервера на клиент весь массив данных, где те будут обрабатываться. После обработки пользователю предоставляется полученная статистика в виде, например, единственного числа. Операция была бы гораздо эффективней, если бы расчет производился на стороне сервера, а по сети передавался только окончательный результат. Для этой цели в современных СУБД предусматриваются т.н. *хранимые процедуры* – пользовательские исполняемые модули, которые хранятся на сервере, имеют доступ к данным и могут вызываться с удаленного клиента. Таким образом, хранимые процедуры позволяли перенести часть бизнес-логики с клиентов на сервер (рис. 16).

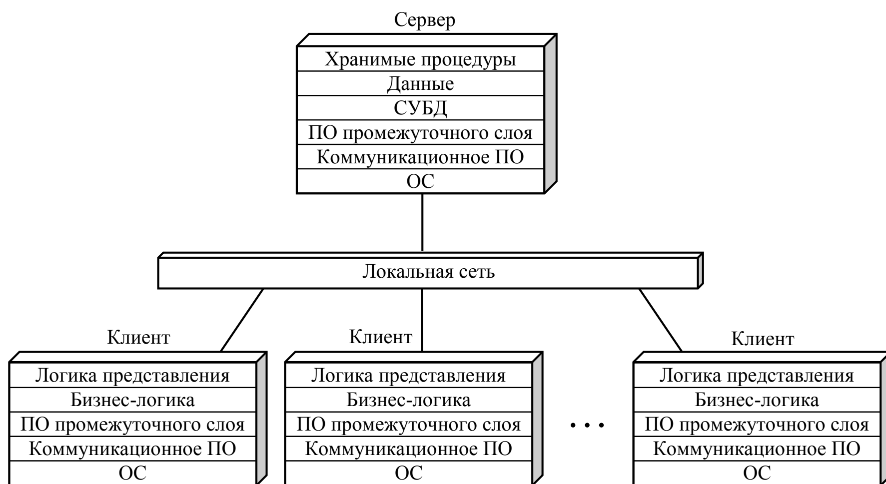


Рис. 16. Перенос на сервер части функциональности в виде хранимых процедур

В предельном случае вся бизнес-логика переносится с клиента на сервер, а на клиенте остается только логика представления (рис. 17). Подобные клиенты называют тонкими. Системы с тон-

кими клиентами оказались очень удобными в смысле простоты их модификации. Изменения в структуре данных (за которой автоматически следуют изменения в бизнес-логике) или в бизнес-логике производятся на сервере централизовано. Зачастую нет необходимости обновлять программное обеспечение на клиентах (которых может быть множество, они м.б. значительно удалены или вообще недоступны).

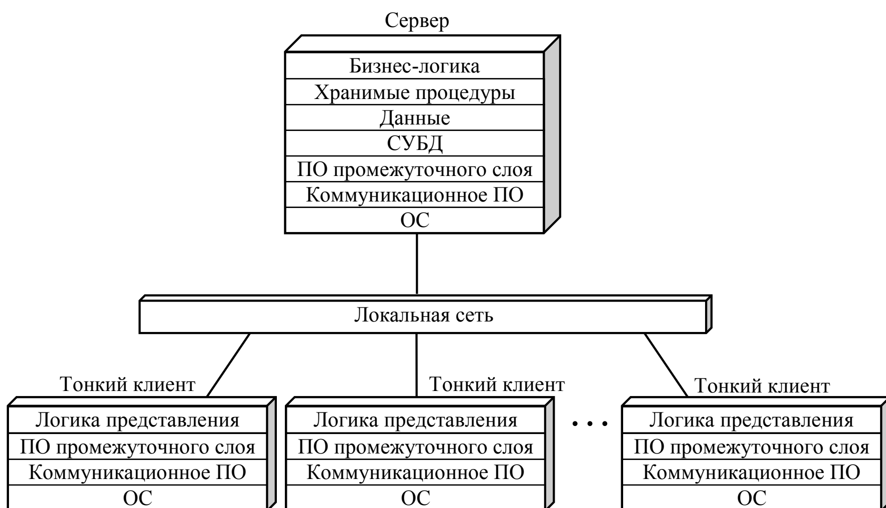


Рис. 17. Перенос на сервер всей бизнес-логики

Со временем стали популярны т. н. сети Интранет. Сети Интранет представляют собой Интернет в миниатюре, например, в пределах предприятия. Используются хорошо зарекомендовавшие себя интернет-протоколы и соответствующие технологии. Как правило, доступ к информационной системе осуществляется через внутренний Интернет-сайт предприятия. При этом настройка и установка специфического программного обеспечения на компьютерах пользователей не требуется – достаточно иметь предустановленный браузер, которым сегодня оснащен практически любой компьютер. По аналогии с WWW в интранет-сетях логика представления информации (в виде HTML-страниц) формируется

не на клиенте, а на сервере. Т.о. вся специфическая функциональность переносится на сервер, а клиенты практически превращаются в терминалы (рис. 18).

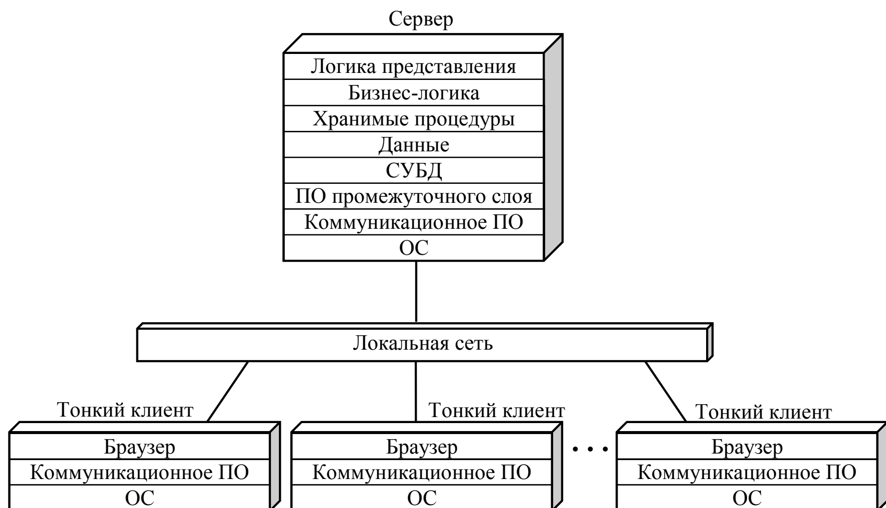


Рис. 18. Перенос на сервер логики представления с использованием технологии интранет

Из приведенного (конечно же, упрощенного) материала можно проследить любопытную эволюционную метаморфозу. Информационные системы предприятий возникли, в виде мэйнфреймов, к которым подключались терминалы пользователей. Затем, при появлении персональных компьютеров, системы распределились по узлам сети. Со временем, они вновь вернулись к центральному компьютеру (теперь уже серверу или серверному кластеру), а компьютеры клиентов вновь стали исполнять роль терминалов. Отличие от мэйнфреймов состоит в том, что серверы не просто предоставляют свои аппаратные ресурсы для решения отдельных пользовательских задач, а предоставляют единое информационное обеспечение предприятия, включая базы данных и бизнес-логику. Централизация упрощает контроль над ресурсами предприятия и модификацию системы.

Рассмотренная двухзвенная архитектура клиент-сервер все чаще заменяется трехзвенной (трехуровневой). А именно, сервер разделяется на два: сервер данных и сервер приложений. Сервер данных по-прежнему содержит в себе базу данных, а на сервер приложений выносится бизнес-логика (рис. 19).

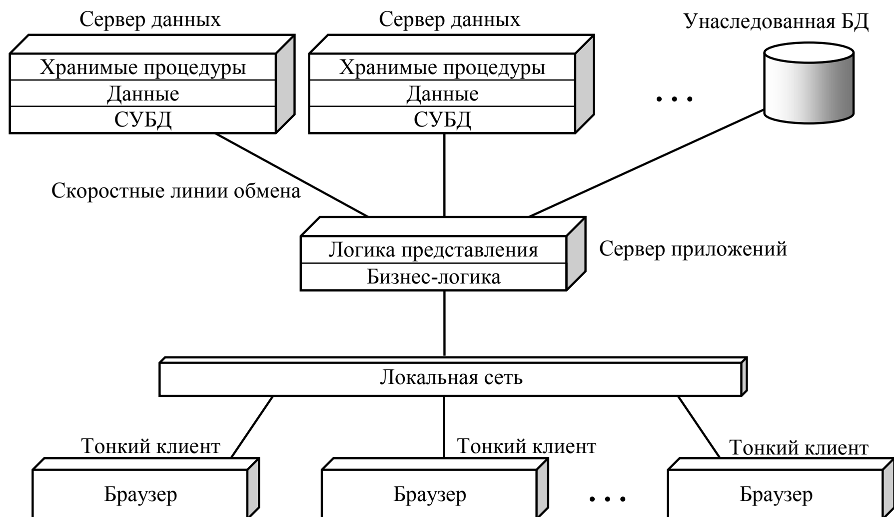


Рис. 19. Трехзвенная архитектура

Трехзвенная архитектура облегчает масштабирование (увеличение пропускной способности) системы и ее переконфигурацию. На рисунке продемонстрировано масштабирование системы двумя серверами данных и интеграция в систему *унаследованной* базы данных (предыдущей версии базы). В данном примере сервер приложений предоставляет в общее пользование некоторый обобщенный интерфейс, за которым скрываются различные варианты реализации прикладной функциональности.

Отделение данных от методов их обработки стало традиционным приемом декомпозиции систем. Данные хранят историю. Для одних и тех же данных могут использоваться различные методы обработки.

4.2. Интеграция систем

Рассмотренный выше пример демонстрирует интеграцию различных баз данных в единую систему. Нередко возникает задача интеграции существовавших независимо систем не только с их базами данных, но и с бизнес-логикой (создать систему из систем). Подобные задачи нередко возникают при слиянии предприятий в корпорации. Для таких случаев используются т. н. *сервисные шины предприятия* (ESB – Enterprise Service Bus). Пример интеграции показан на рис. 20.

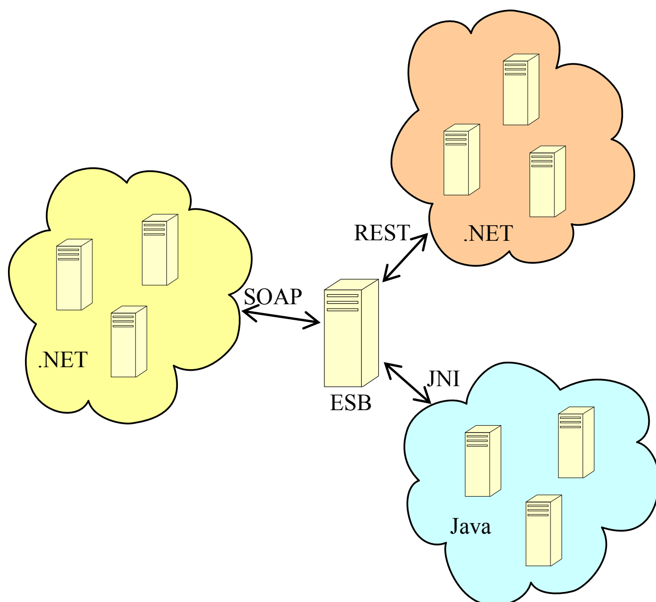


Рис. 20. Взаимодействие систем через центр интеграции

Сервер интеграции (ESB) объединяет несколько систем, разработанных различными командами по разным технологиям. ESB производит маршрутизацию сообщений между системами, транзакции, синхронизацию, преобразование форматов, адаптацию, вызов служб, аудит, протоколирование и пр.

Рассмотренный централизованный способ интеграции систем обладает рядом недостатков. Во-первых, сбой или выход из строя сервера интеграции приводит к парализации взаимодействия всех систем. Во-вторых, ограниченная пропускная способность сервера сужает возможности масштабирования системы.

4.3. Монолитные системы

Сервис-ориентированные архитектуры являются противоположностью *монолитных систем,* в которых компоненты тесно связаны друг с другом, например, при взаимодействии существенно используют знания о внутреннем устройстве, расположении или циклограмме друг друга.

В качестве примера монолитных систем приводится пример первой серии ракетных крейсеров ВМФ США типа «Тикондерога»[1] (рис. 21). Это был огромный и дорогостоящий проект, который по уровню компьютеризации и боевым возможностям явно опередил свое время (начало 1980-х). Однако при плановой модернизации вооружения (которое успевает морально устареть уже к середине срока эксплуатации корабля) оказалось, что стоимость обновления программного и аппаратного обеспечения системы управления крейсера существенно превосходит затраты на постройку нового корабля. В итоге, прослужив половину положенного срока, пять первых кораблей данного типа были списаны при вполне пригодных к эксплуатации корпусах, машинах и механизмах.

Рис. 21. Ракетный крейсер типа «Тикондерога»

[1] Schneider S. What Is Real-Time SOA? http://www.rti.com/docs/RTI_WP_Real-TimeSOA.pdf

Причина сложившейся ситуации заключается в том, что крейсер был спроектирован как монолит, без возможности обновления программного и аппаратного обеспечения. Монолитные системы имеют определенные преимущества, в частности, их проще оптимизировать, например, сделать более компактными. Однако модернизация подобных систем бывает весьма затратной. В настоящее время монолитные архитектуры считаются малопригодными для больших систем.

4.4. Сервис-ориентированные системы реального времени

Урок с монолитными системами управления крейсеров «Тикондерога» не прошел для их разработчиков даром. Следующая модификация боевой информационно-управляющей системы (БИУС) «Иджис» (Aegis) была реализована в соответствии с принципами сервис-ориентированных архитектур. Такой подход позволил не только создать самую эффективную корабельную систему управления оружием[1], но и адаптировать ее под разные типы кораблей (система установлена на ста с лишним кораблях семи типов), а также проводить ее модификации.

Разработанное фирмой Real-Time Innovations (RTI) промежуточное программное обеспечение успешно использовано не только в БИУС «Иджис», но и на множестве морских, авиационных и беспилотных платформ (см. рис. 22). Всего RTI принадлежит более 70% рынка связующего программного обеспечения для распределенных систем реального времени[2].

Удачные технические решения, отработанные на множестве приложений, в 2004 году были специфицированы в виде откры-

---

[1] БИУС «Иджис» позволяет одновременно решать задачи противовоздушной и противолодочной обороны, а также наносить удары по кораблям противника. Она способна осуществлять автоматический поиск, обнаружение, сопровождение нескольких сотен целей, и наведение по наиболее угрожающим из них до 18 ЗУР одновременно. Решение на поражение угрожающих кораблю целей может приниматься автоматически. На испытаниях система поражала за пределами атмосферы групповые баллистические головки, а также отслуживший свой срок спутник.

[2] Демьянов А.В. Связующее ПО стандарта DDS на земле, на воде и в воздухе. АВД Системы. http://www.uav.ru/articles/dds-uav4.pdf

того стандарта DDS[1] (Data Distribution Service), который принят Министерством обороны США в качестве обязательного для программ ВМС OACE (Open Architecture Computing Environment), Армии – FCS (Future Combat Systems) и др. Упомянутые программы предусматривают тесное информационное взаимодействие практически всех боевых единиц посредством цифровой сети.

Эсминец Raytheon DDG-1000

БИУС Aegis

Экспериментальное судно Sea SLICE

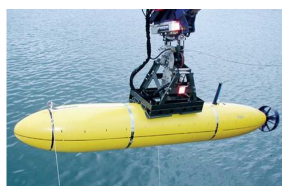
Самолет ДРЛО E-2C Hawkeye

Самолет ДРЛО E-3 Sentry

Истребитель F-35 JSF

БПЛА Predator

Робот RoboScout



Подводный аппарат Bluefin 9

Рис. 22. Примеры использования промежуточного ПО фирмы RTI

---

[1] На момент написания данной работы стандарт реализован многими поставщиками, в том числе и с открытым исходным кодом. В частности, реализация DDS фирмы RTI работает в поверх операционных систем VxWorks, Integrity, LynxOS, QNX, Windows, Linux, Solaris и может использовать в качестве транспортной среды IPv4, IPv6, WAN, разделяемую память (shared memory), шину и коммутируемую структуру (switched fabric). Имеются адаптации под множество языков программирования: C, C++, Java, C#, Ada и др.

Какие же причины способствовали столь успешной реализации и применению DDS?

С организационной точки зрения важно, что стандарт родился не на пустом месте, а стал логическим результатом применения и отработки технических решений на множестве проектов. Кроме того, в разработке стандарта не участвовало множество конкурирующих фирм с противоречащими интересами, т.е. разработка проходила в условиях доверия и сотрудничества.

Не менее важны архитектурные решения, заложенные в стандарт. В DDS использована децентрализованная сервис-ориентированная архитектура, обеспечивающая слабую связность между компонентами. Децентрализация и слабая связность повышает живучесть и устойчивость системы в случае выхода из строя отдельных ее компонент. В таких случаях переконфигурация производится автоматически непосредственно во время работы. Кроме того, децентрализация снимает ограничения по пропускной способности центрального сервера (его просто нет в отличие, например, от рассмотренной выше ESB), что позволяет эффективно масштабировать систему.

Перечисленные свойства удалось обеспечить за счет простых и эффективных решений, при некоторых малозначимых ограничениях (небольшие ограничения дают возможность существенно упростить стандарт и его реализацию). Рассмотрим их детальнее.

Ключевое допущение состоит в том, что система строится в соответствии с *информационно-центрическим* (data-centric) дизайном, в отличие от традиционных технологий типа «удаленный объект» или «клиент-сервер»[1]. Фундаментальным понятием DDS является *информационная модель* системы. В соответствии с данной моделью, единицы циркулирующей в системе информации (т.н. topics) строго типизированы (специфицируются в виде структур данных с помощью языка интерфейсов IDL) и имеют уникальные имена[2].

---

1 Schneider S. What Is Real-Time SOA? http://www.rti.com/docs/RTI_WP_Real-TimeSOA.pdf

2 Подобный дизайн известен также под названием «Классная доска» [2]

Все компоненты распределенной системы взаимодействуют друг с другом посредством заранее зарегистрированных в системе единиц информации. Взаимодействующие компоненты разделяются на поставщиков и потребителей информации (паттерн «издатель/подписчики»). Формат информации должен быть известен компонентам заранее, а параметры обмена могут изменяться и уточняться позже – на этапе эксплуатации.

Принципиально, что ни потребители информации, ни ее поставщики не должны знать ничего друг о друге. Наиболее подходящего поставщика для конкретного потребителя DDS подбирает автоматически во время функционирования. Похожее взаимодействие происходит в системах спутниковой навигации. Спутники предоставляют свою информацию в одностороннем порядке, не зная, кто и как будет ее потреблять. В свою очередь навигационные приемники используют в качестве поставщиков информации те спутники, которые в данный момент находятся в зоне их видимости. Разница состоит в том, что в случае DDS подбор наиболее подходящих поставщиков производит не потребитель, а связующее ПО. Для подбора конкретного поставщика DDS использует дополнительную информацию – т. н. параметры качества обслуживания (QoS – Quality of Service), которые потребители и поставщики указывают системе заранее. В системе предусмотрено более 20 параметры QoS: темп поступления информации, приоритетность, допустимые задержки, возможность (или невозможность) пропуска некоторых данных и т. п.

Эффективность рассмотренного способа взаимодействия позволяет обеспечивать минимально возможные задержки поступления информации и удовлетворять очень жестким требованиям реального времени (микросекунды, десятки микросекунд); автоматизировать взаимодействие сотни приложений; масштабировать систему до тысячи компьютеров.

Для сравнения рассмотрим традиционные модели взаимодействия компонент системы «удаленный объект» или «клиент-сервер». В этих моделях связи клиенты устанавливают связь непосредственно с серверами и взаимодействуют с ними по типу

«точка-точка». Взаимосвязи в подобных системах продемонстрированы на примере самолета ДРЛО E-2C Hawkeye[1] (см. рис. 23).

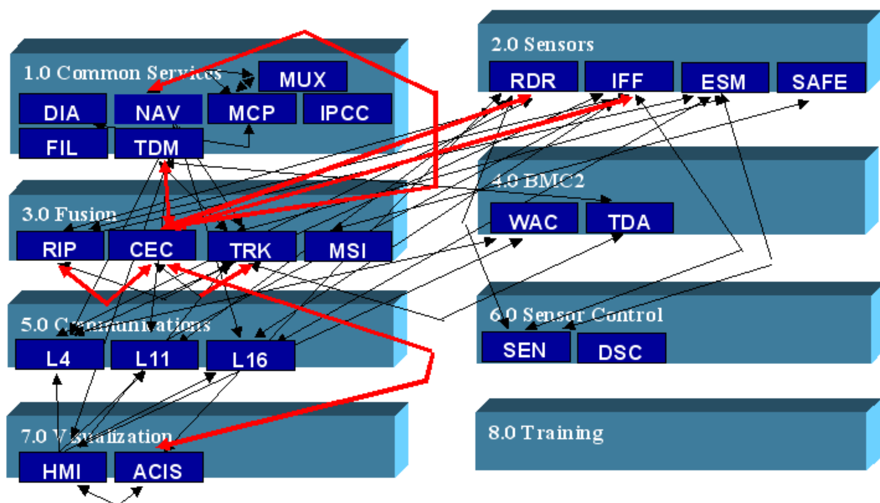


Рис. 23. Архитектура бортовой системы самолета E-2C до модернизации

Структура системы напоминает антипаттерн «Спагетти». Кроме того, данная структура должна быть статичной, серверы всегда доступны, клиенты должны знать, где располагаются серверы, когда и в какой последовательности можно посылать им запросы, взаимодействие компонент синхронно (запрос/ответ). Подобный «серверо-центрический» дизайн делает систему сильносвязанной, монолитной. Добавление новых потоков данных (красные линии) или локальные изменения могут привести к неработоспособности всей системы.

Традиционная объектно-ориентированная концепция скрывает состояние объекта (удаленного объекта, сервера) за его функциональным интерфейсом. Хотя состояние скрыто, но оно неявно подразумевается. Нужно как-то узнать инициализирован объект

[1] Schneider S. The Data-Centric Future. http://www.rti.com/whitepapers/Data-Centric_Future_Pt1.pdf

или нет, в каком режиме он функционирует и пр. В зависимости от режима нужно правильно сформировать запрос. Именно такой стиль навязывают технологии RPC, CORBA, EJB и пр. Ситуация существенно усложняется, когда в подобном стиле взаимодействуют несколько объектов.

В информационно-центрических системах наоборот – данные выносятся во главу угла, а объекты и их методы скрываются. Компоненты взаимодействуют не между собой, а со связующим ПО. Добавление новых компонент и/или связей практически не влияет на соседние компоненты (см. рис. 1). С применением DDS структура программного обеспечения самолета E-2C Hawkeye упорядочилась следующим образом (см. рис. 24).

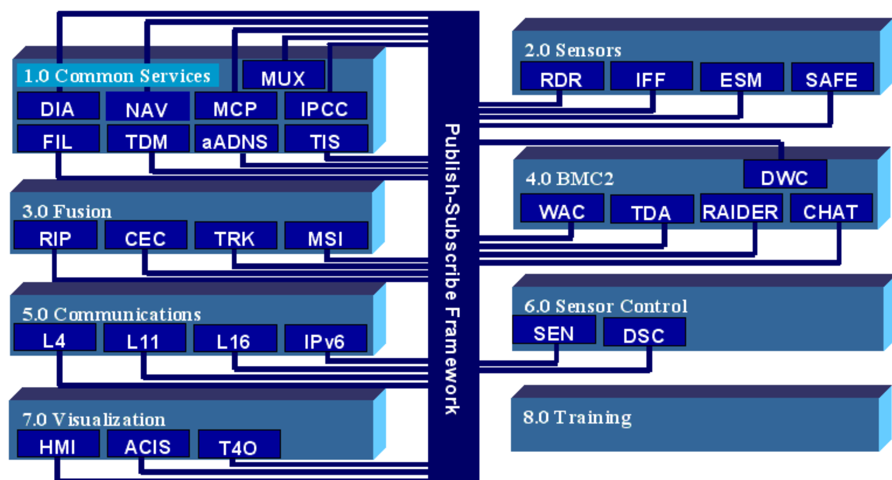


Рис. 24. Архитектура бортовой системы самолета E-2C после модернизации

Преимущество информацинно-ориентированной системы рассмотрим на упрощенном примере зенитно-ракетного комплекса (рис. 25). Пусть комплекс оснащен несколькими радиолокационными станциями (РЛС) и несколькими батареями зенитных управляемых ракет (ЗУР). Их координацию традиционно обеспечивает командный пункт (КП).

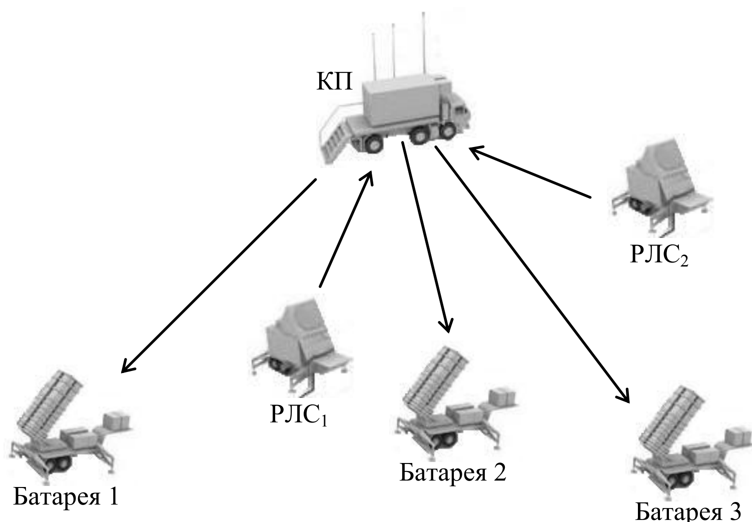


Рис. 25. Централизованное управление зенитно-ракетным комплексом

Выход из строя батареи ЗУР или одной из РЛС несколько снижает эффективность комплекса, но не сводит ее к нулю, поскольку командный пункт адаптируется под ситуацию. Если же будет выведен из строя командный пункт, то весь комплекс перестанет функционировать. Другими словами, централизованная система более уязвима. Кроме того, пропускная способность командного пункта ограничена некоторым максимальным числом ЗУР и РЛС.

Перечисленных недостатков лишен комплекс ПВО, построенный по правилам DDS (см. рис. 26).

РЛС (поставщики информации) предоставляют свои измерения во всеобщее использование, не зная, кому они предназначаются. Батареи ЗУР (потребители) используют наиболее подходящие из доступных измерений РЛС. Выход из строя любого компонента не приводит к потере боеспособности комплекса. Инфраструктура также трудноуязвима в силу своей децентрализованости (могут использоваться также обходные, дублированные, резервные каналы обмена и т.п.).

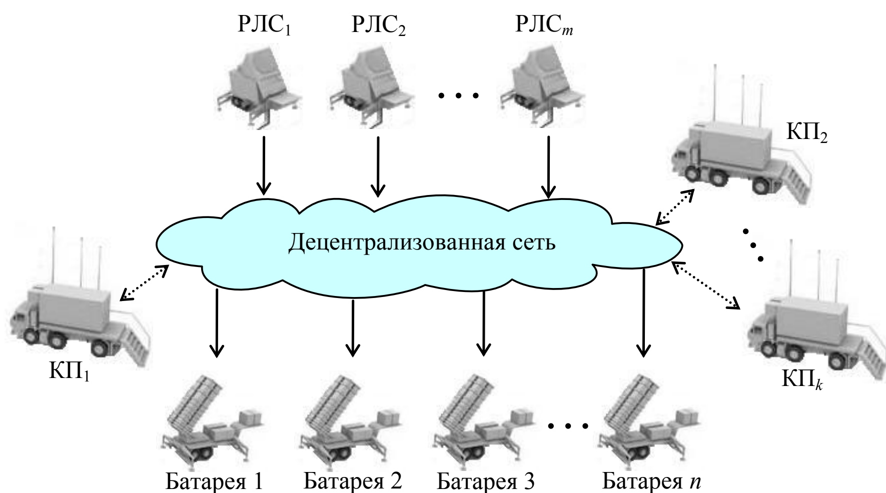


Рис. 26. Децентрализованное управление зенитно-ракетным комплексом

Рассмотренная концепция вовсе не отменяет возможности централизованной координации действий компонент. Просто централизация производится не на уровне инфраструктуры, а на прикладном уровне. Командные или координационные пункты могут подключаться к любой точке входа в систему, могут резервироваться и пр. Кроме того, функции командного пункта теоретически может выполнять любой компонент системы, обладающий достаточными вычислительными ресурсами.

Оперативную информацию могут поставлять не только РЛС комплекса, но и другие источники: разведывательные аппараты, корабли, спутники и пр. Аналогично могут выбираться наиболее подходящие системы поражения для уничтожения обнаруженных целей: авиация, артиллерия и т.д. Подобное сетевое взаимодействие всевозможных боевых компонентов является главной особенностью разрабатываемой в США «сетецентрической» боевой системы будущего (рис. 27).

Рис. 27. Перспективная сетецентрическая организация армии США

Для реализации подобной глобальной системы необходимо интегрировать не только отдельные компоненты, но и целые системы (система систем). Для этой цели DDS предусматривает простые и эффективные инструменты. Например, чтобы воспользоваться информацией соседней системы нужно «подписаться» на нее в качестве одного из потребителей и распространить эту информацию в своей системе в качестве поставщика данных. Имеются также возможности автоматического преобразования данных в нужный формат (например, перевод из одной системы координат в другую и пр.).

Хотя DDS и предоставляет важные средства для подобных систем, но этих средств явно недостаточно. Разработка систем подобного масштаба требует привлечения множества технологий и стандартов. На рис. 28 продемонстрирована многоуровневая архитектура глобальной информационной инфраструктуры армии США.

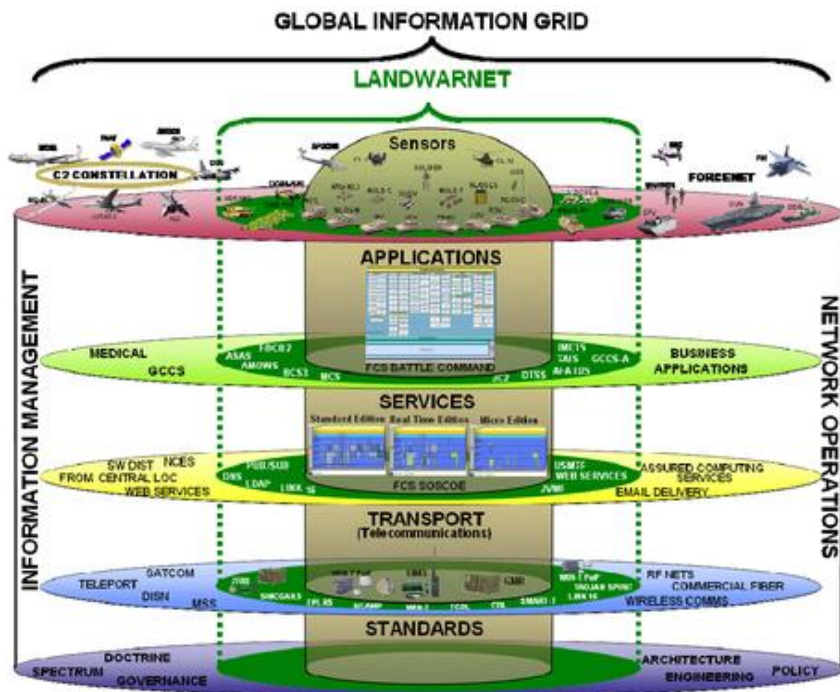


Рис. 28. Разрабатываемая глобальная информационная инфраструктура армии США

4.5. Интернет

Обзор цифровых технологий был бы ущербным без рассмотрения, пожалуй, самого успешного технического проекта человечества – Интернета.

Интернет изначально проектировался, как инфраструктура, объединяющая разнообразные локальные компьютерные сети (составная сеть). Для этого, в частности, был разработан протокол IP[1] (Internet Protocol), который предусматривает возможность

[1] Данный протокол предусматривает разбиение потока данных на пакеты. Каждый пакет имеет заголовок определенного формата, в котором указываются IP-адреса получателя и отправителя.

глобальной адресации компьютеров (узлов сети) с помощью IP-адресов. Использование IP для межсетевого взаимодействия позволяет решать задачу *маршрутизации* – выбора конкретных линий передачи пакетов от отправителя к получателю[1]. При этом пакеты (датаграммы) передаются независимо друг от друга и их надежная доставка не гарантируется (надежность доставки обеспечивают вышележащие протоколы). Протокол IP соответствует сетевому уровню модели OSI[2] (см. выше).

IP является ярким примером инфраструктуры открытой системы, в которой изначально тщательно продуманы возможности наращивания сети[3], ее переконфигурации, что послужило фундаментом для успешного развития и эксплуатации Интернета. Наверное самой важной идеей IP является уникальный адрес каждого абонента сети.

Как и полагается для многоуровневой архитектуры (рис. 29), слой IP может работать поверх различных физических линий передачи и соответствующих протоколов канального уровня: Ethernet, Wi-Fi, PPP, ATM, xDSL, Token Ring и пр. С другой стороны, поверх IP находятся другие слои. В частности, протокол TCP (Transfer Control Protocol – протокол управления обменом) позволяет устанавливать двустороннюю связь между двумя произвольными узлами Сети[4] и гарантирует надежную доставку потока данных в обе стороны (транспортные уровни OSI и DOD). Протоколы TCP и

---

1 Решаемая задача неоднозначна из-за возможности существования альтернативных маршрутов в сетях со сложной конфигурацией.

2 Здесь уместнее говорить не о модели OSI, а о модели сетевого взаимодействия DOD (Department of Defense – Министерство обороны США), в соответствии с которой реализован Интернет.

3 На момент разработки IP вряд ли кто-нибудь мог себе представить, какую популярность завоюет Интернет. Поэтому 32-х разрядов, выделенных для IP-адреса и обеспечивающих подключение около 4 млрд компьютеров, казалось на тот момент вполне достаточным. Однако совсем скоро стал очевидным недостаток длины IP-адреса. Новая версия IPv6 предусматривает 128 разрядов под IP-адрес.

4 Иногда, вместо названия Интернет используют слово Сеть (с большой буквы).

IP органично дополняют друг друга так, что их обозначают TCP/IP, как нечто неразрывное, и даже весь стек протоколов, включая ниже- и вышележащие, называют TCP/IP. Над транспортными слоями TCP/IP располагаются протоколы прикладного уровня: HTTP, FTP, SMTP, DNS и пр., реализующие различную прикладную функциональность: передача информации в гипертекстовом виде, обмен файлами, почтовыми сообщениями и т.д.

Уровень	Примеры протоколов
Прикладной	HTTP, FTP, SMTP, POP, IMAP, DNS, ...
Транспортный	TCP, UDP
Сетевой	IP
Канальный	Ethernet, Wi-Fi, ATM, PPP, Token Ring, ...

Рис. 29. Модель сетевого взаимодействия в Интернет

Стек Интернет-протоколов является ярчайшим примером эффективности многослойных архитектур, декомпозирующих различные технологии. С одной стороны, подмена нижележащих слоев обеспечивает функционирование Сети поверх различных физических носителей (электрические, оптические, электромагнитные, инфракрасные). С другой стороны, поверх инфраструктуры Интернета функционируют независимо друг от друга множество прикладных систем, разделяя инфраструктуру, как общий ресурс. Сеть непрерывно наращивается новыми узлами и подсетями, переконфигурируется, старые стандарты постепенно заменяются новыми (продолжая сосуществовать в разных сегментах), появляются новые слои (например, SOAP поверх HTTP, см. выше), регулярно появляются новые виды Интернет-услуг, сервисов и т.д.

Можно заметить, что у столь широких и разнообразных возможностей взаимодействия есть нечто общее, объединяющее. Таким общим знаменателем представляется протокол IP (см. таблицу выше). Это тот «язык», который должны понимать все взаимодействующие участники Сети. Похожую роль играл рассматривавшийся ранее байт-код, который, с одной стороны, позволял

объединять программные модули, написанные на разных языках высокого уровня (предварительно скомпилированные в байт-код), с другой стороны, подходящие интерпретаторы позволяли исполнять полученный байт-код на разных процессорах. Аналогичную роль при взаимодействии программ играли, рассматривавшиеся ранее языки IDL и SOAP. Интерфейс сервера приложений в рассматривавшейся трехзвенной архитектуре также играет похожую роль единого стандарта, на базе которого интегрируются различные компоненты корпоративной системы. Приведенные примеры демонстрируют эффективность интеграции многослойных систем на базе единого языка/протокола/стандарта.

Учитывая упоминавшиеся проблемы с промежуточным программным обеспечением, может возникнуть недоумение: почему не удалось разработать единый мировой стандарт байт-кода или языка IDL, а протокол IP стал стандартом де-факто? Ответ на этот вопрос попытаемся дать после рассмотрения технологии WWW.

4.6. Всемирная паутина

WWW или *World Wide Web,* или *Всемирная паутина,* или просто *веб* сейчас многими воспринимаются, как синонимы Интернета, хотя это не совсем верно. Веб – это одна из многих распределенных систем, развернутых поверх инфраструктуры Интернета. Однако данная система является наиболее используемой. Благодаря ей Интернет приобрел широчайшую популярность и распространение.

Веб представляет собой информационную систему, предоставляющую доступ к связанным между собой документам, расположенным на различных компьютерах, подключенных к Интернету.

Большинство документов веб *(веб-страниц)* оформляются в соответствии с правилами *HTML*[1] (Hyper Text Markup Language) –

[1] Для передачи по сети HTML-документов используется протокол прикладного уровня HTTP (Hyper Text Transfer Protocol – протокол передачи гипертекста). Данный протокол является одним из самых распространенных в Интернете протоколов прикладного уровня.

языка разметки гипертекста. Правила HTML просты настолько, что позволяют создавать веб-страницы с использованием обычных текстовых редакторов, например, с помощью Notepad. Существует также большое количество редакторов, позволяющих создавать страницы в интуитивно понятном режиме WYSIWYG (What You See Is What You Get – что видишь, то и получишь): Dreamweaver, FrontPage, AceHTML, WebStorm и др. Простота HTML позволяет без затруднений формировать веб-страницы динамически (автоматически) в момент запроса клиента, в зависимости от контекста. Другими словами, технология разработки HTML-документов доступна настолько, что позволяет создавать и публиковать веб-страницы и *веб-сайты* (совокупности веб-страниц, объединенные общей темой) даже неискушенным разработчикам.

Неотъемлемой частью гипертекста являются т. н. *гиперссылки* URL (Uniform Resource Locator – универсальный локатор ресурса) на другие документы, которые делают текст не ограниченным рамками текущего документа (отсюда приставка гипер-). Гиперссылка содержит в себе параметры, необходимые для получения искомого документа: место расположения сервера в сети (IP-адрес или т. н. *доменное имя*), место расположения документа на сервере (URL-путь) и др. С помощью гиперссылок веб-страницы могут устанавливать произвольные, в том числе перекрестные, ссылки друг на друга. Ссылающиеся друг на друга документы являются одновременно и базой знаний и реестром. Следуя этим ссылкам, можно осуществлять поиск нужной информации *(веб-серфинг)*. Поиск может осуществляться не только в ручном режиме, но и автоматически – с помощью специальных программ, а чаще всего, с помощью специализированных поисковых серверов.

Таким образом, веб предоставляет пользователям простой способ публикации и поиска нужной информации для рекламы, коммерческих и др. целей. Пользователи самостоятельно или с привлечением специалистов наполняют Сеть веб-страницами и устанавливают ссылки между ними (конфигурируют систему). По сути, мировая информационная система создается (точнее – на-

полняется информацией и конфигурируется) руками самих же пользователей. Для них предоставляется только инфраструктура, технологии и инструменты. Веб является ярким примером привлечения к проекту миллионов сторонних разработчиков.

4.7. Причины успеха Интернета

Почему же Интернет приобрел столь широкую популярность? Почему его не постигла участь, например, промежуточного программного обеспечения? Данному феномену видится несколько причин.

Во-первых, инфраструктура типа Интернет оказалась востребованной широчайшим кругом пользователей.

Во-вторых, в Интернет заложены «генетически» правильные технические решения, позволяющие интегрировать различные информационные ресурсы, развивать, наращивать и переконфигурировать систему (открытую систему) мирового масштаба. В этом смысле следует особо выделить протоколы TCP/IP, в частности, IP-протокол и его IP-адрес. Принципиальное свойство Интернета заключается в его децентрализованости, обеспечивающей живучесть и устойчивость.

В-третьих, предоставляемая пользователям инфраструктура, базовые технологии и инструментарий доступны и просты в использовании. Пользователи (коммерческие и некоммерческие) сами наполняют Сеть ресурсами, наращивают и конфигурируют ее (привлечение сторонних разработчиков).

В-четвертых, все базовые стандарты Интернет, в частности, протоколы, полностью открыты и бесплатны. Гарантируется, что ни коммерческий, ни государственный, ни какой другой субъект не будет владеть правами на эти стандарты.

В-пятых, базовые технологии Интернета успели созреть до того момента, когда коммерческие организации начали проявлять к нему интерес. Технологии изначально развивались в военном ведомстве США, затем в университетах, т.е. в достаточно узкой, высококвалифицированной среде, сконцентрированной на достижении именно технического результата, а не коммерческого (т.е. в атмосфере доверия и сотрудничества). Когда Интернет был открыт для всеобщего использования, его базовые технологии были восприняты, как данность, а противостояние коммерческих компаний происходило уже на другом уровне[1], например, на рынке браузеров (известном, как война браузеров). Вполне вероятно, что если бы базовые технологии разрабатывались конкурирующими коммерческими организациями, то мы до сих пор не имели бы единой, столь эффективной и успешной инфраструктуры.

Последний тезис неочевидный и неоднозначный. Можно привести немало примеров, когда при всех благоприятных условиях, в отсутствии конкуренции проекты не оправдывали полагавшихся на них надежд. В то же время аналогичные проекты, развивавшиеся в условиях жесткой конкуренции и при недостатке средств, показывали лучшие результаты.

[1] После открытия Интернета для всеобщего доступа коммерческие организации не сразу оценили его значимость. Взрыв популярности Интернета произошел при появлении Веб, точнее HTML/HTTP и браузера Mosaic. Всего лишь за два года с момента их появления Интернет получил распространение по всему миру, хотя до этого был известен только узкому кругу специалистов. Возможно, из-за такого стремительного распространения Интернета коммерческие организации не успели сориентироваться и фрагментировать рынок на уровне базовых стандартов.

ВЫВОДЫ

Беглый и поверхностный обзор технологий проектирования не может передать всего многообразия идей и решений, использовавшихся и используемых при разработках сложных систем. Тем не менее, самые эффективные решения прошли жесткий эволюционный фильтр и стали общепринятыми. В данной работе проводилась попытка выделить именно такие общепринятые решения и принципы. Еще раз кратко перечислим наиболее значимые из них.

Основополагающее правило – максимальное использование накопленного опыта. Данное правило проявляется в разных формах: непосредственное или косвенное привлечение к разработке специалистов, автоматизация, многократное повторное использование собственных и сторонних удачных решений, прототипов, компонент, технологий, инструментария, документации, обучении и пр.

Многообразие используемых в проекте сложной системы средств, компонент и технологий порождает одну из самых сложных проблем – проблему их увязки в единое целое. Ключевую роль здесь играют: разделение труда, декомпозиция, агрегирование, структура, инфраструктура и архитектура системы, общесистемные ресурсы, интерфейсы и стандарты. Наиболее ярко разделение технологий проявляется в многослойных архитектурах.

Система серьезного масштаба не может родиться в одночасье. Соответственно, грамотно спроектированная система продолжает развиваться и после сдачи в эксплуатацию посредством наращивания, переконфигураций, модификаций, модернизаций и т. д. Как показывает мировая практика, наиболее эффективными в этом смысле оказываются открытые системы со слабой связностью компонент, в частности, сервис-ориентированные архитектуры (яркий пример открытой системы – Интернет).

Организация процесса разработки также играет важную роль в успехе проекта. В зависимости от конкретных условий и решаемой задачи подходящий процесс разработки может варьироваться от предельно формализованного до минимально контролируемого. Поскольку ключевыми участниками процесса являются разработчики, т.е. люди, то немаловажную роль играют человеческие отношения, сплоченность команды, мотивация разработчиков, выражающаяся не только в денежном вознаграждении. Какими бы совершенными ни были используемые технологии, самым главным ресурсом любого проекта и любой проектной фирмы являются ее специалисты.

ЛИТЕРАТУРА

1. Александреску А. Современное проектирование на C++. – М.: Вильямс, 2008. 336 с.
2. Басс Л., Клементс П., Кацман Р. Архитектура программного обеспечения на практике. 2-е издение. – СПб.: Питер, 2006. 575 с.
3. Брукс Ф. Мифический человеко-месяц, или как создаются программные системы. – СПб.: Символ-Плюс, 2010. 304 с.
4. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2008. 366 с.
5. Гома Х. UML. Проектирование систем реального времени, параллельных и распределенных приложений. – М.: ДМК Пресс, 2011. 704 с.
6. Гринфилд Дж., Шорт К. и др. Фабрики разработки программ. Потоковая сборка типовых приложений, моделирование, структуры и инструменты. – М.: Диалектика-Вильямс, 2007. 592 с.
7. Коплиен Дж. Программирование на C++. – СПб.: Питер, 2005. 408 с.
8. Макконнелл С. Совершенный код. – СПб.: Питер, 2005. 896 с.
9. Таненбаум Э., Стеен М. Распределенные системы. Принципы и парадигмы. – СПб.: Питер, 2003. 877 с.
10. Фаулер М. Рефакторинг. Улучшение существующего кода. – СПб.: Символ-Плюс, 2003. 432 с.
11. Шмидт Д., Хьюстон С. Программирование сетевых приложений на C++. Том 1. Профессиональный подход к проблеме сложности: ACE и паттерны. – М.: ООО «Бином-Пресс», 2007. 304 с.
12. Шмидт Д., Хьюстон С. Программирование сетевых приложений на C++. Том 2. Систематическое повторное использование: ACE и каркасы. – М.: ООО «Бином-Пресс», 2007. 400 с.
13. Якобсон А., Буч Г., Рамбо Дж. Унифицированный процесс разработки программного обеспечения. – СПб.: Питер, 2002. 496 с.
14. Boehm B. Software Engineering. Englewood Cliffs, N.J.: Prentice Hall, 1981.
15. Brooks F. No Silver Bullet – Essence and Accidents of Software Engineering. Proceedings of the IFIP 10-th World Computing Conference, pp. 1069-1076, 1986 (издана на русском языке в сборнике [3]).
16. Henning M. The Rise and Fall of CORBA, ACM Queue, Volume 4, Number 5, June 2006 (перевод: С. Кузнецова на http://citforum.ru/SE/middleware/corba_history).
17. Schmidt D. et al, Pattern Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2. Wiley, 2000.

А.И. Петербург, Ю.Д. Тычинский

ТЕНДЕНЦИИ И ПРИНЦИПЫ ПРОЕКТИРОВАНИЯ СЛОЖНЫХ СИСТЕМ
Цифровые системы

Учебное пособие

Наш сайт: www.etnosocium.ru
e-mail: etnosocium@mail.ru
Необходимую научную литературу Вы можете
приобрести на сайте: www.knigadom.com

Оригинал-макет подготовлен Международным
издательским центром «ЭТНОСОЦИУМ»
Отпечатано в типографии Международного
издательского центра «ЭТНОСОЦИУМ»,
105066, Москва, Спартаковская ул., д. 19, стр. 3.

Зам. гл. ред. С.В. Чапкин
Редактор А.Н. Андреев
Корректор Ф.Э. Вайс
Дизайн и верстка О.А. Шишова

Бумага офсетная № 1. Гарнитура CharterC.
Формат 60x90/16. Тираж 1000 экз. Усл. п. л. 7,875