ПРОФЕССИОНАЛЬНО

Нил Форд, Майкл Найгард,
Билл де Ора и др.

97

ЭТЮДОВ ДЛЯ АРХИТЕКТОРОВ ПРОГРАММНЫХ СИСТЕМ

СИМВОЛ®

97 Things
Every Software Architect Should Know

Collective Wisdom from the Experts

Edited by Richard Monson-Haefel

O'REILLY®

ПРОФЕССИОНАЛЬНО

97 этюдов для архитекторов программных систем

Опыт ведущих экспертов

Нил Форд, Майкл Хайгард,
Билл де Ора и др.

СИМВОЛ®

Санкт-Петербург — Москва
2010

Серия «Профессионально»

Нил Форд, Майкл Хайгард, Билл де Ора и др.

97 этюдов для архитекторов программных систем

Перевод Е. Матвеева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*С. Тепляков*
Редакторы	*В. Подобед, Е. Тульсанова*
Корректор	*О. Макарова*
Верстка	*Д. Орлова*

Форд Н., Хайгард М., де Ора Б.

97 этюдов для архитекторов программных систем. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 224 с., ил.

ISBN 978-5-93286-176-9

Успешная карьера архитектора программного обеспечения требует хорошего владения как технической, так и деловой сторонами вопросов, связанных с проектированием архитектуры. В этой необычной книге ведущие архитекторы ПО со всего света обсуждают важные принципы разработки, выходящие далеко за пределы чисто технических вопросов.

Архитектор ПО выполняет роль посредника между командой разработчиков и бизнес-руководством компании, поэтому чтобы добиться успеха в этой профессии, необходимо не только овладеть различными технологиями, но и обеспечить работу над проектом в соответствии с бизнес-целями. В книге более 50 архитекторов рассказывают о том, что считают самым важным в своей работе, дают советы, как организовать общение с другими участниками проекта, как снизить сложность архитектуры, как оказывать поддержку разработчикам. Они щедро делятся множеством полезных идей и приемов, которые вынесли из своего многолетнего опыта. Авторы надеются, что книга станет источником вдохновения и руководством к действию для многих профессиональных программистов.

ISBN 978-5-93286-176-9
ISBN 978-0-596-52269-8 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 30.03.2010. Формат 70×90 1/16. Печать офсетная.
Объем 14 печ. л. Тираж1600 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

Архитекторы программного обеспечения занимают особое место в мире информационных технологий. От архитектора ожидают одинаково глубокого понимания как технологий и программных платформ, используемых в его организации, так и вопросов бизнеса, которому они служат. Хороший архитектор ПО должен в совершенстве знать обе стороны архитектуры – и коммерческую, и технологическую. Эта задача не из простых – вот почему и была написана эта книга.

Перед вами сборник советов от архитекторов ПО со всего света. Их советы затрагивают широкий круг тем – от предотвращения распространенных ошибок до создания выдающихся проектных команд. В сущности, это «сборная солянка» из рекомендаций признанных архитекторов ПО для других архитекторов и тех, кто хочет стать архитектором.

Искренне надеюсь, что эта книга явится источником вдохновения и руководством к действию для многих людей, профессионально занимающихся разработкой программного обеспечения. Хочется надеяться также, что архитекторы программного обеспечения будут использовать эту книгу (а также родственный ей веб-сайт) для обмена рекомендациями и находками в своей профессиональной сфере – возможно, самой сложной на сегодняшний день в области информационных технологий.

Книга «97 этюдов для архитекторов программных систем» очень сильно отличается от всех прочих книг, которые вам доводилось читать. Она была создана совместными усилиями более чем полусотни авторов; все они щедро делились своими мыслями и советами в области проектирования архитектуры программного обеспечения без денежного вознаграждения. Можно сказать, что эта книга следует принципам продуктов с открытым исходным кодом в их истинном смысле. Каждый автор внес свой вклад, написав одну или несколько статей, которые затем были тщательно проработаны и отрецензированы. Лучшие статьи были отобраны для публикации. В этом отношении книга похожа на программный проект с открытым исходным кодом, только здесь вклад участников – не программный код, а знания и мудрость.

Разрешение на использование материалов

Использование всех материалов этой книги также осуществляется на условиях, сходных с условиями распространения ПО с открытым исходным кодом. Каждая статья находится в свободном доступе на условиях лицензии Creative Commons, Attribution 3; это означает, что вы можете использовать отдельные статьи в своей работе, если упоминаете их авторов. Попытки публикации книг в соответствии с принципами распространения продуктов с открытым исходным кодом совершались и прежде, но все они (за немногочисленными исключениями) окончились неудачей. Вероятно, это объясняется трудностями координации вкладов отдельных участников, если проект не имеет четкого модульного разделения. Именно модульность обеспечила успех этой книги. Каждая статья существует независимо от других статей и ценна как в контексте сборника в целом, так и сама по себе.

Как с нами связаться

Пожалуйста, со всеми комментариями и вопросами, касающимися этой книги, обращайтесь к издателю:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (США или Канада)

707-829-0515 (международные или местные звонки)

707 829-0104 (факс)

На сайте издательства имеется веб-страница книги со списком обнаруженных опечаток и другой дополнительной информацией. Страница доступна по адресу:

http://www.oreilly.com/catalog/9780596522698

Сопроводительный сайт книги, на котором размещены все тексты и полные биографии авторов, доступен по адресу:

http://97-things.near-time.net

С комментариями и техническими вопросами по поводу книги обращайтесь по электронной почте:

bookquestions@oreilly.com

Дополнительную информацию о книгах, конференциях, ресурсах и O'Reilly Network вы можете найти на сайте издательства O'Reilly:

http://www.oreilly.com

Safari® Enabled

Логотип Safari® Enabled на обложке вашей любимой книги, посвященной какой-либо технологии, означает, что книга доступна через O'Reilly Network Safari Bookshelf.

Система Safari превосходит обычные электронные книги. Это целая виртуальная библиотека, позволяющая выполнять поиск по тысячам лучших технических книг, копировать примеры кода, загружать главы и быстро находить ответы, когда вам потребуется самая точная и актуальная информация. Safari можно бесплатно опробовать на сайте *http://safari.oreilly.com*.

Благодарности

Идея книги «97 этюдов для архитекторов программных систем» родилась не на пустом месте. Множество людей заслуживают благодарности за идею этой книги и ее исполнение. Я хочу поблагодарить Джея Циммермана (Jay Zimmerman), предложившего мне провести презентацию «10 вещей, которые должен знать каждый архитектор программного обеспечения» на симпозиуме «No Fluff, Just Stuff»; Брюса Эккеля (Bruce Eckel) за ведение списка рассылки, на базе которого зародилась идея этой книги; Джереми Мейера (Jeremy Meyer) за предложение написать книгу по материалам простой слайдовой презентации; Нитина Борванкара (Nitin Borwankar), предложившего создать общедоступный вики-сайт, чтобы все желающие могли принять участие; участников списка рассылки Брюса, которые, не имея ничего, кроме моих обещаний, приняли решение уделить время этой идее и предложили первые статьи для книги. Хочу поблагодарить также десятки архитекторов программного обеспечения, приложивших серьезные усилия к работе над материалами, которые в результате не вошли в эту книгу. Мне было очень трудно решить, какие статьи должны стать частью книги. Я глубоко благодарен всем, кто внес свой вклад в общую работу, независимо от того, попали их материалы в книгу или нет.

Я благодарен также издательству O'Reilly, которое без предубеждения восприняло идею и поддержало этот никем прежде не опробованный метод работы над книгой. O'Reilly заслуживает благодарности и за согласие лицензировать весь материал на условиях продуктов с открытым исходным кодом (лицензия Creative Commons, Attribution 3) и предоставить свободный доступ к содержимому на веб-сайте. Среди сотрудников O'Reilly мне хотелось бы особо поблагодарить Майка Лукидеса (Mike Loukides), Майка Хендриксона (Mike Hendrickson), Лору Пейнтер (Laura Painter) и Лорел Экерман (Laurel Ackerman). Без их помощи и поддержки этот проект был бы невозможен.

В настоящее время мы (O'Reilly и я) работаем над другими проектами новой уникальной серии «97 Things», которая позволяет воспользоваться коллективным разумом экспертов в различных практических областях. Управление проектами, разработка программного обеспечения, архитектура данных – вот лишь некоторые из тем, над которыми мы сейчас трудимся.

Надеюсь, книга покажется вам интересной, а возможно, даже вдохновит вас на подготовку собственных статей для будущих проектов!

С наилучшими пожеланиями,

Ричард Монсон-Хейфел,
редактор серии «97 Things»

Не ставьте свое резюме выше интересов клиента

Нитин Борванкар

Мы, технари, подчас выбираем для использования те или иные технологии, методологии и подходы к решению задач не потому, что они обеспечивают оптимальное решение, а лишь потому, что в глубине души нам хочется упомянуть их в своем резюме. Такой выбор очень редко приводит к положительному результату.

Самым мощным катализатором вашей карьеры будут благодарные заказчики, выстроившиеся в длинную очередь, чтобы порекомендовать вас другим – ведь вы так хорошо для них потрудились. Благосклонность клиентов послужит вам на порядок лучше, чем любой новомодный объект новомодного языка и любая свежеизобретенная парадигма. Хотя для архитектора очень важно (и даже жизненно необходимо) быть в курсе новейших тенденций и технологий, никогда не пытайтесь расширять свой кругозор за счет клиента. Помните, что вам как архитектору доверено благополучие вашей организации; соответственно от вас ожидают, что вы будете честно и грамотно действовать в интересах заказчика, избегая любых конфликтов интересов и сохраняя полную лояльность своей организации. Если предложенный проект недостаточно актуален или перспективен для ваших текущих карьерных целей, найдите другой проект.

А что, если это невозможно, и вы все же вынуждены участвовать в таком проекте? И вам самому, и всем остальным будет лучше, если вы будете выбирать технологию в интересах клиента, а не своего резюме. Подчас трудно устоять перед искушением применить новое модное решение, даже если оно плохо подходит к текущей ситуации.

При правильно выбранном решении вы получаете довольную команду и удовлетворенного клиента, а общая напряженность работы над проектом заметно

снижается. Часто это позволяет вам глубже изучить уже знакомую технологию или заняться освоением новинки в свободное время. А может быть, у вас даже высвободится время для посещения курсов живописи, о которых вы всегда мечтали. Ваши близкие это тоже оценят – они заметят разницу в вашем состоянии, когда вы приходите домой после работы.

Всегда ставьте долгосрочные потребности клиента над своими собственными краткосрочными потребностями, – и вы не ошибетесь.

В начале 1990-х Нитин Борванкар (Nitin Borwankar) работал в Ingres и Sybase, где создавал первые веб-приложения на базе SybPerl и OraPerl, а вскоре после этого – на базе ранних версий Enterprise Java. Он также был активным участником New-EDI – процесса IETF-стандартизации[1] *EDI в Интернете. С 1994 года работал независимым консультантом и исследователем в области организации и интеграции данных в масштабах предприятия, а также обмена сообщениями. В настоящее время занимается схемами баз данных для приложений с фолксономической*[2] *организацией контента в системах масштаба предприятия, а также проблемами сопровождения БД в социальных сетях, находящих практическое применение в коммерческих отраслях. Входит в Policy Group проекта Data Portability, где занимается подготовкой лицензионных соглашений и вопросами прав пользователя на владение данными. Помимо этого пишет о базах данных на сайте GigaOm.com и ведет блог http://tagschema.com. Является владельцем патента в области передачи сообщений через границы доверия (trust boundaries).*

---

1 IETF (Internet Engineering Task Force) – рабочая группа по развитию Интернета. – *Примеч. перев.*

2 *Фолксономия* (folksonomy) – «народная классификация» – выполняемая силами пользователей организация информационного контента посредством назначения специальных меток (тегов). Часто противопоставляется *таксономии* как принципу иерархической категоризации информации. – *Примеч. науч. ред.*

Снижайте неотъемлемую сложность, устраняйте второстепенную сложность

Нил Форд

Неотъемлемая сложность (*essential complexity*) представляет собой проблему, изначально присущую любой задаче. Например, задача координации воздушного движения в национальном масштабе является сложной сама по себе. Управляющая система должна отслеживать в реальном времени точное местоположение каждого самолета (включая высоту), его скорость, направление и место назначения, чтобы предотвратить столкновения в воздухе и на посадочных полосах. Необходимо также оперативно управлять расписаниями полетов, чтобы избежать заторов в аэропортах в постоянно меняющихся условиях – при резком изменении погоды все расписание приходится пересматривать.

С другой стороны, *второстепенная сложность* (*accidental complexity*) обусловлена теми задачами, которые, как нам кажется, необходимо решить для снижения неотъемлемой сложности. Примером второстепенной сложности может служить устаревшая система управления полетами, используемая по сей день. Система проектировалась для решения сложной проблемы управления движением тысяч самолетов, но это решение порождает свои собственные проблемы. Современные системы управления полетами настолько сложны, что само их обновление становится трудным, если не невозможным делом. Почти во всем мире управление полетами осуществляется по технологиям более чем 30-летней давности.

«Синдром второстепенной сложности» проявляется во многих инфраструктурах (framework) и фирменных «решениях». Инфраструктуры для решения узких, конкретных задач приносят реальную пользу; чрезмерно сложные инфраструктуры создают больше трудностей, чем устраняют.

Сложные решения привлекают разработчиков, как пламя привлекает мотыльков, причем часто с тем же результатом. Решать сложные задачи интересно, а работа программиста по сути состоит из сплошного разгадывания головоломок. Кто не испытывал азарта при решении какой-нибудь невероятно сложной задачи? Однако в крупномасштабных проектах бывает очень трудно избежать второстепенной сложности, сосредоточившись на работе со сложностью неотъемлемой.

Как этого добиться? Отдавайте предпочтение инфраструктурам, созданным на базе работающего кода, а не в башнях из слоновой кости. Соотносите объем кода, направленного на решение непосредственной задачи, с объемом кода, который просто обслуживает взаимодействие пользователя с приложением. С осторожностью относитесь к решениям, продвигаемым фирмами-разработчиками: такие решения не всегда изначально плохи, но в них часто присутствует второстепенная сложность. Проверяйте соответствие решения поставленной задаче.

Обязанность архитектора – решать проблемы, лежащие в плоскости неотъемлемой сложности, без введения второстепенной сложности.

> *Нил Форд (Neal Ford) – архитектор программного обеспечения и «мемовод»*[1] *из ThoughtWorks, международного консалтингового агентства, специализирующегося на разработке и поставке комплексных решений. Он является создателем многих приложений, учебных материалов, компьютерных учебных курсов, видео/DVD-презентаций, а также автором и/или редактором пяти книг и многочисленных журнальных статей. Часто выступает на конференциях. Жгучее любопытство по поводу личности Нила можно утолить на сайте http://www.nealford.com.*

[1] Термин «мемовод» (meme wrangler) придумал сам Нил Форд. Он считает, что «...это гораздо лучше обесценившегося термина “архитектор”. Очень жаль, что из-за отсутствия отраслевой системы сертификации эту (номинально самую высокую) должность присваивают себе люди, полностью утратившие связь с реальностью». – *Примеч. перев.*

Возможно, ваша главная проблема не в технологиях

Марк Рэмм

Прямо сейчас где-то терпит бедствие очередной проект системы расчета зарплаты... А скорее всего, и не один.

Почему это случилось? Потому что разработчики выбрали Ruby вместо Java или Python вместо Smalltalk? Потому что решили использовать Postgres, а не Oracle? Или потому что предпочли платформу Windows, хотя следовало выбрать Linux? Как известно, во всех неудачах проектов обычно винят технологию. Но действительно ли ваша задача была настолько сложна, что возможностей Java оказалось для нее недостаточно?

Проекты обычно создаются людьми, и именно от этих людей зависит успех или провал всего проекта. А раз так, стоит немного подумать, как помочь им добиться успеха.

Возможно, в команде есть кто-то, кто, с вашей точки зрения, не справляется со своей работой и тем самым препятствует успеху проекта. Технология, применяемая для решения подобных проблем, очень стара и проверена временем; в сущности, это едва ли не самое важное техническое достижение в истории человечества. То, что вам нужно, называется *общением*.

При этом простого владения приемами общения как технологией недостаточно. Уважительное отношение к людям, умение предоставлять им кредит доверия – важнейшие навыки, превращающие умного руководителя в эффективного.

Конечно, дело этим отнюдь не исчерпывается, но несколько простых советов существенно повысят эффективность вашего общения с подчиненными:

- Смотрите на обсуждение проблем как на конструктивный диалог, а не как на конфликтную ситуацию.

Исходите из позитивных предположений о людях и рассматривайте общение как возможность задать интересующие вас вопросы; так вы определенно сможете получить больше полезной информации, а ваши собеседники не займут оборонительную позицию.

- Приступайте к беседе только в подходящем для общения настроении.

  Если вы рассержены, раздражены, расстроены и вообще выведены из душевного равновесия, ваш собеседник, скорее всего, истолкует ваши невербальные проявления как признак нападения.

- Используйте такие ситуации как возможность достичь взаимной выгоды.

  Не говорите разработчику, чтобы он вел себя потише на собраниях, поскольку не дает никому говорить; лучше спросите, не поможет ли он вам вовлечь в обсуждение других людей. Объясните, что интровертам необходима более длинная пауза для вступления в разговор, и если он выдержит паузу в пять секунд перед произнесением первых слов, то тем самым очень поможет вам.

Если вы сосредоточены на общей цели, рассматриваете «проблемы» своих собеседников как возможность чему-то научиться, управляете своими эмоциями, то вы не только повысите свою эффективность, но и будете каждый раз узнавать что-то новое.

> *Марк Рэмм (Mark Ramm) – «великодушный пожизненный диктатор»*[1] *для TurboGears 2, страстный поклонник Python и, вообще говоря, совершенно сумасбродный парень. Он перепробовал все мыслимые и немыслимые виды деятельности – от архитектора программного обеспечения и сетевого администратора до ловца лобстеров и уборщика в баре для байкеров. Его основное увлечение – разработка инструментов, повышающих производительность труда программистов (как профессионалов, так и любителей).*

[1] «Великодушный пожизненный диктатор» (Benevolent Dictator For Life, BDFL) – шутливый термин в области разработки свободного ПО, которым именуют главу или основателя проекта, сохраняющего за собой право принимать окончательные решения (см. *http://ru.wikipedia.org/wiki/BDFL*). – *Примеч. перев.*

Общение – король, ясность и лидерство – его верные слуги

Марк Ричардс

Слишком часто архитекторы программного обеспечения обитают в башнях из слоновой кости, спуская оттуда разработчикам спецификации и навязывая им технологии и направления. Сплошь и рядом это приводит к раздорам, за которыми быстро следует «народное восстание». В итоге появляется программный продукт, который не имеет ничего общего с исходными требованиями. Каждый архитектор программного обеспечения должен уметь объяснять своим коллегам цели и задачи программного проекта. Ключами к эффективному общению являются ясность и лидерство.

Ясность характеризует процесс общения. Никто в вашей группе не станет читать 100-страничный документ с обоснованием ваших архитектурных решений. Умение четко и ясно выражать свои мысли жизненно важно для успеха любого программного проекта. С самого начала работы над проектом придерживайтесь простых объяснений и ни в коем случае не начинайте составлять длинные описания в Word. Используйте такие инструменты, как Visio, для создания простых диаграмм, поясняющих ваши идеи. Диаграммы должны быть простыми, потому что они почти наверняка будут часто изменяться. Еще одним эффективным средством распространения информации являются неформальные встречи. Лучший способ донести вашу мысль до участников проекта – собрать группу разработчиков (или других архитекторов) в одной комнате и изобразить свои идеи на доске. Обязательно захватите с собой цифровой фотоаппарат (ничто так не раздражает, как требование освободить конференц-зал, когда вы только что закончили рисовать). После собрания сделайте снимок, размножьте его и распространите среди остальных участников через вики. Отложите создание пространных документов и направьте усилия на то, чтобы донести свои идеи до коллег, а уже потом можно будет подумать и о более подробном изложении ваших архитектурных решений.

Большинству архитекторов программного обеспечения не приходит в голову, что они должны быть еще и *лидерами*. Чтобы работать в здоровой и продуктивной атмосфере, вы как лидер должны завоевать уважение своих коллег. Держать разработчиков в неведении относительно того, как выглядит общая картина и почему приняты те или иные конкретные решения, – верный путь к катастрофе. Напротив, привлекая разработчиков на свою сторону, вы формируете среду коллективной работы, в которой проверяется правильность ваших архитектурных решений. В свою очередь, подключение разработчиков к процессу создания архитектуры обеспечивает их приверженность и заинтересованность.

Действуйте совместно с разработчиками, а не против них. Учитывайте, что ясность и лидерство важны для взаимодействия со всеми участниками команды (не только с разработчиками, но и с группой тестирования, бизнес-аналитиками и руководителями проектов). Ясность в передаче информации и эффективное проявление лидерства улучшают качество коммуникации, формируют сильную и здоровую рабочую среду.

Если «общение – король», то ясность и лидерство – его верные слуги.

Марк Ричардс (Mark Richards) – директор и ведущий архитектор в фирме Collaborative Consulting, LLC, где он занимается разработкой архитектуры и проектированием крупномасштабных сервис-ориентированных решений на базе J2EE и других технологий главным образом в области финансовых операций. Работает в программной отрасли с 1984 года, обладает значительным опытом проектирования и разработки на J2EE, объектно-ориентированного проектирования и разработки, а также опытом системной интеграции.

Производительность приложения определяется его архитектурой

Рэнди Стаффорд

Производительность приложения определяется его архитектурой. На первый взгляд кажется, что это утверждение должно быть очевидным, но опыт реальной работы показывает обратное. Например, архитекторы программного обеспечения нередко полагают, что проблемы с производительностью приложения можно решить простым переходом на программную инфраструктуру от другого производителя. Источником этой веры может быть рекламная шумиха вокруг результатов тестирования – например, заявляется, что продукт фирмы-лидера на 25% превосходит по производительности ближайшего конкурента. Однако если продукт-лидер выполняет операцию за 3 миллисекунды, а конкурирующий продукт – за 4 миллисекунды, заявленные 25% (одна миллисекунда) значат очень мало на фоне общей низкой производительности, уходящей корнями в неэффективность архитектуры.

Помимо ИТ-менеджеров и команд тестирования производительности есть и другие группы людей, например служба поддержки фирмы-разработчика и авторы книг по управлению производительностью приложений, которые рекомендуют заняться тонкой настройкой инфраструктуры приложения: поиграть с операциями выделения памяти, размерами пулов подключений, размерами пулов потоков и так далее. Но если приложение спроектировано недостаточно эффективно для ожидаемой нагрузки или его функциональная архитектура нерационально использует вычислительные ресурсы, то никакая тонкая настройка не обеспечит желаемого быстродействия и масштабируемости. Потребуется полное перепроектирование внутренней логики и/или стратегии развертывания.

В конечном счете за фасадом продукта любого производителя и архитектуры любого приложения действуют одни и те же фундаментальные принципы распределенной обработки данных и физические закономерности: приложения

и используемые ими продукты выполняются как процессы на компьютерах ограниченной мощности, взаимодействуя друг с другом через стеки протоколов и каналы связи с ненулевыми задержками. А значит, людям следует понять и принять мысль о том, что архитектура приложения является главным фактором, определяющим его производительность и масштабируемость. Эти качественные характеристики невозможно улучшить как по волшебству – чудодейственной сменой технологий или тонкой настройкой инфраструктуры. Любые усовершенствования в этих областях требуют упорного труда по тщательной проработке архитектуры.

Рэнди Стаффорд (Randy Stafford) – профессионал в области создания программного обеспечения, обладающий 20-летним опытом работы в качестве разработчика, аналитика, архитектора, менеджера, консультанта и автора/лектора.

В настоящее время занимается разработкой промежуточного (middleware) программного обеспечения в составе группы A-Team фирмы Oracle, а также участвует в концептуальных проектах, занимается оценкой архитектуры и помогает устранять кризисы производства различным организациям во всем мире. Основные области его специализации – сервис-ориентированные архитектуры, производительность, высокая доступность и JEE/ORM.

Ищите истинный смысл требований

Эйнар Ландре

Заказчики и конечные пользователи часто под видом требования выдвигают то, что им кажется эффективным решением некоторой задачи. Классический пример такого рода приводит Гарри Хиллейкер (Harry Hillaker), ведущий конструктор истребителя F-16 Falcon. Перед его группой была поставлена цель спроектировать самолет, развивающий скорость M2–2,5, что было (и вероятно, остается) весьма нетривиальной задачей, особенно если сопутствующая цель состоит в создании «дешевого» легкого самолета. Учтите, что сила, необходимая для преодоления сопротивления воздуха, при удвоении скорости полета возрастает вчетверо, и представьте себе, как это обстоятельство влияет на вес самолета.

Когда конструкторская группа спросила заказчиков из ВВС, зачем им нужна скорость M2–2,5, те ответили: «Чтобы самолет мог при необходимости выйти из боя». Когда настоящая потребность стала очевидной, конструкторы смогли решить главную проблему и представить работоспособное решение: подвижный самолет с высокой тяговооруженностью, обеспечивающей хорошее ускорение и маневренность вместо высокой максимальной скорости.

Выводы из этого урока в равной степени справедливы и в области разработки программного обеспечения. Узнав у заказчика истинный смысл запрашиваемой возможности или требования, архитектор сможет проникнуть в суть проблемы и, если повезет, предложить решение более качественное и экономичное по сравнению с тем, на котором настаивает клиент. Повышенное внимание к сути требований упрощает и расстановку приоритетов: самые значимые для заказчика требования становятся определяющими.

Как же следует действовать? В манифесте гибкой (agile) разработки есть принцип, который ко многим случаям подходит очень хорошо: «Сотрудничество

важнее контракта». С практической точки зрения это подразумевает проведение семинаров и встреч, на которых архитекторы анализируют потребности заказчика, помогая ему ответить на вопрос «Почему?». Учтите, что поиск ответа на этот вопрос может оказаться весьма непростой задачей, потому что в процессе анализа требований речь зачастую идет о неявных допущениях и подразумеваемых знаниях. На таких встречах следует избегать дискуссий по поводу технических решений, потому что они переводят обсуждение из предметной области заказчика в область программирования.

Эйнар Ландре (Einar Landre) – профессионал в области программного обеспечения с 25-летним опытом работы в качестве разработчика, архитектора, менеджера, консультанта и автора/лектора.

В настоящее время работает в группе коммерческих приложений фирмы StatoilHydro, где занимается разработкой приложений, критических для бизнеса, оценкой архитектуры и оптимизацией процессов разработки. Основные сферы его интересов – сервис-ориентированные архитектуры, проектирование на основе предметной области (domain-driven design), применение мультиагентов и проектирование интенсивно используемых крупномасштабных сетевых систем.

Встаньте!

Уди Дахан

Для многих из нас карьера архитектора начиналась с какой-либо чисто технической должности, где успех определялся в основном способностью общаться с компьютерами. Однако в роли архитектора нам приходится общаться главным образом с другими людьми. Обсуждаете ли вы с разработчиками преимущества того или иного шаблона или объясняете руководству плюсы и минусы покупки промежуточного программного обеспечения, залогом успеха являются ваши навыки общения.

Объективно измерить степень влияния архитектора на проект довольно сложно, но одно совершенно ясно: если разработчики постоянно игнорируют указания архитектора, а руководство не придает значения его рекомендациям, «правильность» действий архитектора никак не повлияет на развитие его карьеры. Опытные архитекторы понимают, что они должны «продвигать» свои идеи, а эта задача требует умения эффективно общаться.

На тему межличностного общения написано множество книг, но я хотел бы предложить вашему вниманию один простой и практичный прием, который радикально повысит эффективность вашего общения и соответственно упрочит ваш успех в роли архитектора. В какой бы ситуации вам ни приходилось объяснять свою позицию сразу нескольким слушателям, встаньте. Неважно, где вы при этом находитесь – на официальном анализе проекта системы или неформальном обсуждении пары диаграмм. Встаньте – особенно если все остальные сидят.

Когда вы встаете, окружающие автоматически начинают воспринимать вас как авторитетного и уверенного в себе человека. Вы становитесь центром внимания. Вас будут реже перебивать. Все это окажет существенное влияние на обсуждение независимо от того, будут приняты ваши рекомендации или нет.

Также следует отметить, что стоящий человек более интенсивно использует жестикуляцию и мимику. Если вы общаетесь с группой из 10 и более человек, вставание поможет установить зрительный контакт со всей аудиторией. Зрительный контакт, жестикуляция, мимика и другие визуальные элементы играют важную роль в общении. Кроме того, положение стоя изменяет тональность и громкость голоса, а также темп речи: вы начинаете говорить так, чтобы вас было слышно в большом помещении, делая паузы для выделения важных моментов. Все эти элементы вносят существенный вклад в эффективность общения.

Хотите повысить эффективность передачи своих идей в процессе общения более чем вдвое? Это очень просто: встаньте.

Уди Дахан (Udi Dahan) – Шаман от Программирования; его заслуги были признаны корпорацией Microsoft и в течение трех лет оценивались престижным званием «Наиболее ценного специалиста» (Most Valuable Professional) в области архитектуры решений. В качестве советника по коммуникационным технологиям Уди работает с компанией Microsoft над WCF, WF и Oslo. Он также участвует в работе консультативных комитетов Microsoft Software Factories Initiative и проекта Prism группы Patterns & Practices. Он оказывает консультации в области современных архитектур и обучает клиентов по всему миру. Его основной специальностью является проектирование сервис-ориентированных, масштабируемых и безопасных архитектур на базе платформы .NET.

Сбои неизбежны

Майкл Найгард

Оборудование подвержено сбоям, поэтому мы вводим в наши системы избыточность. Она позволяет пережить отдельные сбои оборудования, но повышает вероятность того, что в любой момент времени в системе будет присутствовать хотя бы одна неисправность.

Программный код тоже подвержен сбоям. Наши приложения строятся на основе программного кода, а значит, они тоже уязвимы. Мы реализуем средства мониторинга, сообщающие о сбоях приложения, однако в основе этих средств тоже лежит программный код, а значит, они сами подвержены сбоям.

Люди тоже совершают ошибки, поэтому мы стараемся автоматизировать наши действия, диагностику и рабочие процессы. Автоматизация снижает вероятность ошибок, обусловленных нарушениями правил, но повышает вероятность ошибок, возникающих из-за отсутствия правил. Ни одна автоматизированная система не способна реагировать на такой спектр ситуаций, как человек.

Поэтому мы добавляем к средствам автоматизации механизмы мониторинга. Новое программное обеспечение, новые возможности для ошибок.

Сети строятся из оборудования, программного обеспечения и очень длинных линий связи. А значит, и сети подвержены сбоям. Даже если сеть работает нормально, ее поведение формально не прогнозируемо, потому что пространство состояний большой сети с практической точки зрения бесконечно. Отдельные компоненты могут обладать детерминированным поведением, но поведение их совокупности по сути своей хаотично.

Любой механизм безопасности, который мы применяем для устранения некоторой разновидности ошибок, вводит новые виды сбоев. Мы организуем кластеризацию, чтобы приложение автоматически переходило со сбойного

сервера на рабочий, но теперь при капризах кластерной сети возникает риск «расщепления мощностей»[1].

Стоит напомнить, что авария на Тримайл-Айленд[2] произошла в основном из-за клапана сброса давления – механизма безопасности, который должен был предотвращать некоторые виды сбоев, связанных с избыточным давлением.

Стало быть, сбои в системах в любом случае неизбежны. Так что же нам делать?

Осознайте один факт: независимо ни от чего в вашей системе будут происходить различные сбои. Отрицая их неизбежность, вы утрачиваете возможность контроля и изоляции этих сбоев. Но, приняв этот факт, вы сможете спроектировать реакцию своей системы на конкретные сбои. По аналогии с тем, как конструкторы автомобилей создают зоны деформации (области, деформирующиеся в первую очередь и гасящие энергию столкновения для пассивной защиты пассажиров), вы можете спроектировать защитные режимы сбоев, которые будут изолировать повреждения и защищать остальные компоненты системы.

Если этого не сделать, вам придется иметь дело со всеми непредсказуемыми – и обычно опасными – сбоями, возникающими в ходе работы системы.

> *Майкл Найгард (Michael Nygard) написал книгу «Release It! Design and Deploy Production-Ready Software» (Выпускаем в свет! Разработка и внедрение ПО, готового к выпуску) (Pragmatic Bookshelf), получившую премию Jolt Productivity в 2008 году. Другие его публикации можно найти по адресу http://www.michaelnygard.com/blog.*

---

1 Ситуация, при которой каждая сторона убеждена в полной неработоспособности другой стороны и продолжает захватывать ресурсы так, как если бы другой стороне никакие ресурсы не принадлежали. – *Примеч. перев.*

2 Крупнейшая авария в ядерной энергетике США, сопровождавшаяся частичным расплавлением активной зоны реактора. – *Примеч. перев.*

Вы ведете переговоры чаще, чем вам кажется

Майкл Найгард

Все мы попадали в «бюджетектурные» переделки, когда разумные технологические решения «хоронятся» ради экономии. Разговор проходит примерно так:

«Нам действительно так необходимы X?» – спрашивает спонсор проекта.

На место X можно подставить практически всё жизненно необходимое для работы системы: лицензии на программные продукты, избыточные серверы, внешние резервные копии или источники питания. Вопрос всегда задается отеческим тоном, словно вы спускаете все карманные деньги на комиксы и жвачку, тогда как серьезным взрослым людям нужно думать о покупке новых ведер, в которых они будут таскать свою будущую прибыль.

Правильный ответ на этот вопрос звучит так: «Да. Абсолютно необходимы». Но *так* почему-то почти никто не отвечает.

В конце концов, у нас техническое образование, а любая техническая дисциплина – это искусство компромисса. Понятно, что экзотика вроде источников питания никому не будет нужна, если поставить в центре обработки данных несколько беличьих колес и запустить в них практикантов. И вместо того чтобы сказать «Да, абсолютно необходимы», мы говорим что-то вроде: «Вообще-то без второго сервера можно обойтись, если вы согласны смириться с простоями из-за профилактики и при каждом сбое памяти. Хотя если мы купим память с автоматическим контролем четности, то и эту проблему можно обойти. Так что остаются только сбои операционной системы в среднем через каждые 3,9 дня, а значит, по ночам сервер придется перезагружать, но это вполне могут делать практиканты, когда устанут крутить колеса».

И все это может быть совершенно справедливо, но говорить так ни в коем случае не следует. Спонсор перестает вас слушать уже после слов «вообще-то».

Проблема в том, что вы рассматриваете происходящее с технической точки зрения, а ваш спонсор четко понимает, что он ведет переговоры. Вы занимаетесь совместным поиском решения, тогда как он проводит тактический маневр типа «выйдет/не выйдет». А в любых переговорах ни в коем случае не следует делать уступки по первому требованию. Правильный ответ на вопрос «Нам действительно так необходимы X?» звучит примерно так:

«Без второго сервера вся система будет «падать» примерно три раза в день, особенно в периоды максимальной нагрузки и при демонстрации на собрании совета директоров. На самом деле нам нужно четыре сервера, чтобы одна независимая пара обеспечивала сохранение 100-процентной работоспособности, даже если другая пара неожиданно перестанет работать».

Конечно, вы прекрасно знаете, что третий и четвертый серверы на самом деле не нужны. Это тактический гамбит, который заставит вашего спонсора перевести разговор на другую тему. Вы повышаете ставку и показываете, что система и так работает на минимальной, рискованной конфигурации. Кроме того, если вам каким-то чудом удастся получить дополнительные серверы, вы всегда можете передать один под тестирование (чтобы среда тестирования полностью соответствовала рабочей среде), а из другого получится отличная машина для сборки.

Биография автора приведена на стр. 31.

Используйте количественные критерии

Кейт Брайтуэйт

«Быстрый» не может быть требованием. Как и «обладающий хорошим временем отклика». Или, скажем, «расширяемый». Главная причина заключается в отсутствии объективных критериев выполнения таких требований. Но пользователям эти характеристики все равно нужны. Задача архитектора – позаботиться о том, чтобы система обладала необходимыми качествами, а также сбалансировать неизбежные противоречия, возникающие между ними. Без объективных критериев архитектор зависит от капризов заказчика («Нет, я не могу принять программу – она работает недостаточно быстро») и разработчиков, одержимых навязчивыми идеями («Нет, программа еще не готова – она работает недостаточно быстро»).

Обычно мы стараемся записать все подобные пожелания, как и любые другие требования. Но эта запись очень часто выглядит как набор туманных эпитетов: «гибкий», «удобный в сопровождении» и так далее. Однако все критерии такого рода (при достаточном усердии даже «удобство использования») могут измеряться в числовых величинах, для которых можно установить пороговые значения. Если этого не сделать, пользователи лишатся объективных оснований для принятия системы, разработчики потеряют полезные ориентиры, которыми они могут руководствоваться во время работы, а представление архитекторов о системе утратит четкость.

Задайте простые вопросы: сколько? в течение какого времени? как часто? как скоро? увеличивается или уменьшается? с какой скоростью? Если у вас нет ответов, значит, вы не понимаете, что нужно заказчику. Ответы должны находиться в экономической модели системы, и если их там нет, то вам предстоит основательно подумать. Если вы работаете над архитектурой системы, а заказчик не дал (или не дает) вам эти цифры, спросите себя, почему. А потом получите их. Когда в следующий раз вам кто-нибудь скажет, что система

должна быть «масштабируемой», спросите его, откуда возьмутся новые пользователи и почему. Спросите, сколько их будет и когда это произойдет. Не удовлетворяйтесь ответами «много» и «скоро».

Нечеткие количественные критерии должны задаваться в виде диапазона: минимум, норма, максимум. Если задать такой интервал невозможно, значит, непонятно, какое поведение требуется от системы. В ходе работы над архитектурой можно проверять систему на соответствие этим критериям, чтобы узнать, находится ли она (все еще) в пределах допустимых отклонений. Отклонения в степени соответствия некоторым критериям, происходящие с течением времени, дают полезную обратную связь. Определение этих интервалов и проверка системы на соответствие – дело трудоемкое и дорогостоящее. Если никто не беспокоится о *производительности* системы (ни о самой характеристике, ни о смысле термина) настолько, чтобы платить за тестирование производительности, скорее всего, этот показатель вообще не является существенным. Сосредоточьтесь при создании архитектуры на тех аспектах системы, которые стоят потраченных усилий.

«Система должна реагировать на входные данные пользователя не более чем за 1500 мс. При стандартной нагрузке (определяемой как...) среднее время отклика должно лежать в интервале от 750 до 1250 мс. Время отклика менее 500 мс не воспринимается пользователями, поэтому его падение ниже этого порога оплачиваться не будет.» *А вот это уже можно назвать требованием.*

Кейту Брайтуэйту (Keith Braithwaite) впервые заплатили за разработку программного обеспечения в 1996 году, хотя до этого он много лет занимался программированием на любительском уровне. После первого проекта – сопровождения компилятора, написанного с использованием lex и yacc, – он занялся сначала моделированием распространения микроволн для планирования сетей GSM, а затем расчетом на C++ сезонных колебаний спроса на воздушные перевозки. Перейдя на должность консультанта (и одновременно на язык Java), он познакомился с CORBA и EJB, а потом и с тем, что тогда называлось «электронной коммерцией». В настоящее время Кейт работает ведущим консультантом в фирме Zuhlke, где возглавляет Центр гибкой разработки.

Одна строка рабочего кода стоит 500 строк спецификации

Эллисон Рэндал

Проектирование – красивая штука. Систематическое, детальное представление пространства задачи и ее решения обнажает ошибки и выявляет возможности для совершенствования, причем иногда весьма радикальным образом. Спецификации играют в этом важную роль, поскольку они определяют шаблон для построения системы. Очень важно неспешно продумать всю архитектуру – как на макроуровне, рассматривая взаимодействие между компонентами, так и на микроуровне, вникая в поведение самих компонентов.

К сожалению, архитекторы часто увлекаются процессом проектирования, попадая под очарование архитектурных абстракций. Однако сами по себе спецификации не обладают никакой ценностью. Конечной целью программного проекта является реально работающая система. Архитектор всегда должен держать в поле зрения эту цель и помнить, что проектирование – всего лишь средство, а не конечный результат. Архитектор небоскреба, пренебрегающий законами физики ради изящества здания, обречен вскоре пожалеть об этом. Стоит упустить из вида конечную цель – рабочий код, – и у проекта начинаются серьезные неприятности.

Уважайте коллег, работающих над реализацией вашего видения системы. Прислушивайтесь к ним. Если у них возникают проблемы с вашим дизайном, вполне возможно, что они правы, а дизайн ошибочен или, по крайней мере, невнятен. В таких случаях привести дизайн в соответствие практическим требованиям – ваша непосредственная задача, и решить ее вы можете, общаясь с членами вашей команды, которые помогут вам определить, что работает, а что нет. Ни один дизайн не бывает идеальным с самого начала; любой дизайн подвержен изменениям по мере реализации.

Если вы участвуете в проекте и в качестве разработчика, научитесь ценить время, потраченное на написание кода, и не верьте тем, кто говорит, что это лишь отнимает время от работы по созданию архитектуры. Вы обретете гораздо более точное видение проекта на макро- и микроуровнях, после того как сами попробуете вдохнуть жизнь в свое творение.

Эллисон Рэндал (Allison Randal) – главный архитектор и ведущий разработчик проекта с открытым исходным кодом Parrot. За свою 25-летнюю карьеру она разрабатывала буквально все – от игр до средств лингвистического анализа, сайтов электронной коммерции, систем контроля доставки, компиляторов и систем репликации баз данных. Ей довелось побывать проектировщиком языков программирования, руководителем проектов, организатором конференций, редактором и консультантом. Эллисон была президентом фонда ПО с открытым кодом, написала две книги и основала издательство технической литературы.

Решений на все случаи жизни не существует

Рэнди Стаффорд

Архитектор должен непрерывно развивать и совершенствовать свое «контекстное чутье», поскольку единственного универсального решения для широкого круга разнородных проблем не существует.

Броское выражение «контекстное чутье» впервые использовал (с содержательным описанием его смысла) Эберхардт Рехтин (Eberhardt Rechtin) в своей книге «Systems Architecting: Creating & Building Complex Systems» (Системная архитектура: создание и построение сложных систем) (Prentice Hall, 1991):

> «Чтобы узнать основные принципы “эвристического подхода” к проектированию сложных систем, спросите опытного архитектора, что он делает, когда сталкивается с особенно сложной задачей. Его ответ, скорее всего, будет таким: «Просто использую здравый смысл». <...> Вместо термина «здравый смысл» лучше было бы использовать выражение «контекстное чутье»[1] – знание о том, что является разумным в данном контексте. Образование, полученный опыт и изучение примеров позволяют архитектору-практику набрать значительную мощь контекстного чутья к тому моменту, когда ему доверяется решение проблемы системного уровня, – обычно на это уходит *лет десять*.»

---

[1] Английское слово *sense* в зависимости от ситуации может переводиться и как «смысл», и как «чувство, чутье», поэтому в английском тексте перекличка терминов (*common sense* – «здравый смысл» и *contextual sense* – «контекстное чутье») выглядит более явной. – *Примеч. ред.*

Одна из серьезных проблем в индустрии разработки ПО, как мне кажется, состоит в том, что решение задач часто поручается людям, не успевшим развить достаточное контекстное чутье. Возможно, это связано с тем, что отрасль насчитывает всего два поколения и сейчас переживает стадию взрывного роста; не исключено, что исчезновение этой проблемы можно будет считать признаком зрелости отрасли.

Я часто сталкиваюсь с проявлениями этой проблемы в своей работе консультанта. Вот характерные примеры: отказ от проектирования на основе предметной области (domain-driven design)[1] там, где оно было бы уместным; утрата прагматического взгляда на вещи и чрезмерное увлечение проектированием программного решения, когда речь идет о задаче, не терпящей отлагательств; необоснованные или не имеющие отношения к делу предложения в тот момент, когда работы по оптимизации быстродействия системы заходят в тупик.

Самое важное, что необходимо знать о программных шаблонах, – то, когда их следует и когда не следует применять. То же самое верно в отношении гипотез о глубинных причинах проблемы и соответствующих корректирующих действий. В обеих ситуациях – при создании архитектуры системы и при анализе проблемы – универсального решения «на все случаи жизни» не существует по определению; архитектор должен развивать и тренировать свое контекстное чутье, формулируя архитектурные решения, а также выявляя и устраняя их недостатки.

Биография автора приведена на стр. 25.

[1] См. Эрик Эванс (Eric Evans) «Domain-Driven Design: Tackling Complexity in the Heart of Software» (Проектирование на основе предметной области: как совладать с присущей программам сложностью) (Addison-Wesley Professional).

Думать о производительности никогда не рано

Ребекка Парсонс

Потребности пользователей бизнес-приложений проявляются прежде всего в функциональных требованиях. Нефункциональные аспекты системы (такие как производительность, гибкость, время безотказной работы, потребности службы поддержки и т. п.) находятся в ведении архитектора. При этом предварительное тестирование нефункциональных требований зачастую откладывается до очень поздней стадии цикла разработки, а иногда полностью делегируется команде, обслуживающей систему.

Эта ошибка встречается намного чаще, чем следовало бы. В ее основе могут лежать различные причины. Забота о быстроте и гибкости программы, которая еще толком не выполняет требуемую функцию, может показаться лишенной смысла. Тестовые среды и сами тесты достаточно сложны. Возможно, ранние рабочие версии системы не подвергнутся реалистичной нагрузке в силу недостаточно интенсивного использования.

Однако, выпуская производительность из поля зрения до поздних стадий жизненного цикла проекта, вы теряете огромное количество информации о том, когда произошли изменения в производительности. Если производительность входит в число важных критериев, по которым будут оцениваться архитектура и дизайн системы, то тестирование производительности необходимо начать как можно раньше. Если вы используете методологию гибкой разработки с двухнедельными итерациями, то тестирование производительности, на мой взгляд, следует включить в процесс не позднее третьей итерации.

Почему это так важно? Прежде всего, вы как минимум будете знать о том, какие именно изменения привели к резкому падению производительности. Если в системе возникнут проблемы с производительностью, вам не придет-

ся анализировать всю архитектуру целиком – достаточно будет сосредоточиться на тех моментах, которые менялись недавно. Рано приступив к тестированию производительности и часто его выполняя, вы сузите круг изменений, которые следует подвергать анализу.

Даже если в ходе раннего тестирования вы не будете пытаться диагностировать производительность, вы по крайней мере получите набор базовых показателей, от которых сможете отталкиваться в дальнейшей работе. Впоследствии эта информация сыграет очень важную роль при поиске источника проблем с производительностью и их устранении.

Этот подход позволяет также проверять решения, принятые в ходе проектирования и работы над архитектурой системы, в контексте реальных требований к производительности. Если к системе предъявляются жесткие требования, такая ранняя проверка особенно важна для своевременной поставки готовой системы.

Хорошо известно, что организовать техническое тестирование – непростая задача. Настройка окружения, генерация наборов данных, определение необходимых тестовых сценариев (test case) – все это занимает много времени. Раннее тестирование производительности способствует поэтапному формированию тестовой среды, избавляя вас от существенно бóльших затрат времени и усилий при обнаружении проблем с производительностью на более поздней стадии.

Доктор Ребекка Парсонс (Rebecca Parsons) – технический директор ThoughtWorks. Она обладает более чем 20-летним опытом разработки приложений в различных отраслях, от телекоммуникаций до новых интернет-сервисов. Ребекка имеет печатные публикации по языкам программирования и искусственному интеллекту, работала в ряде комитетов в области программирования и написала множество журнальных статей. У нее есть обширный опыт в области создания крупномасштабных распределенных объектных приложений и интеграции разнородных систем.

Создание архитектуры как искусство баланса

Рэнди Стаффорд

Соотнесите интересы сторон с техническими требованиями

Когда речь заходит о разработке архитектуры программного обеспечения, в первую очередь мы представляем себе классические технические операции: разбиение системы на модули, определение интерфейсов, распределение ответственности, применение шаблонов и оптимизация производительности. Кроме этого архитектор должен учитывать ряд других аспектов, в том числе вопросы безопасности, удобства использования, простоты сопровождения, управления выпуском, выбора параметров развертывания и т. п. Но все перечисленные технические и процедурные аспекты должны быть соотнесены с потребностями заинтересованных сторон. Принять во внимание эти интересы при анализе требований – отличный способ обеспечить полноту спецификаций требований для разрабатываемого продукта.

У всех вовлеченных в проект сторон есть интересы, затрагивающие как процесс разработки программного обеспечения, принятый в организации, так и организацию в целом. Именно анализ этих интересов формирует итоговый набор приоритетов для архитектора. Можно сказать, что создание архитектуры – это процесс балансировки приоритетов в краткосрочной и долгосрочной перспективах в рамках имеющегося контекста.

Для примера возьмем инженерно-технический отдел организации, предоставляющей услуги по разработке программного обеспечения. Скорее всего, у этой организации существуют определенные приоритеты: соблюдение контрактных обязательств, получение дохода, поддержание хорошей репутации среди клиентов, снижение затрат и создание ценных технических активов. Эти бизнес-приоритеты преобразуются в приоритеты отдела: обеспечить функциональность, корректность и «атрибуты качества» разрабатываемого

продукта, а также продуктивность команды разработки, устойчивость процесса разработки, возможность его аудита, адаптируемость и долговечность программных продуктов.

Работа архитектора состоит не в том, чтобы просто создать функциональный, качественный программный продукт для пользователей, а в том, чтобы сделать это с соблюдением баланса интересов других сторон – руководства компании (сокращение затрат), персонала, обслуживающего продукт (простота администрирования), будущих членов команды программистов (простота изучения и сопровождения), а также с учетом опыта, накопленного профессиональным сообществом архитекторов.

Архитектор может на короткое время сознательно сместить баланс в пользу одного из приоритетов, но в долгосрочной перспективе, чтобы работа была выполнена действительно хорошо, следует избегать каких-либо перекосов. При этом поддерживаемый баланс должен отвечать имеющемуся контексту с учетом таких факторов, как предполагаемая продолжительность жизненного цикла программного продукта, критичность продукта для организации, техническая и финансовая культура компании.

Говоря коротко, создание архитектуры программного продукта не ограничивается одними лишь техническими аспектами; в процессе работы необходимо также найти правильное соотношение технических требований и бизнес-требований всех заинтересованных сторон.

Биография автора приведена на стр. 25.

Сделать наспех и сбежать – преступление

Никлас Нильссон

Время близится к вечеру. Команда дружно корпит над новой функциональностью, запланированной для текущей итерации; кажется, даже воздух в комнате пульсирует в рабочем ритме. Однако Джон немного спешит: его ждет свидание. Впрочем, он успевает дописать свою часть кода, компилирует ее, регистрирует в системе управления исходным кодом – и поспешно уходит. Несколько минут спустя загорается «красный свет»: сборка приложения нарушена. У Джона не было времени на автоматизированные тесты, поэтому он поступил по принципу «сделать наспех и сбежать», из-за чего застопорилась работа всей команды.

Ситуация изменилась – рабочий ритм сбился. Теперь все знают, что при обновлении кода из системы контроля версий неработоспособный код окажется и на их локальных компьютерах, а поскольку для подготовки к предстоящей демонстрации команде предстоит интегрировать много кода, это становится серьезным препятствием. По сути дела, Джон поставил команде подножку – ведь интеграция станет возможна только после того, как кто-то потратит свое время на отмену его изменений.

Такая ситуация возникает до обидного часто. Сделать наспех и сбежать – преступление, поскольку в результате нарушается нормальный ход работы. Это печально распространенный среди разработчиков способ сэкономить немного времени лично для себя, в итоге потратив впустую чужое время, что служит проявлением прямого неуважения к другим людям. И все же это происходит повсеместно. Почему? Обычно потому, что полноценная сборка системы или проведение тестов занимает слишком много времени.

Здесь в игру вступаете вы – архитектор. Вы тратите массу усилий на создание гибкой архитектуры, обучаете разработчиков гибким методам разра-

ботки (таким как разработка через тестирование) и обеспечиваете наличие сервера для непрерывной интеграции. Кроме того, вам необходимо сформировать культуру, правила которой запрещают понапрасну расходовать чужое рабочее время. Для этого помимо прочего нужно позаботиться о создании качественной инфраструктуры автоматизированного тестирования, поскольку она способна изменить поведение разработчиков. Если тесты выполняются быстро, разработчики будут проводить их чаще, что уже само по себе хорошо, но кроме этого они не будут оставлять своим коллегам нерабочий код. Если тесты зависят от внешних систем или для их выполнения необходимы обращения к базе данных, измените их так, чтобы они могли выполняться локально с заглушками (или хотя бы с базой данных, хранящейся в памяти), и пусть на сборочном сервере эти тесты выполняются медленно. Не заставляйте людей дожидаться, пока компьютер выполнит свою работу, иначе они начинают искать лазейки, которые в результате создают проблемы для всех остальных.

Не жалейте времени на то, чтобы обеспечить быструю работу с системой. Это повышает эффективность труда, устраняет поводы для уклонения сотрудников от работы и в конечном итоге способствует ускорению процесса разработки. Создавайте суррогаты и симуляторы, устраняйте зависимости, делите систему на меньшие модули – делайте что угодно, чтобы у ваших коллег не было даже тени искушения последовать принципу «сделать наспех и сбежать».

Никлас Нильссон (Niclas Nilsson) – консультант и преподаватель в области разработки программного обеспечения, а также писатель, глубоко увлеченный искусством программирования, ценитель хорошей архитектуры и дизайна. Является одним из соучредителей factor10 и редактором материалов в сообществе архитектуров ПО на сайте InfoQ.

Решений может быть несколько

Кейт Брайтуэйт

Одна модель данных, один формат сообщений, один транспортный механизм (и вообще ровно один основной архитектурный компонент, политика, принцип и т. п.) не может одинаково хорошо обслуживать все аспекты деятельности коммерческой организации. Похоже, этот факт является бесконечным источником удивления и огорчения для создателей систем. В то же время это совершенно естественно: раз уж организация (слово «организация» здесь подчеркнуто жирной красной линией) достаточно велика, чтобы волноваться о том, как несколько различных таблиц счетов повлияют на систему в следующем десятилетии, она наверняка слишком велика и неоднородна, чтобы можно было обойтись одной таблицей счетов.

В технической области единообразие можно ввести принудительно. И для нас это весьма удобно. Однако в сфере бизнеса в игру врывается противоречивый, многогранный, неформальный, хлопотный реальный мир. Что еще хуже, бизнесу приходится иметь дело даже не с реальным миром, а с мнениями людей о тех или иных аспектах ситуаций в тех или иных частях этого мира. Можно, конечно, попытаться отнестись к этой сфере как к технической и внедрить единое решение в приказном порядке. Однако реальность неформально определяется как «то, что не исчезает, когда в него перестаешь верить» (Филип К. Дик), и по мере развития бизнеса проблемы всегда возвращаются. Так возникают команды, занимающиеся корпоративными данными и тому подобным, которые тратят все свое (очень дорогое) время на обуздание экзистенциального ужаса посредством жонглирования DTD[1].

---

[1] DTD (Document Type Definition) – определение типа документа – преамбула документа, определяющая в языках структурной разметки данных (SGML, XML) состав и структуру документа. – *Примеч. ред.*

Время отклика подобных систем обычно оказывается неудовлетворительным с точки зрения заказчика.

Почему бы не принять реальность непоследовательного мира и не допустить существование нескольких несогласованных, перекрывающихся представлений, служб, решений? Да, технический специалист в каждом из нас, услышав такое предложение, съеживается от ужаса. Мы сразу представляем себе кошмарные сценарии: несогласованные обновления, лишние затраты на сопровождение, клубки зависимостей, которыми приходится управлять... Но давайте позаимствуем полезный опыт из мира хранения данных. Витрины данных (data marts) часто денормализуются (в реляционном смысле), в них произвольно смешиваются импортированные и вычисленные значения, а представления данных сильно отличаются от представлений данных в исходных базах данных. И при этом от наличия у витрины данных нефункциональных свойств никакой катастрофы не происходит. На границе двух совершенно разных миров – как правило, операций с данными и аналитической обработки с характерными для них различиями в частоте обновления и выборки, в пропускной способности, в периодичности изменений структуры и, возможно, даже в объемах – находится ETL-процесс[1]. В этом ключ к задаче: достаточно сильные различия в нефункциональных свойствах системы формируют границу, через которую удается организовать практическое управление несогласованными представлениями.

Конечно, не стоит дублировать представления или создавать альтернативные транспортные механизмы просто ради развлечения. Но вы всегда должны иметь в виду то, что декомпозиция системы по нефункциональным параметрам способна открыть благоприятные возможности для создания разнородных решений в интересах ваших клиентов.

Биография автора приведена на стр. 35.

[1] ETL (Extract, Transform, Load – извлечение, преобразование, загрузка) – один из основных процессов в управлении хранилищами данных (см. *http://ru.wikipedia.org/wiki/ETL*). – *Примеч. ред.*

Всем заправляет бизнес

Дэйв Мурхед

В контексте разработки корпоративных программных приложений архитектор должен стать в компании своего рода «мостом» между деловым и техническим сообществами, представляя и защищая интересы обеих сторон и часто выступая в роли посредника между ними, но при этом позволяя бизнесу заправлять делами. Принимая решения в области технологий, архитектор должен руководствоваться бизнес-целями компании и окружающими ее реалиями.

Прежде чем браться за проект по разработке программного продукта, коммерческая компания обычно планирует и озвучивает желаемый показатель окупаемости инвестиций (ROI – Return on Investment). Архитектор должен принять этот показатель и вытекающие из него ограничения ценности создаваемого продукта для компании. Это поможет избежать таких технических решений, которые способны привести к перерасходу средств. Показатель окупаемости должен служить важным компонентом целевого контекста как при общении с руководством (в ходе поиска компромисса между ценностью той или иной функции и затратами на ее реализацию), так и при обсуждении технической архитектуры и реализации с командой разработчиков. В частности, перед командой разработки архитектор должен представлять интересы бизнеса, не соглашаясь на выбор технологии с неприемлемо высокими стоимостью лицензии и затратами на поддержку в фазе тестирования или реальной эксплуатации продукта.

Одна из сложных задач, возникающих, когда всем заправляет бизнес, – предоставлять достаточный объем сведений качественного характера о том, как движется разработка продукта, чтобы бизнес-руководство могло принимать обоснованные деловые решения. Здесь важнейшую роль играет прозрачность. Архитектор вместе с руководством проекта должен создать и усовершенствовать

средства получения регулярной, оперативной обратной связи. Этой цели можно достичь, применяя различные приемы бережливой разработки ПО (lean software development): большие, заметные диаграммы, непрерывная интеграция, частые выпуски рабочих версий продукта и их передача бизнес-руководству уже на самых ранних стадиях проекта.

Разработка программного обеспечения в существенной степени представляет собой деятельность по проектированию, что подразумевает наличие непрерывного процесса принятия решений вплоть до того момента, когда готовая система запускается в эксплуатацию. Для разработчиков совершенно естественно принимать много решений, но эти решения обычно не относятся к деловой стороне вопроса. Однако в той степени, в которой бизнес-руководство пренебрегает своими обязанностями, не задавая команде разработчиков направление работы, не отвечая на их вопросы и не принимая бизнес-решений, оно фактически передает принятие деловых решений самим разработчикам. Для текущих микрорешений, принимаемых разработчиками, архитектор должен предоставить макроконтекст, донося информацию о программной архитектуре и бизнес-целях и защищая их. Он должен также добиваться того, чтобы разработчики не принимали бизнес-решений. Принятие технических решений в отрыве от обязательств, ожиданий и реалий бизнеса (которые должны регулярно озвучиваться деловой стороной в процессе работы над проектом) сводится к умозрительным догадкам и часто приводит к неоправданным затратам дефицитных ресурсов.

В долгосрочной перспективе интересы команды разработки лучше всего соблюдаются тогда, когда у руля стоит бизнес.

Дэйв Мурхед (Dave Muirhead) – ветеран в области искусства программирования и бизнес-технологий, владелец и ведущий консультант Blue River Systems Group, LLC (BRSG) – расположенной в Денвере компании, оказывающей консультационные услуги в сфере бережливой разработки программного обеспечения и технических стратегий.

Простота лучше универсальности

Кевлин Хенни

Типичная проблема многих компонентных инфраструктур (framework), библиотек классов, базовых сервисов и прочего инфраструктурного кода заключается в том, что они проектируются с расчетом на универсальное применение, без привязки к конкретным приложениям. В результате мы получаем ошеломляющий набор возможностей и настроек, которые часто не используются вообще или используются не по назначению, а то и попросту оказываются бесполезными. Большинство разработчиков работает над конкретными системами, и стремление к неограниченной универсальности редко способно сослужить им хорошую службу. Лучший путь к универсальности пролегает через глубокое понимание известных конкретных примеров и анализ их сути с целью поиска фундаментального общего решения: простота как результат практического опыта, а не универсальность, опирающаяся на умозрительные догадки.

Приоритет простоты перед универсальностью помогает сделать выбор между двумя архитектурными альтернативами, равнозначными в остальных отношениях. Когда возможны два решения, отдавайте предпочтение более простому и ориентированному на конкретную потребность, а не более изощренному и претендующему на универсальность. Конечно, вполне может случиться (и не так уж редко случается), что более простое решение на практике окажется и более универсальным. Но, даже если этого не произойдет, легче изменить простое решение, когда вы хорошо представляете, что же вам требуется, нежели нужным образом адаптировать «универсальное» решение, которое, как назло, оказалось недостаточно универсальным.

Несмотря на благие намерения архитектора, многие решения, задуманные для универсального применения, в конечном итоге оказываются не пригодными ни для чего конкретного. Программные компоненты в первую очередь

должны проектироваться для определенной задачи, и они должны хорошо справляться с этой задачей. Эффективная универсальность рождается из понимания, а понимание ведет к упрощению.

Обобщение иногда позволяет привести задачу к более фундаментальному виду; в созданном решении воплощаются закономерности нескольких известных примеров – четкие, компактные и хорошо обоснованные. Однако чрезмерное обобщение само превращается в задачу, которая ведет в противоположном направлении и повышает сложность, вместо того чтобы сокращать ее. Стремление к абстрактным обобщениям часто порождает решения, не привязанные к реалиям фактической разработки. Такие обобщения опираются на предположения, которые позднее оказываются неверными, предлагают варианты выбора, которые позже оказываются невостребованными, и создают балласт, который потом трудно или невозможно удалить. В конечном итоге это лишь повышает второстепенную сложность, с которой придется столкнуться будущим разработчикам и архитекторам.

Многие архитекторы высоко ценят универсальность, но такое отношение не должно быть безусловным. Как правило, люди не готовы платить за универсальность (или не нуждаются в ней): перед ними обычно стоит совершенно конкретная задача, и они ценят конкретное решение этой задачи. Универсальность и гибкость могут проявиться при создании конкретных решений, но если при этом мы слишком быстро снимемся с якоря и забудем о конкретике, то начнем дрейфовать в море безграничных возможностей – море, полном хитроумных настроек, громоздких (не просто обширных) списков параметров, бескрайних интерфейсов и неправомерных абстракций. В стремлении к абстрактной гибкости часто (случайно или намеренно) утрачиваются ценные свойства альтернативных, более простых решений.

Кевлин Хенни (Kevlin Henney) – независимый консультант и преподаватель. Он специализируется на шаблонах и архитектуре, методах и языках программирования, процессах и практике разработки. Является соавтором книг «A Pattern Language for Distributed Computing» (Язык шаблонов для распределенных компьютерных систем) и «On Patterns and Pattern Languages» (О шаблонах и языках шаблонов) – обе книги опубликованы издательством Wiley.

Архитектор должен быть практиком

Джон Дэвис

Хороший архитектор должен подавать личный пример другим. Он должен быть способен заменить любого члена своей команды и выполнить любую работу – от прокладки сети и настройки процесса сборки до написания модульных тестов и выполнения тестов производительности. Без хорошего понимания всего диапазона технологий архитектор мало чем отличается от обычного руководителя проекта. Члены команды могут обладать более глубокими знаниями в своих узких областях – это совершенно нормально, – но вряд ли они смогут доверять своему архитектору, если тот не разбирается в используемых технологиях. Как уже было сказано, архитектор – это интерфейс между технической командой и бизнесом, а значит, он должен понимать все технические аспекты, чтобы играть роль представителя команды перед бизнес-руководством, не обращаясь постоянно за помощью. Из тех же соображений архитектор должен понимать деловые аспекты организации, чтобы успешно привести разработчиков к цели – удовлетворению коммерческих интересов компании.

Работа архитектора сродни работе пилота самолета: он может не выглядеть занятым, но в действительности использует десятилетия накопленного опыта для постоянного наблюдения за ситуацией и немедленно вмешивается при возникновении нештатной ситуации. Руководитель проекта (второй пилот) обеспечивает повседневное управление, избавляя архитектора от рутины и управления персоналом. В конечном итоге архитектор должен отвечать за качество продукта и его своевременную сдачу. Эти задачи трудно решить без личного авторитета, который играет чрезвычайно важную роль в успехе любого проекта.

Лучший способ обучаться – наблюдать за другими; именно так мы учимся в детстве. Хороший архитектор должен уметь выявить проблему, собрать

команду и, не занимаясь поисками виновных, объяснить суть проблемы, а затем предложить элегантное решение или обходной путь. При этом архитектор может попросить у команды помощи, нисколько не теряя авторитета. Разработчики должны ощущать свой вклад в решение задачи, но архитектор при этом направляет ход обсуждения и определяет правильный подход.

Архитекторам следует присоединяться к команде уже на самых ранних стадиях проекта; они должны не сидеть в башне из слоновой кости, указывая оттуда путь вперед, а работать «в поле» вместе со всеми остальными. Вопросы выбора стратегического направления или технологии не следует превращать в самостоятельные исследования или новые проекты – к ним надо подходить прагматически, разыскивая ответ в ходе практической работы или обращаясь за советом к коллегам-архитекторам (все хорошие архитекторы знают друг друга).

Хороший архитектор обязан на уровне эксперта владеть как минимум одним из инструментов своей профессии, например интегрированной средой разработки (IDE); помните, что архитектор должен быть практиком. Вполне логично, что архитектору ПО следует хорошо знать IDE, архитектору баз данных – инструментарий построения диаграмм «сущность–связь» (ER-диаграмм), а информационному архитектору – инструменты XML-моделирования. Однако ведущий архитектор обязан уметь применять инструменты всех уровней, от контроля сетевого трафика с использованием Wireshark до моделирования сложных финансовых сообщений в XMLSpy, – для него не существует слишком низких или слишком высоких уровней.

Как правило, архитектор приходит с хорошим резюме и впечатляющим прошлым. Этим обычно несложно произвести впечатление на руководство и технический персонал, но без демонстрации своих умений на практике он вряд ли сумеет завоевать уважение команды. В такой ситуации команда будет испытывать трудности с обучением, а ее члены вряд ли смогут справиться с той задачей, для решения которой их наняли.

Джон Дэвис (John Davies) в настоящее время является ведущим архитектором в фирме Revolution Money (США). Недавно основал новую компанию Incept5.

Обеспечьте непрерывную интеграцию

Дэвид Бартлетт

Сборка давно перестала играть роль «Большого взрыва» в разработке проектов. И архитекторы (как уровня приложения, так и корпоративного уровня) должны поощрять использование методов и инструментов непрерывной интеграции в каждом проекте.

Термин *непрерывная интеграция* (CI, Continuous Integration) впервые был предложен Мартином Фаулером в качестве шаблона проектирования. Он означает совокупность методов и инструментов, обеспечивающих регулярную автоматическую сборку и тестирование приложения через короткие промежутки времени (как правило, на интеграционном сервере, специально созданном для выполнения этих операций). Для любого современного программного проекта практика непрерывной интеграции, комбинирующая методы и инструменты модульного тестирования с инструментами автоматизированной сборки, становится обязательной.

Непрерывная интеграция воздействует на неотъемлемый элемент процесса разработки ПО – точку преобразования исходного кода в работающее приложение. В этой точке происходит объединение и тестирование составных частей проекта. Вам, вероятно, доводилось слышать принцип «Выполняйте сборку рано и часто» («Build early and often»); когда-то этот принцип служил методом снижения рисков, избавляя от неприятных сюрпризов в процессе разработки. В наши дни на смену «ранней и частой сборке» пришла непрерывная интеграция, которая также включает в себя сборку, но добавляет к ней возможности, улучшающие взаимодействие в команде разработчиков и повышающие ее координацию.

Самой известной частью практики непрерывной интеграции является сборка, которая обычно автоматизирована. Возможность ручной сборки остается,

но можно автоматически запускать сборку каждую ночь или при внесении изменений в исходный код. При запуске процесса сборки из репозитория извлекается последняя версия исходного кода; инструмент непрерывной интеграции пытается собрать проект и затем протестировать его. Процесс завершается рассылкой уведомлений с описанием результатов. Уведомления могут рассылаться в разных форматах, в том числе по электронной почте или через системы обмена мгновенными сообщениями.

Непрерывная интеграция делает процесс разработки более стабильным и целенаправленным. Несомненно, вам как архитектору это придется по душе, но еще важнее другое: непрерывная интеграция повысит эффективность вашей компании и команд разработки.

Дэйв Бартлетт (Dave Bartlett) – увлеченный своим делом профессионал. За 25 с лишним лет он успел побывать программистом, разработчиком, архитектором, руководителем, консультантом и преподавателем. В настоящее время он выполняет работы для клиентов Commotion Technologies, Inc. (частная консалтинговая компания), а также читает лекции в Высшей технической школе Пенсильванского университета. Его основные текущие проекты связаны с Федеральным резервным банком в Филадельфии, которому он помогает проектировать и создавать веб-приложения, порталы и комплексные приложения для взаимодействия с Федеральной резервной системой и Казначейством США.

Старайтесь не нарушать график

Норман Карновейл

Программный проект может потерпеть неудачу по многим причинам. Одним из самых распространенных источников провала проекта является изменение графика работ в ходе выполнения проекта без надлежащего планирования. Таких неудач можно избежать, но это требует значительных усилий со стороны множества людей. Корректировка временной шкалы или добавление в проект ресурсов обычно особых проблем не создает. Проблемы начинаются тогда, когда от вас требуют выполнить больший объем работы за тот же срок или сокращают график без снижения нагрузки.

Идея о том, что путем сокращения графика можно снизить расходы или ускорить сдачу продукта, является очень распространенным заблуждением. Обычно для более быстрой сдачи продукта или для расширения функциональности без изменения срока сдачи прибегают к сверхурочной работе или жертвуют «менее важными задачами» (такими как модульное тестирование). Избегайте такого сценария любой ценой. Напомните тем, кто требует от вас подобных мер, следующие факты:

- Сокращение сроков проектирования приводит к некачественному дизайну и плохой документации, а также повышает вероятность проблем с контролем качества и риск того, что система не будет принята пользователем.
- Сокращение сроков кодирования или сдачи **напрямую связано с количе**ством ошибок в конечном продукте.
- Сокращение сроков тестирования ведет к появлению плохо протестированного кода и напрямую связано с количеством проблем при тестировании.

- Все перечисленное влечет за собой проблемы на этапе эксплуатации, а устранение таких проблем обходится намного дороже.

В конечном итоге затраты не только не уменьшаются, но увеличиваются. Обычно именно здесь и кроется причина провала.

Вы как архитектор рано или поздно попадете в ситуацию, когда шансы на успех определяются тем, насколько быстро вы действуете. Выскажите свое мнение как можно раньше. Сначала попытайтесь обеспечить качество, сохранив изначально запланированный график. Если без сокращения графика не обойтись, попробуйте переместить некритичную функциональность в будущие версии. Разумеется, для этого потребуются хорошая подготовка, навыки ведения переговоров и умение убеждать в своей правоте. Начните оттачивать свое мастерство в этих областях прямо сегодня. Поверьте: вы об этом не пожалеете.

Норман Карновейл (Norman Carnovale) – IT-архитектор проектов, связанных с национальной безопасностью, выполняющий работу для Lockheed Martin Professional Services. Ранее работал консультантом в области программного обеспечения, преподавателем и архитектором в компании Davalen, LLC (http://www.davalen.com), которая является ведущим бизнес-партнером IBM и специализируется на проектах WebSphere Portlet Factory, WebSphere Portal и Lotus Domino.

Архитектурные компромиссы

Марк Ричардс

Каждый архитектор программного обеспечения должен знать и понимать, что нельзя получить все сразу. На практике невозможно спроектировать архитектуру, одновременно обладающую высокой производительностью, высокой доступностью, высоким уровнем безопасности и высоким уровнем абстракции. Существует одна реальная история, которую архитекторы программного обеспечения должны знать, понимать и рассказывать своим клиентам и коллегам. Я имею в виду историю корабля «Ваза».

В 1620 году шла война между Швецией и Польшей. Желая побыстрее завершить эту дорогостоящую войну, король Швеции приказал построить галеон, который назывался «Ваза». Это был необычный корабль. Требования к нему были не похожи на требования к любому другому кораблю того времени. Он должен был иметь длину более 60 метров, нести 64 пушки на двух батарейных палубах, а также брать на борт 300 солдат за раз для безопасной доставки в Польшу по морю. Время поджимало, денег не хватало (звучит знакомо, да?). Занимавшийся этим судном кораблестроитель еще никогда не проектировал такие корабли. Он специализировался на меньших, однопалубных судах. Тем не менее он взялся за проектирование и постройку «Вазы», полагаясь на свой прежний опыт. В итоге корабль, соответствующий этим спецификациям, был построен. Наступил день спуска на воду. Корабль гордо выплыл в гавань, отсалютовал из всех пушек... и вслед за тем затонул.

Проблема «Вазы» была очевидной; каждый, кто хоть раз видел палубу большого боевого корабля XVII века, знает, что палубы таких кораблей были переполнены и небезопасны, особенно во время сражения. Постройка корабля, который служил бы одновременно боевым и транспортным, было

дорогостоящей ошибкой. Кораблестроитель, стремясь выполнить все пожелания короля, создал неустойчивое и плохо сбалансированное судно.

Архитектор ПО может почерпнуть из этой истории много полезного и учесть этот печальный опыт при проектировании архитектуры программных продуктов. Попытка выполнить все требования сразу (как в случае с «Вазой») создает неустойчивую архитектуру, которая не решает толком ни одну из поставленных задач. Хороший пример такого рода – требование заставить сервис-ориентированную архитектуру (SOA, service-oriented architecture) заодно функционировать в качестве решения «точка–точка». Обычно это вынуждает обходить различные уровни абстракции, создаваемые подходом SOA, и в результате возникает архитектура, напоминающая блюдо спагетти из итальянского ресторана. В распоряжении архитектора имеется несколько инструментов для определения тех компромиссов, на которые приходится идти при проектировании архитектур. Два популярных метода такого рода – ATAM (Architecture Tradeoff Analysis Method) и CBAM (Cost Benefit Analysis Method). Дополнительную информацию об ATAM и CBAM можно найти на сайте SEI (Software Engineering Institute).

Биография автора приведена на стр. 23.

База данных как Крепость

Дэн Чак

В БАЗЕ ДАННЫХ ХРАНИТСЯ ВСЯ ИНФОРМАЦИЯ – как вводимая сотрудниками, так и полученная от клиентов. Пользовательские интерфейсы, бизнес-логика и прикладная логика (и даже сотрудники) приходят и уходят, а данные остаются. Трудно выразить словами то, насколько важно в первые же дни работы над проектом построить надежную модель данных.

Огромная популярность методологий гибкой разработки привела многих к мысли, что приложения можно (и даже предпочтительно!) проектировать по ходу работы. Эпоха предварительного написания сложных, исчерпывающих технических спецификаций ушла навсегда! Новая школа призывает поставлять продукт рано и часто. Одна строка эксплуатируемого кода полезнее 10 строк у вас в голове. Звучит слишком хорошо, чтобы быть правдой... во всяком случае в том, что касается данных.

В то время как бизнес-логика и пользовательские интерфейсы эволюционируют довольно быстро, структурам данных и их отношениям это обычно не свойственно. Следовательно, очень важно с самого начала четко определить модель данных как на структурном, так и на аналитическом уровнях. Миграция данных из одной схемы в другую «на месте» в лучшем случае сложна, всегда занимает много времени и часто чревата ошибками. Если с ошибками уровня приложения еще можно какое-то время мириться, ошибки в базе данных могут привести к катастрофическим последствиям. Даже если вы нашли и исправили ошибку проектирования на уровне данных, это не восстановит поврежденную информацию.

Надежная модель данных – это такая модель, которая гарантирует безопасность сегодняшних данных и может расширяться для данных завтрашних. Гарантировать безопасность означает обеспечить неуязвимость для ошибок, которые все равно (сколько бы усилий вы ни приложили) проникнут в вечно

изменяющийся прикладной уровень; поддерживать ссылочную целостность данных; задавать ограничения предметной области везде, где они известны; выбирать подходящие ключи, которые помогут обеспечить ссылочную целостность и соблюдение ограничений. Обеспечить расширяемость означает правильно нормализовать данные, чтобы модель данных можно было легко дополнить новыми архитектурными уровнями, не прибегая к использованию всевозможных лазеек и обходных путей.

База данных – последний страж, охраняющий ваши драгоценные данные. Прикладной уровень, эфемерный по своей природе, не может следить за собой сам. Для того чтобы база данных обеспечивала необходимую защиту, модель данных нужно спроектировать так, чтобы она отвергала неподходящие данные и не позволяла создавать отношения, не имеющие смысла. Ключи, отношения по внешнему ключу и ограничения предметной области, описанные в схеме, лаконичны, понятны, легко проверяются и в конечном итоге самодокументируемы. Правила предметной области, закодированные в модели данных, также имеют физический, долгосрочный характер; они не теряются при изменениях в логике приложения.

Чтобы извлечь из реляционной базы данных максимальную пользу, то есть сделать ее полноценной частью приложения, а не просто хранилищем данных приложения, необходимо с самого начала хорошо понимать, что же вы создаете. По мере развития вашего продукта будет совершенствоваться и уровень данных, но в каждой фазе развития он должен сохранять свой статус Крепости. И тогда вы сможете довериться ему и с уверенностью возложить на него ответственность за перехват ошибок с других уровней приложения – и он вас не разочарует.

Дэн Чак (Dan Chak) – директор по разработке ПО в CourseAdvisor Inc., компании Washington Post. Является автором книги «Enterprise Rails» (O'Reilly).

Руководствуйтесь неопределенностью

Кевлин Хенни

Столкнувшись с альтернативными вариантами, люди обычно полагают, что самое важное – сделать правильный выбор. В проектировании (программных продуктов или любом другом) это не так. Наличие альтернативы – признак того, что необходимо проанализировать неопределенность в дизайне системы. Используйте неопределенность как определяющий фактор для выявления тех мест, где можно отложить переход к деталям или применить разбиение и абстракцию, чтобы снизить важность проектировочных решений. Если вы жестко «прошьете» в системе первое же решение, которое пришло вам в голову, оно, скорее всего, в дальнейшем свяжет вам руки. В результате случайные решения начнут играть важную роль, а гибкость программного продукта снизится.

Одно из самых простых и конструктивных определений архитектуры дал Гради Буч (Grady Booch): «Любая архитектура является результатом проектирования, но не любое проектирование направлено на создание архитектуры. Архитектура служит представлением важных проектировочных решений, формирующих систему, причем важность здесь определяется стоимостью изменений». Отсюда следует, что эффективной является та архитектура, которая в общем случае снижает важность решений, принятых в ходе проектирования. Неэффективная архитектура эту важность увеличивает.

Если проектировочное решение может повести по одному из двух путей, архитектор должен отступить на шаг. Вместо того чтобы выбирать между вариантами А и Б, он должен подумать над другим вопросом: «Как спроектировать решение, чтобы снизить важность выбора между А и Б?». В этой ситуации интересен не выбор между А и Б, а сам факт существования этого выбора (а также то, что правильный выбор необязательно очевиден или неизменен).

Иногда архитектору приходится ходить кругами, прежде чем его озарит и он распознает возникшую дихотомию. Вы стоите у доски, обсуждая (весьма энергично) разные варианты с коллегой? Вы задумчиво бормочете «М-м-м» и «Э-э-э» над кодом, бесконечно перебирая возможные реализации? Когда новое требование или уточнение существующего требования ставит под сомнение разумность текущей реализации, перед вами неопределенность. Как реагировать на нее? Подумайте, как с помощью разделения или инкапсуляции изолировать принимаемое решение от кода, который в конечном итоге зависит от этого решения. Альтернативой этому часто становится невнятный код, который, словно нервничающий человек на собеседовании, бормочет что-то невразумительное и пытается компенсировать неуверенность множеством догадок и общих фраз. А если выбор делается с твердой, но необоснованной уверенностью, то проект на полной скорости и без оглядки сворачивает на неверный путь.

Часто возникает искушение принять «решение ради решения». В таких ситуациях вам поможет опционное мышление[1]. Если существует неопределенность относительно различных путей, по которым может пойти разработка системы, примите решение не принимать решение. Отложите конкретное решение до того момента, когда его можно будет принять более ответственно на основании реальных знаний, но не слишком надолго, чтобы не попасть в ситуацию, когда эти знания уже бесполезны.

Архитектура и процесс разработки тесно переплетены; это главная причина, по которой архитектор должен отдавать предпочтение таким циклам разработки и архитектурным подходам, которые имеют эмпирическую природу и обеспечивают обратную связь, чтобы конструктивно использовать неопределенность для деления на части самой системы и графика ее разработки.

Биография автора приведена на стр. 51.

---

[1] Опционное мышление (options thinking) – подход к оценке эффективности проектов с точки зрения концепции реальных опционов (см. *http://ru.wikipedia.org/wiki/Реальный_опцион*), представляющий собой поиск дополнительных возможностей, не учитываемых при традиционном анализе. Опционное мышление применимо к широкому спектру проектов: слияния и поглощения, банковское кредитование, инновационные проекты и т. п. – *Примеч. ред.*

Проблемы могут быть больше, чем их отражение в зеркале[1]

Дэйв Куик

Я РАБОТАЛ НАД СОТНЯМИ ПРОГРАММНЫХ ПРОЕКТОВ. В каждом из них встречались проблемы, которые создавали больше неприятностей, чем ожидала команда разработчиков. При этом часто происходило следующее: некоторые члены команды замечали такие проблемы рано, однако большинство сотрудников отвергало или игнорировало все симптомы, поскольку не понимало их важности, пока не становилось слишком поздно.

Это случается по разным причинам:

- Проблемы, которые кажутся тривиальными на ранней стадии проекта, становятся критическими тогда, когда их уже поздно исправлять. История про сварившуюся лягушку, вероятно, является преувеличением, однако она прекрасно иллюстрирует то, что творится во многих проектах.
- Некоторые сотрудники часто сталкиваются с сопротивлением в тех случаях, когда другие члены команды не обладают сходным опытом или знаниями. Для преодоления этого сопротивления необходимы исключительная смелость, уверенность и настойчивость, а подобными качествами редко обладают даже высокооплачиваемые опытные консультанты, нанятые специально для предотвращения таких проблем.
- Большинство разработчиков – оптимисты. Горький жизненный опыт учит нас умерять свой оптимизм, но неофиты склонны смотреть на мир оптимистично. Люди, от природы пессимистичные, в командах обычно непопулярны, даже если раз за разом оказываются правы. Мало кто

---

[1] Автор обыгрывает стандартное предупреждение, размещаемое на выпуклых автомобильных зеркалах заднего вида: «Предметы находятся ближе, чем их отражение в зеркале». – *Примеч. ред.*

захочет рисковать своей репутацией и пойдет против большинства без очень серьезных оснований. Многим из нас знакомо ощущение «не нравится мне все это, но не могу объяснить почему», однако оно редко становится действенным доводом в споре.

- У всех членов команды есть собственное мнение о том, что важно, а что нет. При этом их внимание сфокусировано на том, за что они отвечают лично, а не на целях проекта.
- У каждого из нас есть свои «слепые пятна» – слабости и недостатки, которые нам трудно осознать или принять.

Вот возможные стратегии противодействия этим факторам:

- Выработайте организованный подход к управлению рисками. Одно из самых простых решений – следить за рисками так же, как вы это делаете с программными ошибками. Кто угодно может выявить какой-либо риск, а затем каждый риск отслеживается до тех пор, пока не перестанет быть риском. При изменении статуса рисков или при появлении новых сведений риски пересматриваются и заново ранжируются. Это помогает устранить из обсуждений лишние эмоции и напоминает о необходимости периодической переоценки рисков.
- Выступая против большинства, подумайте, как помочь другим участникам команды понять вас. В любой команде, членом которой вы являетесь, поощряйте свободу мнений и старайтесь максимально спокойно обсуждать спорные темы.
- Предчувствия типа «не нравится мне все это» заслуживают пристального внимания. Если достоверных фактов еще нет, попробуйте придумать простейший способ проверки, который их предоставит.
- Постоянно соотносите свое видение системы с представлениями команды и заказчика. Такие средства, как ранжированный по приоритету список неформальных требований, полезны, но не заменяют регулярного общения с заказчиком и открытости мышления.
- Увидеть свои «слепые пятна» трудно по определению. Люди, от которых вы готовы услышать неприятную правду, когда она вам нужна, – ваш драгоценный ресурс.

Дэйв Куик (Dave Quick) – владелец, главный архитектор, уборщик и единственный работник Thoughtful Arts. Эта фирма разрабатывает программы для музыкантов и предоставляет консультации в области проектирования ПО компаниям, выпускающим программные продукты для создания музыки или произведений изобразительного искусства.

Повторное использование зависит не только от архитектуры

Джереми Мейер

Казалось бы, удачно спроектированная инфраструктура или хорошо продуманная и умно реализованная архитектура идеально подходит для повторного использования в вашей организации. Однако в действительности даже самые красивые и элегантные архитектуры, инфраструктуры или системы будут повторно использоваться только теми людьми, которые:

...знают об их существовании

Разработчики и проектировщики вашей организации должны знать о существовании архитектуры, инфраструктуры, библиотеки или фрагмента кода и о том, где можно найти всю необходимую информацию об этих элементах (документацию, данные о версиях и совместимости). Простая и логичная истина: люди не рассматривают возможности, о существовании которых не знают. Вы можете рассчитывать на повторное использование тех или иных элементов, только если информация о них активно распространяется.

Есть множество способов распространения информации о повторно используемых элементах в пределах организации – от вики-сайта с RSS-потоком, содержащим информацию об обновлениях (полезен в очень больших командах), до электронной почты с оповещениями об изменениях версий в репозитории исходного кода. В совсем маленькой команде проектировщик или ведущий разработчик может информировать своих коллег в личном разговоре или просто громким объявлением на весь офис. В общем, способ распространения информации о повторно используемых элементах может быть любым – главное, чтобы он был. Не оставляйте распространение информации на волю случая.

...умеют ими пользоваться

Умение повторно использовать элементы зависит от навыков и квалификации. Конечно, существуют люди, способные (по выражению Дональда Кнута (Donald Knuth)) «попадать в резонанс» с программированием и проектированием. Все мы работали с такими людьми – одаренными разработчиками и архитекторами, которые обладают впечатляющей (и даже пугающей) скоростью и глубиной усвоения информации. Но такие люди встречаются редко. Остальные члены команды, вероятно, просто хорошие, надежные, разумные разработчики и проектировщики. Их необходимо учить.

Может оказаться, что разработчики и проектировщики не знают тот конкретный шаблон, который использован при проектировании, или не в полной мере понимают модель наследования, выбранную проектировщиком инфраструктуры. Необходимо предоставить им простой доступ к информации в виде актуальной документации, а еще лучше – посредством обучения. Небольшие усилия, потраченные на обучение, способны хорошо подготовить команду к повторному использованию тех или иных элементов.

...считают, что это лучше, чем сделать заново самим

Многие люди – и особенно разработчики – предпочитают решать проблемы самостоятельно, вместо того чтобы обращаться за помощью. Спрашивать о том, как что-то работает, кажется им признаком слабости или даже проявлением невежества. Во многом это зависит от уровня зрелости и склада личности отдельных членов команды – для разных людей слова «лучше, чем сделать заново» означают разные вещи. «Молодые стрелки» всегда предпочитают писать нужный код самостоятельно, потому что это тешит их самолюбие. Более опытные участники чаще готовы принять тот факт, что кто-то другой уже размышлял над такой задачей и способен предложить что-то для ее решения.

Если ваша команда не знает, где искать элементы для повторного использования, или не умеет работать с ними, то, естественно, она самостоятельно создаст их заново. А платить по счетам придется вам.

Джереми Мейер (Jeremy Meyer) занимается проектированием и разработкой программного обеспечения, а также преподаванием в течение почти 20 лет. В настоящее время он является ведущим консультантом Borland Software в области моделирования и проектирования.

«Я» в архитектуре не существует

Дэйв Куик

На самом деле «я» в архитектуре, конечно, существует. Но это не заглавное «Я», привлекающее к себе внимание и доминирующее во всех обсуждениях. Строчной буквы вполне достаточно.

Как это относится к нам – архитекторам программных продуктов? Наше эго подчас оказывается нашим самым опасным врагом. Кто из нас не встречал архитекторов, которые:

- считают, что они понимают требования лучше заказчиков,
- рассматривают разработчиков как ресурс, нанятый для реализации их идей,
- в штыки воспринимают любые сомнения в своих идеях или игнорируют идеи других?

Подозреваю, что любой опытный архитектор когда-нибудь допускал хотя бы одну из этих ошибок. Я допускал их все и извлек болезненные уроки из своих неудач.

Почему это происходит?

- *Мы уже добивались успеха.* Успех и опыт развивают уверенность в себе и позволяют нам стать настоящими архитекторами. Успех ведет к более крупным проектам. Однако грань между уверенностью в себе и самонадеянностью весьма тонка. В какой-то момент проект выходит за рамки наших возможностей. Самонадеянность начинает проявляться тогда, когда мы пересекаем эту черту, но не хотим это признать.
- *Люди нас уважают.* Решение сложных вопросов проектирования создает кредит доверия – своего рода «страховочную сетку». Наша собственная нетерпимость, самонадеянность или стремление полагаться только

на личный опыт могут привести к тому, что мы упустим из вида важные вопросы проектирования.

- *Все мы люди.* Архитектор вкладывает в каждую архитектуру частицу самого себя. Критика ваших творений воспринимается как нападки лично на вас. Поддаться искушению и занять оборонительную позицию легко; научиться не делать это сложно. Гордиться своими достижениями легко; научиться без специальных усилий чувствовать пределы своих возможностей сложно.

Как этого избежать?

- *Требования не лгут.* При наличии корректных, полных требований хороша будет любая архитектура, которая им удовлетворяет. Тесно взаимодействуйте с заказчиком, чтобы и он, и вы правильно понимали смысл каждого требования для бизнеса. Архитектуру формируете не вы, а требования. Вы всего лишь должны приложить максимум усилий, чтобы обеспечить их выполнение.
- *Сосредоточьтесь на своей команде.* Члены команды – это не просто ресурс; это ваши помощники по проектированию и ваша «страховочная сетка». Из людей, которые чувствуют себя недооцененными, обычно получается плохая «подстраховка». Архитектура формируется командой, а не лично вами. Вы даете указания, но трудную работу выполняют все вместе. Их помощь нужна вам в не меньшей степени, чем им – ваша.
- *Проверяйте свою работу.* Модель – не архитектура, а всего лишь ваше понимание того, как должна работать архитектура. Работайте вместе с командой над созданием тестов, показывающих, как архитектура проекта поддерживает каждое из требований.
- *Наблюдайте за собой.* Большинству из нас приходится побеждать в себе природную склонность защищать свою работу, заботиться только о собственных эгоистических интересах и считать себя самым умным человеком в комнате. Под давлением эти склонности всплывают на поверхность. Уделяйте ежедневно несколько минут размышлениям о том, как происходило ваше взаимодействие с окружающими. Отнеслись ли вы к идеям всех своих собеседников с должным уважением и признательностью? Не было ли с вашей стороны отрицательной реакции на ценные предложения? Действительно ли вы понимаете, почему кто-то не согласился с вашим подходом?

Исключение «Я» из архитектуры не гарантирует успеха. Оно всего лишь устраняет распространенный источник ошибок, происходящих только по вашей вине.

Биография автора приведена на стр. 65.

Посмотрите с высоты 300 метров

Эрик Дорненбург

Нам, архитекторам, хочется знать, насколько хороша та программа, над которой мы работаем. Качество программы имеет очевидный внешний аспект – программа должна представлять ценность для пользователей, – но у него есть и более тонкий внутренний аспект, относящийся к ясности дизайна, – к тому, насколько легко нам понимать, сопровождать и расширять программный продукт. Если от нас настойчиво требуют определения качества, обычно мы в конце концов говорим: «Я узна́ю это, когда увижу». Но как можно увидеть качество?

Маленькие квадратики на архитектурных диаграммах представляют целые системы, а линии между ними могут обозначать все что угодно: зависимости, потоки данных или совместно используемые ресурсы (например, шину). Такие диаграммы изображают систему с высоты 10 километров, примерно в таком же масштабе, в котором мы видим ландшафт с самолета. Единственной альтернативой, как правило, является исходный код, который можно сравнить с видом на уровне земной поверхности. Оба представления не в силах передать сколько-нибудь существенной информации о качестве программного продукта: одно находится на слишком высоком уровне, а другое настолько перегружено деталями, что за ними не видно структуры. Очевидно, не хватает промежуточного варианта – взгляда с высоты 300 метров.

«Вид с высоты 300 метров» предоставляет информацию на нужном уровне. Он объединяет большие объемы данных и различные метрики (количество методов, количество зависимостей[1], цикломатическая слож-

[1] Количество зависимостей (class fan out) определяется количеством классов, на которых основана реализация данного класса. Квадрат этой величины определяет стоимость поддержки текущей функциональности. – *Примеч. науч. ред.*

ность[1]). То, как это будет представлено на практике, сильно зависит от конкретного аспекта качества. Это может быть визуальное представление графа зависимостей, гистограмма с отображением метрик на уровне классов или сложный полиметрический вид, показывающий связи различных входных значений.

Создавать такие представления вручную и поддерживать их синхронизацию с программой – безнадежное занятие. Нужны инструменты, которые способны создавать такие представления на основе единственного надежного источника информации – исходного кода. Для некоторых представлений, скажем для структурной матрицы решений (design structure matrix), существуют коммерческие инструменты, однако специализированные представления на удивление легко создаются посредством объединения небольших инструментов, извлекающих данные и метрики, с общими пакетами визуализации. Простой пример: вывод Checkstyle (по сути, набор метрик уровня классов и методов) загружается в электронную таблицу для построения диаграмм. Те же метрики могут отображаться и в формате TreeMap с использованием инструментария InfoViz. Отличным инструментом для отображения графов сложных зависимостей является также GraphViz.

Как только подходящее представление найдено, качество программного продукта начинает восприниматься менее субъективно. Появляется возможность сравнивать разрабатываемый продукт с другими подобными системами. Сравнение разных версий одной системы позволяет выявлять тенденции, а сравнение представлений различных подсистем способно выявить резкие отклонения от нормы. Даже с одной-единственной диаграммой мы можем положиться на свое эстетическое чувство и умение подмечать закономерности. Хорошо сбалансированное дерево, вероятно, отражает удачную иерархию классов; гармоничный набор прямоугольников может изображать код, организованный в виде классов правильного размера. В большинстве случаев работает весьма простой принцип: то, что хорошо выглядит, скорее всего, хорошо устроено.

Эрик Дорненбург (Erik Doernenburg) – технический директор компании ThoughtWorks, Inc.; он помогает клиентам в проектировании и реализации крупномасштабных систем уровня предприятия.

---

[1] Цикломатическая сложность (cyclomatic complexity) определяется путем подсчета условных и логических операторов, циклов, операторов switch, блоков обработки исключений и т. д. в телах всех методов для определения минимального числа путей выполнения программы. Этот показатель определяет сложность покрытия кода тестами. Значение этого показателя, превышающее 10 (в общем случае), означает срочную потребность в рефакторинге кода. – *Примеч. науч. ред.*

Пробуйте, прежде чем сделать выбор

Эрик Дорненбург

В процессе создания приложения приходится принимать много решений. Какие-то из них могут быть связаны с выбором инфраструктуры или библиотеки, другие относятся к использованию конкретных шаблонов проектирования. В любом случае ответственность за принятие решения обычно лежит на плечах архитектора. В расхожем представлении архитектор собирает всю доступную информацию, какое-то время размышляет над ней, а потом изрекает указания, которые должны быть воплощены разработчиками... Вряд ли вас удивит, что существует лучший способ.

В своем труде, посвященном бережливой разработке (lean development), Мэри и Том Поппендик (Mary и Tom Poppendieck) описывают методику принятия решений. Они считают, что принятие окончательного решения следует откладывать до самого ответственного момента – той точки, когда отсутствие у команды решения приведет к тому, что решение будет принято за нее, а бездействие повлечет за собой необратимые (или труднообратимые) последствия. И это разумно: ведь чем позднее принимается решение, тем больше информации для его принятия. Однако во многих случаях «больше информации» не равносильно «достаточно информации», а лучшие решения, как всем хорошо известно, принимаются «задним числом». Что это означает для хорошего архитектора?

Архитектор должен постоянно думать о том, какие решения ему придется принять в ближайшем будущем. Если команда состоит более чем из двух-трех разработчиков и в ней практикуется коллективное владение кодом, то при приближении к точке принятия решения архитектор может предложить нескольким разработчикам выбрать один из вариантов и некоторое время поработать в этом направлении. Когда наступает ответственный

момент, команда встречается для обсуждения преимуществ и недостатков разных решений.

Теперь, когда есть возможность оглянуться назад и посмотреть на последствия, лучшее решение задачи обычно для всех очевидно. По сути, архитектору не приходится принимать решение – он просто дирижирует процессом его принятия.

Этот подход применим как к простым, так и к серьезным задачам. С его помощью команда может решить, стоит ли использовать шаблоны Hibernate, предоставляемые инфраструктурой Spring, но с таким же успехом он поможет ответить на вопрос, какую инфраструктуру JavaScript следует выбрать для проекта. Разумеется, время созревания решения очень сильно зависит от сложности задачи.

Чтобы опробовать два и более решения одной задачи, нужно больше усилий, чем на реализацию априорно принятого решения. Однако априорный выбор чаще приводит к таким решениям, которые позднее оказываются неоптимальными, и перед архитектором возникает дилемма: отказываться от текущей реализации или оставлять ее со всеми вытекающими отсюда последствиями; оба варианта приводят к неэффективному расходованию ресурсов. Еще хуже, если никто из команды не осознает неоптимальность выбранного подхода из-за того, что не были изучены альтернативы. Тогда усилия расходуются впустую без каких-либо шансов осознать это. В конечном итоге опробование нескольких решений может оказаться наиболее экономичным вариантом.

Биография автора приведена на стр. 71.

Разберитесь в предметной области

Марк Ричардс

Эффективный архитектор программного обеспечения разбирается не только в технологиях, но и в предметной области решаемой задачи. Без этих познаний трудно понять деловой аспект задачи, цели и требования компании, а значит, трудно спроектировать эффективную архитектуру, отвечающую нуждам бизнеса.

Роль архитектора программного обеспечения состоит в том, чтобы понять задачу, стоящую перед бизнесом, бизнес-цели и бизнес-требования и преобразовать их в техническое архитектурное решение, удовлетворяющее им. Знание предметной области помогает архитектору решить, какие шаблоны следует применять, как планировать будущие расширения и как учесть тенденции развития отрасли, чтобы лучше подготовиться к изменениям. Например, для одних предметных областей (скажем, для страхового бизнеса) хорошо подходят сервис-ориентированные архитектурные решения (service-oriented architecture), а для других (например, для финансовых рынков) – архитектурные решения на основе бизнес-правил (workflow-based architecture). Знание предметной области помогает решить, какие архитектурные шаблоны лучше всего отвечают конкретным потребностям организации.

Знание отраслевых тенденций в конкретной области также помогает архитектору спроектировать эффективную архитектуру. Например, в страховом бизнесе набирает силу тенденция к автострахованию «по требованию», когда клиент оплачивает страховку только за время фактического вождения автомобиля. Страховка такого типа очень удобна, если вы оставляете свою машину в аэропорту утром в понедельник, отбываете к месту работы, возвращаетесь в пятницу и едете на машине домой. Знание подобных тенденций

позволяет планировать их поддержку в архитектуре, даже если компания, на которую вы работаете, еще не запланировала их в своей бизнес-модели.

Понимание целей бизнеса также помогает спроектировать эффективную архитектуру. Например, входит ли в состав целей той организации, на которую вы работаете, расширение посредством масштабных слияний и поглощений? Ответ на этот вопрос может повлиять на тип проектируемой архитектуры. Если ответ положительный, архитектура, возможно, должна содержать много уровней абстракции, чтобы облегчить слияние компонентов бизнес-логики. Если стратегия бизнеса предусматривает наращивание присутствия компании в Интернете с целью расширения доли рынка, вероятно, очень важным атрибутом будет высокая доступность. Архитектор обязан понимать цели компании, на которую он работает, и постоянно проверять, поддерживает ли архитектура эти цели.

Из всех известных мне архитекторов самыми успешными являются те, кто сочетает широкие практические познания в технической области с хорошим знанием конкретной предметной области. Они могут общаться с руководством и заказчиками на хорошо понятном им языке предметной области. Это, в свою очередь, дает заказчику чувство уверенности в том, что архитектор знает, что делает. Владение предметной областью позволяет архитектору лучше понимать задачи, проблемы, цели, данные и процессы – все те факторы, которые играют ключевую роль в проектировании эффективной архитектуры уровня предприятия.

Биография автора приведена на стр. 23.

Программирование – это часть процесса проектирования

Эйнар Ландре

Кристен Нигаард (Kristen Nygaard), отец объектно-ориентированного программирования и языка программирования Simula, говорил, что *программирование – это изучение*. Осознание того факта, что программирование, а точнее разработка программного обеспечения, является процессом изучения и творческого поиска, а не процессом производства и конструирования, имеет фундаментальное значение для совершенствования приемов разработки. Идеи из традиционных инженерных дисциплин в области разработки ПО не работают. Возникающие при этом проблемы документировались и анализировались ведущими мыслителями нашей области в течение более чем 30 лет. Например, в 1987 году Фредерик Брукс (Frederick Brooks, Jr.) в «Отчете оперативной группы Научного совета Министерства обороны по военному программному обеспечению» утверждал, что документно-ориентированный подход по принципу «сначала спецификация, потом разработка» лежит в основе многих проблем программного обеспечения.

Откуда же индустрия разработки программного обеспечения может черпать практические идеи для совершенствования своих методик? Как насчет отраслей производства сложных товаров массового производства: автомобилей, лекарств, полупроводников?

Возьмем автомобилестроение. Планирование новой модели начинается с выбора концепции или прототипа. Это в первую очередь механизм архитектурного позиционирования. BMW X6 – пример новой концепции, объединяющей свойства внедорожника и купе, которую BMW называет *sports activity coupe*. Прежде чем вы получили возможность приобрести новый X6, BMW потратила тысячи часов и миллионы долларов на проектирование машины и процесса ее производства. Когда BMW получает ваш заказ, одна из

сборочных линий переключается и выдает версию **X6, выполненную по вашему личному заказу.**

Чему можно научиться на примере этого сценария? Здесь важно то, что производство новой машины состоит из двух процессов. Первый из них – процесс творческого проектирования, включающий в себя установку необходимых сборочных линий. Второй – процесс производства машин в соответствии со спецификациями заказчика. Сходные во многих отношениях процессы присутствуют и в отрасли разработки программного обеспечения. Вопрос в том, какой смысл мы вкладываем в термины.

В своей статье «Как проектировать ПО?»[1] Джек Ривз **(Jack Reeves) высказывает** мнение, что единственным артефактом разработки ПО, действительно описывающим дизайн (в том смысле, как понимается и используется это понятие в классических инженерных дисциплинах), является исходный код. Собственно производство ПО автоматизировано и обеспечивается сценариями компиляции, сборки и тестирования.

Соглашаясь с тем, что создание исходного кода является частью проектирования, а не производства, мы получаем возможность применить полезные методы управления, работоспособность которых была проверена временем. Это методы управления творческой и непрогнозируемой работой, такой как разработка новой машины, нового лекарственного препарата или новой компьютерной игры. Речь идет о методах гибкого (agile) управления и бережливого (lean) производства (например, SCRUM). Эти методы направлены на то, чтобы максимизировать эффективность вложений с позиций ценности для потребителя.

Чтобы с выгодой использовать эти методы в области разработки ПО, необходимо запомнить: программирование является частью процесса проектирования, а не производства.

Биография автора приведена на стр. 27.

---

[1] Журнал «Компьютерра», опубликовавший русский перевод этой статьи и авторское послесловие к ней, сделал ее темой номера («Компьютерра», №17 (589), 5 мая 2005 г., *http://offline.computerra.ru/2005/589/*). – *Примеч. ред.*

Предоставьте разработчикам независимость

Филип Нельсон

Почти все архитекторы начинают свою карьеру как разработчики. У архитектора больше обязанностей, но в то же время он обладает б льшим влиянием в том, что касается конструкции системы. Возможно, в новой для вас роли архитектора вам будет трудно избавиться от некоторых привычек разработчика. Что еще хуже, у вас может возникнуть чувство, что вы должны постоянно контролировать разработчиков и их деятельность по реализации вашего дизайна. Однако для вашего успеха (и успеха вашей команды) очень важно предоставить всем коллегам достаточную независимость, которая позволит им продемонстрировать свои навыки и проявить творческие способности.

У разработчика редко находится время для того, чтобы откинуться в кресле и подумать над тем, насколько слаженной является работа системы в целом. В то же время все внимание архитектора должно быть сосредоточено именно на этом. Пока разработчики во весь опор создают классы, методы, тесты, интерфейсы и базы данных, вы следите за тем, как эти компоненты работают в сочетании друг с другом. Ищите «узкие места» и пытайтесь устранить их. У ваших людей возникают проблемы с написанием тестов? Улучшите интерфейсы и ограничьте зависимости. Непонятно, где абстракция действительно необходима, а где можно обойтись без нее? Добейтесь лучшего понимания предметной области. Не знаете, в каком порядке создавать компоненты системы? Составьте план проекта. Разработчики повторяют одни и те же ошибки при использовании спроектированного вами API? Сделайте дизайн более понятным. Разработчики плохо понимают ваш дизайн? Пообщайтесь с ними и все подробно разъясните. Вы сами плохо понимаете, где масштабирование уместно, а где нет? Поработайте с заказчиками и разберитесь в их бизнес-модели.

Если вы хорошо справляетесь с должностью архитектора, у вас попросту не останется времени на то, чтобы вмешиваться в дела разработчиков. Конечно, вам придется достаточно внимательно следить за тем, чтобы реализация вашего дизайна соответствовала задуманному. Однако для этого вовсе не обязательно стоять за спиной у других людей. Вы можете предложить свое решение, когда видите, что разработчики столкнулись с трудностями, но еще лучше создать такую среду, чтобы они сами пришли и обратились к вам за советом. Хороший архитектор искусно удерживает равновесие, обеспечивая успешность архитектуры и оставляя коллегам пространство для реализации их творческого и интеллектуального потенциалов.

Филип Нельсон (Philip Nelson) – технический специалист широкого профиля. На заре своей карьеры он имел дело с аппаратным обеспечением компьютеров, затем занялся сетями, системами и администрированием и в конечном итоге пришел к разработке программного обеспечения и архитектуры, обнаружив, что там-то и происходит самое интересное. Он работал над программными решениями для транспорта, финансов, производства, маркетинга и многих других отраслей, связанных с инфраструктурой.

Время меняет все

Филип Нельсон

Многие годы одним из самых ярких развлечений для меня было наблюдение за тем, что выжило, а что нет. Шаблоны, инфраструктуры, сдвиги парадигм, алгоритмы – их было так много, умные люди так страстно обсуждали их, думали о долгосрочных перспективах, старались сбалансировать все известные аспекты, но в конечном счете они ушли в небытие. Почему? Что история пытается сказать нам?

Выбирайте достойную задачу

Для архитектора программного обеспечения это довольно трудно. Ведь задачи и проблемы мы получаем от заказчика, и у нас нет такой роскоши, как выбор, верно? Все не так просто. Прежде всего, мы часто ошибочно считаем, что не можем повлиять на требования заказчика. Однако обычно это возможно, просто для этого нужно выйти из зоны комфорта в пространстве технологий. Неправильный выбор грозит нам встречей с драконами. Время течет, мы усердно работаем над поставленной задачей, но в конечном итоге весь наш труд оказывается напрасным: мы сделали не то, что требовалось, и вся работа идет прахом. Хорошее решение правильно выбранной задачи с большой вероятностью переживет все остальные решения.

Будьте проще

Мы часто говорим себе: будь проще[1]. Говорим – но не делаем. А не делаем потому, что это необязательно. Ведь мы такие умные, мы без труда управ-

---

[1] В оригинале здесь расшифровка хорошо известного англоязычным разработчикам акронима KISS – Keep It Simple, Stupid. – *Примеч. ред.*

ляемся с возросшей сложностью и легко оправдываем ее – потому что она делает архитектуру более гибкой, потому что такие решения кажутся нам более элегантными, потому что мы считаем, что способны предвидеть будущее. Время идет; на год или больше мы отходим от проекта... А когда возвращаемся, почти всегда недоумеваем, почему было сделано то, что сделано. Если бы сейчас мы делали все заново, то, скорее всего, приняли бы совсем другие решения. Эту шутку сыграло с нами время, представив нас глупцами в наших собственных глазах. Постарайтесь осознать это как можно раньше, преодолейте свою инерцию и попытайтесь понять, что же такое простота, способная выдержать проверку временем.

Будьте довольны сделанным

Архитекторы любят искать «единственный истинный путь» – методологию или философию, способную привнести столь желанную предсказуемость и открыть наконец те ясные ответы, которые, как им кажется, находятся где-то совсем рядом. Проблема в том, что ориентиры, которыми вы руководствуетесь сейчас, вряд ли останутся теми же через пару лет, не говоря уже о десятилетиях. Оглядываясь назад, вы всегда видите результаты, не соответствующие вашему нынешнему уровню притязаний. Научитесь принимать свои прежние достижения и боритесь с искушением попытаться вернуться и «исправить» их. Соответствовало ли решение поставленной задаче? Удовлетворяло ли оно требованиям? Используйте эти вопросы как критерии оценки – и у вас будет радостнее на душе.

Биография автора приведена на стр. 79.

«Архитектор программного обеспечения» пишется со строчной буквы

Барри Хокинс

В последнее время в области разработки ПО наметилась одна прискорбная тенденция: стремление присвоить архитектуре ПО профессиональный статус наравне с классической школой архитектуры. Похоже, она обусловлена желанием архитекторов утвердить свои успехи и достижения в глазах более широкой общественности, нежели круг коллег и непосредственных работодателей. Но ведь архитектура обрела профессиональный статус лишь в конце XIX века – спустя тысячелетия после своего зарождения. Некоторые архитекторы программного обеспечения определенно слишком торопятся в своем стремлении к признанию.

Создание архитектуры программного обеспечения – искусство, и для достижения успеха в этой области, бесспорно, необходимы практика и дисциплина. И все же разработка программного обеспечения находится на относительно ранней стадии развития. Мы пока не располагаем достаточной информацией об этой отрасли, чтобы обоснованно придать ей профессиональный статус. Несмотря на молодость индустрии разработки ПО, ее продукция ценится достаточно высоко, и услуги квалифицированных специалистов в этой области (а также тех, кто стремится выглядеть таковыми) оплачиваются на том же уровне, что и в ведущих профессиональных областях, включая медицину, бухгалтерский учет и юриспруденцию.

Специалисты-практики в области разработки ПО получают хорошую оплату за свою творческую, сопряженную с исследованиями работу. Плоды наших трудов использовались для достижения многих значимых результатов, и некоторые из них принесли пользу всему человечеству. Преградами на пути к признанию являются наши действительные достоинства и возмож-

ности; в областях, достигших профессиональной зрелости, действуют программы обучения и стажировки, заметно превосходящие аналоги из нашей отрасли. Поразмыслите над этим и поймите, сколько у нас причин довольствоваться текущим положением дел и какой дерзостью будет настаивать на том, что архитектор программного обеспечения должен стоять на одной ступени с Адвокатом, Доктором и Зодчим.

Должность «архитектор программного обеспечения» пишется со строчной буквы «а»; осознайте этот факт и примите его.

Барри Хокинс (Barry Hawkins) за свою 13-летнюю карьеру в области создания ПО испробовал различные амплуа – от разработчика-одиночки до руководителя команды и инструктора по методологиям гибкой разработки. В настоящее время Барри занимается преподаванием методологий гибкой разработки и проектирования на основе предметной области.

Масштаб – враг успеха

Дэйв Куик

Границы проекта характеризуют его масштаб. Сколько времени, усилий и ресурсов необходимо для его реализации? Какую функциональность и с каким уровнем качества требуется получить? Насколько сложно сдать продукт к заданному сроку? Какова степень риска? Какие имеются ограничения? Ответы на эти вопросы определяют границы проекта. Архитекторам программного обеспечения больше нравится тот вызов, который им бросают большие, сложные проекты. Потенциальные выгоды даже искушают людей искусственно раздувать размеры проекта для повышения его кажущейся важности. Однако расширение границ – враг успеха, потому что вероятность неудачи растет быстрее, чем можно ожидать. Увеличение масштаба проекта вдвое часто приводит к тому, что вероятность провала возрастает на порядок.

Почему так происходит? Рассмотрим несколько примеров.

- Интуиция подсказывает нам выделить вдвое больше времени или ресурсов, чтобы удвоить объем работы. Однако история показывает[1], что связь между ними не такая линейная, как утверждает интуиция. Например, в команде из четырех человек затраты времени на взаимодействия возрастают более чем вдвое по сравнению с командой из двух человек.
- Наши оценки – отнюдь не точная наука. Кто из нас не попадал в ситуацию, когда реализация какой-то функции оказывалась намного сложнее, чем предполагалось вначале.

Конечно, некоторые задачи изначально подразумевают выполнение проекта определенного масштаба и сложности. Написать текстовый редактор без возможности ввода текста несложно, но его вряд ли можно будет назвать

---

[1] См. книгу Фредерика Брукса «Мифический человеко-месяц, или как создаются программные системы» (СПб: Символ-Плюс, 2000).

текстовым редактором. Какие же стратегии помогают управлять границами в реальных проектах?

- *Узнайте реальные потребности.* Реализация проекта должна обеспечивать выполнение набора требований. Требования определяют функциональность или некоторые ее качества. Подвергайте сомнению любые требования, не описанные так, чтобы их ценность для заказчика можно было измерить. Если требование не имеет никакой практической значимости, зачем оно нужно?
- *«Разделяй и властвуй».* Ищите возможности разделить работу на меньшие независимые фрагменты. Управлять несколькими небольшими независимыми проектами проще, чем одним большим проектом с взаимосвязанными частями.
- *Назначайте приоритеты.* Мир бизнеса изменчив. В крупных проектах требования многократно меняются по ходу дела. Действительно важные требования обычно таковыми и остаются, как бы ни менялись экономические условия, тогда как прочие требования видоизменяются и даже исчезают. Система приоритетов позволяет реализовать самые важные требования в первую очередь.
- *Демонстрируйте результаты как можно скорее.* Люди редко понимают, что им нужно, пока не получат какой-нибудь результат. В известном комиксе[1] представлена эволюция проекта детских качелей: что сказал заказчик и как его требования поняли участники проекта, выполняющие те или иные роли. В итоге получается хитроумное сооружение, лишь отдаленно напоминающее качели. А на последнем рисунке под названием «Чего хотел заказчик» изображены простейшие качели из автомобильной покрышки на веревке. Когда у заказчика есть что-то, что он может самолично испытать, решение порой оказывается проще, чем предполагалось. При первоочередной реализации самых важных функций вы в качестве обратной связи получаете самую важную информацию на ранней стадии, когда она нужна больше всего.

Сторонники гибких методологий увещевают нас строить «самое простое, что будет работать»[2]. Неудачи проектов со сложной архитектурой происходят намного чаще, чем с простой. Сокращение границ проекта часто приводит к упрощению архитектуры – и это одна из самых эффективных стратегий, позволяющих архитектору повысить шансы успешного завершения проекта.

Биография автора приведена на стр. 65.

---

[1] См. *http://www.businessballs.com/treeswing.htm. – Примеч. перев.*

[2] См. книгу Кента Бека (Kent Beck) «eXtreme Programming eXplained: Embrace Change» (Addison-Wesley Professional, 2004).

Ответственное руководство важнее внешнего впечатления

Барри Хокинс

Когда архитектор приступает к проекту, у него появляется понятное желание «показать себя». Назначение на должность архитектора программного обеспечения обычно свидетельствует о доверии к технической компетентности специалиста со стороны компании; естественно, архитектор желает как можно скорее показать, что он заслуживает этого доверия. К сожалению, некоторые из нас ошибочно полагают, что для этого следует «представить себя во всей красе» – удивить, если не оглушить группу своей технической гениальностью.

Умение произвести впечатление на аудиторию играет важную роль в маркетинге, но отрицательно сказывается на руководстве программным проектом. Архитектор должен завоевать уважение своей команды ответственным подходом к руководству и хорошим пониманием технической и предметной областей решаемой задачи.

Ответственное руководство – вот достойная роль для архитектора. Архитектор должен действовать в интересах заказчика, а не потворствовать капризам своего эго.

Деятельность архитектора программного обеспечения направлена на удовлетворение потребностей пользователей чаще всего на основании указаний от тех, кто разбирается в предметной области лучше него. Успешная разработка ведет к поиску компромиссов между затратами/сложностью реализации и временем/рабочей силой, доступными для выполнения проекта.

Время и затрачиваемые усилия – ресурсы, принадлежащие компании; архитектор должен управлять ими добросовестно, без скрытых попыток реализовать собственные цели. Чрезмерно сложные системы на базе новомодных инфраструктур или технологий редко обходятся без дополнительных затрат со стороны компании. Архитектору, словно инвестиционному брокеру, доверены деньги клиентов – и при этом предполагается, что его работа обеспечит приемлемую окупаемость инвестиций.

Управлять со всей возможной ответственностью важнее, чем производить внешнее впечатление; никогда не забывайте, что вы распоряжаетесь деньгами других людей.

Биография автора приведена на стр. 83.

У программной архитектуры есть этические аспекты

Майкл Найгард

Этический аспект программного обеспечения становится очевидным, когда речь заходит о гражданских правах, хищении личных данных или вредоносных программах. Однако он проявляется и в менее экзотических обстоятельствах. Успешно работающие программы влияют на жизнь тысяч, а то и миллионов людей. Их влияние может быть как положительным, так и отрицательным. Программы способны улучшить или ухудшить нашу жизнь – пусть даже в незначительной степени.

Каждый раз, когда я принимаю решение относительно поведения программы, я в действительности решаю, что смогут и что не смогут делать ее пользователи. Причем принятое решение гораздо более жестко, чем законодательство: не существует апелляционного суда, в котором можно было бы опротестовать выбор обязательных для заполнения полей или строгую последовательность операций.

На эту проблему можно взглянуть и с точки зрения эффекта масштаба. Вспомните истории о последних интернет-червях или фильмах-блокбастерах. Наверняка вам встречались подсчеты того, сколько рабочих часов было потеряно из-за них в масштабах страны. Всегда можно найти какого-нибудь аналитика с оценками неслыханного ущерба, нанесенного чем-нибудь, способным отвлечь человека от рабочего стола. Мораль этой истории вовсе не в обличении математического невежества прессы и дешевой погони за сенсациями. В действительности речь здесь идет о том эффекте, который могут дать маленькие числа в большом масштабе.

Представьте себе, что вы должны принять решение относительно некоторой функции. Есть два варианта реализации: простой, требующий одного дня работы, и сложный, который отнимет неделю. В простом варианте

приходится вводить четыре новых обязательных поля, а в сложном программе хватит ума на то, чтобы обработать неполные данные. Какой способ вам следует выбрать?

Обязательные поля выглядят довольно безобидно, но с ними вы навязываете свою волю пользователям, заставляя их собирать дополнительную информацию до начала работы. А это часто означает, что данные придется записывать на бумажках, чтобы собрать их в одном месте и ввести в систему единовременно. Все это ведет к потере данных и задержкам, а также вызывает раздражение у пользователей.

Рассмотрим такую аналогию: допустим, мне нужно прикрепить на здание вывеску. Можно ли смонтировать ее на высоте 1,8 м, заставляя тем самым пешеходов нырять под нее или обходить ее стороной? Конечно, мне будет проще обойтись без лестниц и строительных лесов, при этом вывеска даже не будет полностью перекрывать движение. Я экономлю час на установке ценой двух секунд, которые я отнимаю у каждого пешехода, проходящего мимо моего магазина. С течением времени сумма этих двухсекундных потерь намного превысит сэкономленный мною час.

Неэтично усложнять жизнь другим (даже ненамного) только для того, чтобы немного упростить ее для себя. Успешные программы влияют на жизнь миллионов людей, но каждое принятое вами решение фактически навязывает вашу волю пользователям. Всегда помните о том, что ваши решения повлияют на жизнь этих людей. Будьте готовы взять на себя дополнительную ношу, чтобы снять часть груза с ваших пользователей.

Биография автора приведена на стр. 31.

Небоскребы не масштабируются

Майкл Найгард

Разработку программных продуктов часто сравнивают со строительством небоскребов, дамб и дорог. В некоторых важных аспектах это уместное сравнение.

Самая трудная часть строительства не проектирование здания, которое будет стоять на своем месте после завершения, а проработка процесса строительства. Этот процесс начнется с пустой площадки и завершится готовым зданием. За это время каждый рабочий должен иметь возможность применить свои профессиональные навыки, а частично возведенное строение должно оставаться устойчивым. Мы можем извлечь из этой аналогии полезный урок в том, что касается развертывания больших интегрированных систем. (А к категории «интегрированных» относятся практически все корпоративные и веб-приложения!) Традиционное развертывание по схеме «Большого взрыва» выглядит так, словно вы привезли на пустырь груду балок и брусьев, подбросили их в воздух и ожидаете, что они сами сложатся в форме здания.

Вместо этого следует планировать развертывание по одному компоненту. Такой подход обладает двумя заметными преимуществами как при замене существующей системы, так и при строительстве с нуля.

Во-первых, при развертывании программного продукта мы попадаем в зону кумулятивного технического риска, воплощенного в программном коде. При последовательном покомпонентном развертывании этот технический риск распределяется по более длительному промежутку времени. Каждый компонент получает самостоятельный шанс вызвать сбой при вводе в эксплуатацию, что позволяет нам доводить компоненты до ума по отдельности.

Во-вторых, последовательное развертывание заставляет нас создавать четко определенные интерфейсы между компонентами. Развертывание одного компонента новой системы часто подразумевает его обратную интеграцию со старой системой. Следовательно, к моменту завершения развертывания каждый компонент успеет поработать с двумя разными системами: исходной и замещающей. Тем самым покомпонентное развертывание автоматически улучшает возможность повторного использования компонентов. На практике это приводит также к увеличению сцепления (coherence) и уменьшению связанности (coupling) системы.

С другой стороны, в ряде важных аспектов аналогия со строительством не работает. В частности, материальность реального мира усиливает роль предварительного планирования, заставляя нас использовать метод «водопада». В конце концов, никто не начинает строить небоскреб, не зная заранее, сколько места он займет и сколько этажей в нем будет. Добавлять этажи к существующему зданию слишком дорого и рискованно, поэтому мы стараемся избегать таких крайностей. Местонахождение или высота единожды спроектированного небоскреба уже не должны изменяться. Небоскребы не масштабируются.

Мы не можем с легкостью расширить дорогу новыми полосами, но знаем, как легко включить в программу новые функции. Это не дефект процесса разработки, а достоинство той среды, в которой мы работаем. Ничто не мешает нам выпустить приложение с минимальной функциональностью, если пользователи достаточно ценят эти функции, чтобы заплатить за них. На практике чем раньше вы выпустите свое приложение, тем выше будет чистая стоимость итогового продукта.

На первый взгляд ранний выпуск противоречит подходу «пошагового развертывания», но в действительности они весьма хорошо сочетаются. Ранний выпуск отдельных компонентов означает, что каждый компонент может проходить итеративную разработку независимо от других компонентов. Более того, этот подход заставит вас проработать такие проблемные вопросы, как постоянная доступность в ходе развертывания и контроль версий протоколов.

Нечасто встречаются методы, обеспечивающие более высокую коммерческую ценность в сочетании с улучшением архитектурных качеств, но раннее развертывание отдельных компонентов обладает обоими достоинствами.

Биография автора приведена на стр. 31.

Неоднородность побеждает

Эдвард Гарсон

Естественная эволюция компьютерных технологий привела к важным изменениям в тех инструментах, которые используют архитекторы для создания компьютерных систем. Эти изменения воскресили интерес к многоязыковому программированию, то есть к использованию нескольких языков в качестве основных при реализации программной системы.

Концепция многоязыкового программирования не нова; характерным примером из прошлого служат системы, в которых клиентская часть написана на Visual Basic и использует серверную часть на базе объектов COM, написанных на C++. По сути дела эта архитектура эффективно использовала сильные стороны каждого из упомянутых языков в зените их популярности.

Какие же перемены возродили интерес к многоязыковому программированию?

Новые технические стандарты в сочетании с постоянным ростом ресурсов – пропускной способности каналов и вычислительных мощностей – сделали возможным реальное использование текстовых протоколов. Те времена, когда эффективные распределенные системы были возможны лишь при использовании хитроумных двоичных протоколов, остались в прошлом. Возможность удаленного взаимодействия на текстовом уровне появилась вместе с веб-службами на базе XML/SOAP и продолжает развиваться в направлении архитектурных стилей REST и других вспомогательных (но не менее важных) протоколов типа Atom и XMPP.

Благодаря этому новому поколению технологий у нас есть гораздо больше возможностей для гетерогенной разработки, чем когда-либо прежде, просто потому, что полезная информация теперь представляет собой отформатированный текст, который можно генерировать и обрабатывать универсальными средствами. Гетерогенная разработка позволяет для каждой конкретной

задачи выбирать наиболее подходящий инструмент, а способность систем взаимодействовать на текстовом уровне сметает многие прежние преграды.

Теперь архитекторы могут объединять специализированные мощные инструменты, позволяющие вести речь уже не о способности применить подходящий язык, а о способности применить правильную парадигму. Языки программирования поддерживают различные парадигмы, некоторые из которых объектно-ориентированные, а другие функциональные или ориентированные на параллельное программирование. Для конкретных задач или предметных областей одни из этих парадигм подойдут идеально, а другие окажутся несоответствующими. Однако в наши дни эти нетрадиционные (и на первый взгляд разнородные) инструменты объединяются в элегантные гибридные решения гораздо легче, чем прежде.

Эффект этих изменений уже виден в комбинаторном увеличении сложности архитектурной топологии современных программных систем. Этот аспект – не столько отражение присущей им разнородности, сколько признак новых возможностей.

Наличие выбора не всегда полезно, но в данном случае оно является «меньшим из зол» в контексте современных программных архитектур. Наша отрасль имеет дело с очень серьезными задачами[1], поэтому нам необходимы все возможности взаимодействия, которые только можно обеспечить, особенно если задействованные в настоящий момент платформы не очень хорошо подходят для решения этих задач[2].

В наши дни работа архитектора стала еще более сложной из-за того, что границы технологий трещат под напором новых возможностей. Примите этот вызов, учитесь мыслить широко и обратите богатство возможностей в свою пользу: неоднородность побеждает.

> *Эдвард Гарсон (Edward Garson) стал страстным энтузиастом программирования еще с тех времен, когда учился программировать на* ***Logo на компьютере Apple II. В настоящее время он работает архитектором программного обеспечения*** *в Центре гибких методологий программирования в Zuhlke Engineering – одной из ведущих швейцарских технологических компаний.*

---

[1] Надвигающаяся эпоха многоядерных процессоров вполне может оказаться самым серьезным вызовом, с которым когда-либо сталкивалось сообщество разработчиков.

[2] См. статью «The Free Lunch is Over» Герба Саттера (Herb Sutter) по адресу *http://www.gotw.ca/publications/concurrency-ddj.htm*.

Не забывайте о производительности

Крейг Рассел

Представьте себе автомобиль – просторный, удобный, экономичный, недорогой и утилизируемый на 98%. Хотите такой? Конечно. Кто угодно захочет. Ах, да, единственная проблема: его максимальная скорость составляет 10 км/ч. Не передумали? Этот маленький пример наглядно показывает, что производительность так же важна, как и любой другой критерий.

Многие архитекторы ставят производительность на последнее место в своих списках – возможно, потому, что компьютеры обрабатывают данные несопоставимо быстрее своих пользователей, и архитектору кажется, что скорость системы окажется приемлемой. А если современным системам не хватит быстродействия, обо всем позаботится закон Мура. Однако скорость работы оборудования – лишь один из аспектов системы.

Производительность иногда рассматривается просто как промежуток времени, который нужен системе, чтобы отреагировать на введенные пользователем данные. Но архитекторы систем должны учитывать разнообразные аспекты производительности, включая производительность труда аналитиков и программистов, реализующих дизайн, производительность общения пользователя с системой, а также быстродействие неинтерактивных компонентов.

Производительность труда людей, строящих систему, часто называется продуктивностью. Этот показатель весьма важен, поскольку он непосредственно отражается на затратах и графике проекта. Команда, сдающая проекты с большим опозданием и перерасходом средств, обречена на неприятные разговоры с руководством. Правильный выбор инструментов и готовых компонентов может радикально повлиять на то, как быстро система будет построена и начнет приносить прибыль.

Производительность взаимодействий с пользователем играет решающую роль в том, как пользователи примут систему. В этот аспект производительности вносят свой вклад многие факторы системной архитектуры. Самым очевидным примером, пожалуй, является время отклика, однако это далеко не единственный показатель. Не менее важны интуитивная понятность интерфейса и количество операций, которые должен выполнить пользователь, чтобы достичь своей цели; и то, и другое напрямую влияет на производительность.

Хорошая спецификация системы определяет не столько время отклика само по себе, сколько время выполнения задания. Оно определяется как промежуток времени, необходимый для решения задачи из конкретной предметной области, включая все взаимодействия пользователя с системой. Кроме времени отклика в эту характеристику входят время размышления оператора и время ввода данных оператором – величины, находящиеся вне сферы контроля системы. Однако включение этих метрик способствует качественному проектированию пользовательского интерфейса. Уделив внимание способу представления информации и количеству операций, необходимых для решения задачи, вы в конечном итоге улучшите производительность человека-оператора.

Производительность неинтерактивных компонентов не менее важна для успеха системы. Например, если на выполнение еженощно запускаемых пакетных заданий требуется более 24 **часов, система становится неработоспособной.** Производительность компонентов аварийного восстановления также относится к числу критических факторов. Сколько времени потребуется для восстановления работоспособности системы и продолжения нормальной работы, если одна из частей системы полностью вышла из строя?

Анализируя реализацию и рабочий режим успешной системы, архитекторы и проектировщики всегда должны обращать самое пристальное внимание на производительность.

*Крейг Рассел (Craig Russell) – практикующий архитектор программного обеспечения, специализирующийся на персистентности объектов (object persistence) **и распределенных системах. В настоящее время работает** ведущим специалистом в компании Sun Microsystems.*

Проектирование в пустоте

Майкл Найгард

Система состоит из взаимозависимых программных элементов. Организационная структура этих программных элементов вместе с объединяющими их отношениями называется *архитектурой*. На диаграммах, изображающих такие системы, отдельные программные элементы и серверы часто упрощенно представлены в виде прямоугольников, соединенных стрелками.

Одна маленькая стрелка может означать: «Синхронный запрос/ответ в формате SOAP-XML через HTTP». Для одного элемента диаграммы получается слишком много информации. Места для полной записи обычно не хватает, поэтому мы помечаем стрелку надписью «XML через HTTP» (с точки зрения внутренней реализации) или «Поиск по коду товара» (с точки зрения внешних пользователей).

Стрелка, связывающая программы, выглядит как непосредственный контакт, но в действительности им не является. Пустота между прямоугольниками заполнена аппаратными и программными компонентами. В частности, здесь могут находиться:

- Сетевые карты
- Сетевые коммутаторы
- Брандмауэры
- Системы обнаружения и предотвращения вторжений (IDS/IPS)
- Брокеры или очереди сообщений
- Механизмы преобразования XML
- Серверы FTP
- Зонные таблицы
- Кольца SoNET

- Шлюзы MPLS
- Магистральные линии
- Океаны
- Рыболовные траулеры, повреждающие донные кабели

Между программными элементами А и Б всегда находятся четыре-пять компьютеров, на которых работают программы коммутации пакетов, анализа трафика, маршрутизации, анализа угроз и т. п. И вы как архитектор, связывающий эти программные модули друг с другом, должны учитывать существование этой промежуточной среды.

Однажды я видел стрелку с надписью «Исполнение». Один сервер был установлен в компании моего клиента, второй находился совершенно в другом месте. Эта жизненно важная для клиента стрелка запускала цепочку событий, которая больше напоминала игру «Мышеловка», нежели единый интерфейс. Сообщения передавались брокерам, сохраняющим их в файлах, которые периодически запускаемыми заданиями загружались на FTP... и т. д. Этот «интерфейс» включал в себя более 20 этапов.

Необходимо хорошо понимать, какая статическая и динамическая нагрузка ложится на каждую стрелку. Рядом с «**SOAP-XML через HTTP» около стрелки** стоило бы написать также: «С каждым запросом HTTP принимается одно обращение, с каждым ответом HTTP возвращается один ответ. Ожидается до 100 запросов в секунду, ответ в 99,999% случаев должен выдаваться менее чем за 250 мс».

Но и это еще не все, что необходимо знать об этой стрелке:

- Что если вызывающая сторона обращается по ней слишком часто? Как должен действовать получатель – игнорировать запросы, вежливо отказывать или по возможности стараться их обработать?
- Что делать вызывающей стороне, если обработка запроса заняла более 250 мс? Повторить попытку? Немного подождать? Решить, что на стороне получателя произошел сбой, и продолжить работу без этой функции?
- Что произойдет, если вызывающая сторона отправила запрос по протоколу версии 1.0, а получила ответ в версии 1.1? А если вместо XML был получен код HTML? Или файл в формате MP3 вместо XML-файла?
- Что произойдет, если одна из сторон интерфейса станет временно недоступной?

Ответы на эти вопросы представляют собой суть проектирования «в пустом пространстве» диаграмм.

Биография автора приведена на стр. 31.

Изучите профессиональный жаргон

Марк Ричардс

В ЛЮБОЙ ПРОФЕССИИ СУЩЕСТВУЕТ СВОЙ ЖАРГОН, повышающий эффективность общения представителей этой профессии. Адвокаты говорят друг с другом о Habeas Corpus, Voir Dire и Venire, плотники – о соединениях встык и внахлест и о пропитках, а архитекторы ПО – о **ROA, двухэтапном представлении** и супертипах уровней ...Минуточку... о чем, простите?..

Очень важно, чтобы архитекторы ПО независимо от платформы, на которой они работают, располагали эффективными средствами общения друг с другом. Одним из таких средств являются архитектурные шаблоны и шаблоны проектирования. Чтобы эффективно работать в качестве архитектора, необходимо понимать базовые архитектурные шаблоны и шаблоны проектирования, различать их в коде, знать, когда они применяются, и уметь использовать их в разговоре с другими архитекторами и разработчиками.

Архитектурные шаблоны и шаблоны проектирования можно разделить на четыре основные категории: шаблоны корпоративного уровня, шаблоны уровня приложения, шаблоны интеграции и шаблоны проектирования. Эти категории обычно определяются уровнем абстракции шаблона в общей архитектуре. Шаблоны корпоративного уровня относятся к высокоуровневой архитектуре, тогда как шаблоны проектирования относятся к структуре и поведению отдельных компонентов общей архитектуры.

Шаблоны корпоративного уровня определяют «каркас» высокоуровневой архитектуры. К числу самых распространенных архитектурных шаблонов относятся архитектура, управляемая событиями (EDA, **Event-Driven Architecture**), сервис-ориентированная архитектура (SOA, **Service-Oriented Architecture**), ресурсно-ориентированная архитектура (ROA, Resource-Oriented Architecture) и конвейерная архитектура (pipeline architecture).

Шаблоны уровня приложения описывают дизайн приложения или подсистемы в контексте более крупной корпоративной архитектуры. К этой категории относятся хорошо известные шаблоны J2EE (например, Session Facade (фасад сессии) и Transfer Object (объект перемещения)), а также шаблоны, описанные в книге Мартина Фаулера (Martin Fowler) «Patterns of Enterprise Application Architecture»[1] (Addison-Wesley Professional).

Шаблоны интеграции играют важную роль в проектировании и описании концепций, связанных тем, как компоненты, приложения и подсистемы совместно используют информацию и функциональность. Вот некоторые примеры шаблонов интеграции: общий доступ к файлам, удаленные вызовы процедур, различные шаблоны передачи сообщений. Описания этих шаблонов см. *http://www.enterpriseintegrationpatterns.com/eaipatterns.html*.

Знание основных шаблонов из книги «банды четырех» «Design Patterns: Elements of Reusable Object-Oriented Software»[2] (Addison-Wesley Professional) обязательно для каждого архитектора. На первый взгляд может показаться, что эти шаблоны относятся к слишком низкому для архитектора программных продуктов уровню, однако они – часть лексикона, позволяющего архитекторам и разработчикам эффективно общаться.

Архитектор должен также знать и понимать суть различных антипаттернов. *Антипаттерны* (термин введен Эндрю Кенигом (Andrew Koenig)) представляют собой повторяемые процессы, приводящие к неэффективным результатам. Вот хорошо известные примеры антипаттернов: Analysis Paralysis (аналитический паралич), Design By Committee (разработка комитетом), Mushroom Management (управление грибами), Death March (путь камикадзе[3]). Знание этих шаблонов поможет вам избежать многих типичных ошибок. Список распространенных антипаттернов приведен по адресу *http://ru.wikipedia.org/wiki/Анти-паттерн*.

Архитекторы программного обеспечения должны уметь общаться друг с другом ясным, лаконичным и эффективным образом. На помощь приходят шаблоны; нам, архитекторам, остается лишь изучить и понять их.

Биография автора приведена на стр. 23.

---

[1] Фаулер М. и др. «Шаблоны корпоративных приложений». – М.: Вильямс, 2010.

[2] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – СПб: Питер, 2007. «Бандой четырех» часто называют группу авторов этой книги. – *Примеч. ред.*

[3] Название этого антипаттерна явно перекликается с книгой «Death March» (Йордон Э. «Путь камикадзе». – М.: Лори, 2008). – *Примеч. ред.*

Правила диктует контекст

Эдвард Гарсон

Мне видится здесь некоторая ирония: разговор об идеалах архитектуры я начинаю с заявления о том, что идеалов, по сути дела, не существует. Ну, а если их нет, то и писать, видимо, больше не о чем... Налицо явное противоречие, и попытки продолжать в том же духе могут, чего доброго, привести к гибели Вселенной или чему-нибудь еще в этом роде – как знать?

Но увы, *ceci n'est pas une pipe*[1].

Один из наиболее ценных уроков, которые я вынес из опыта работы архитектором ПО, таков: контекст решает, а простота помогает. В практическом смысле это означает, что контекст – единственная сила, превосходящая простоту при принятии архитектурных решений.

Говоря о *контексте*, я имею в виду не только непосредственные высокоуровневые показатели вроде ключевых бизнес-факторов, но и оказывающие косвенное влияние элементы (например, новые технологии и интеллектуальное лидерство в том или ином вопросе). Хороший архитектор умеет отслеживать несколько быстро движущихся целей.

Из чего состоит хорошая архитектура? Она является продуктом решений, которые принимаются в контексте противоречивых задач с различными приоритетами. Это означает, что самые важные решения иногда связаны не с тем, что следует включить в систему, а с тем, что из нее следует исключить. Ценность хорошей архитектуры заключается в умелом принятии решений (а ее продукт всего лишь отражает ваши намерения).

---

[1] «Это не трубка» (*фр.*). Подпись под курительной трубкой на картине сюрреалиста Рене Магритта «Вероломство образов». – *Примеч. перев.*

История предлагает нам немало интересных примеров влияния контекста на архитектуру. Мой любимый пример – выбор базы данных для программного обеспечения современного танка[1]. (Обычно выбор базы данных не имеет существенного значения для архитектуры системы; этот пример приводится лишь для пояснения.)

Когда пришло время выбирать базу данных, команда занялась изучением различных вариантов. Большинство баз данных оказалось способным обеспечить уровень производительности, необходимый для систем навигации и наведения на цель при быстром движении танка по пересеченной местности. Но потом команда неожиданно выяснила, что выстрел из орудия создает сильный электромагнитный импульс, который вызывает сбой бортовых систем – и, конечно, базы данных! На современном поле боя танк с неработающим программным обеспечением буквально теряет связь с внешним миром. В этом контексте решающим фактором при выборе базы данных стало время восстановления, и лучшими показателями в этом отношении обладала на тот момент InterBase, поэтому для танка M1 Abrams была в итоге выбрана именно эта база данных[2].

Итак, хотя технологические дебаты «X против Y» бушуют на многих форумах, все это пустая забава. Их основой часто служат отнюдь не принципиальные расхождения в техническом исполнении, а более тонкие отличия, которым пользователи отдают предпочтение при отсутствии «козырной карты» в виде определяющего контекста.

Ваша команда должна освободиться от стремления к идеалу. Начните с анализа контекста и используйте его как отправную точку для поиска самых простых решений.

Биография автора приведена на стр. 93.

---

1 Несмотря на свое в высшей степени сомнительное предназначение, танк остается чудом технической мысли.

2 Интересно отметить, что вследствие особенностей архитектуры InterBase база данных в процессе записи на диск всегда находится в согласованном состоянии. Это один из факторов, которые обусловливают столь быстрое восстановление базы при сбоях оборудования.

Гномы, эльфы, волшебники и короли

Эван Кофски

В романе Нила Стивенсона «Cryptonomicon»[1] Рэнди Уотерхауз излагает свою классификацию рас, с которыми ему доводилось встречаться. Гномы – трудяги, корпящие над произведениями искусства во мраке своих пещер. Они повелевают огромными силами, способными перемещать горы и преображать ландшафт, и славятся своим мастерством. Эльфы отличаются изысканностью и высочайшей культурой; они проводят свои дни за созданием новых прекрасных волшебных предметов. Они до такой степени талантливы, что даже не сознают, насколько сверхъестественными представляются эти предметы другим расам. Волшебники наделены огромной силой, но, в отличие от эльфов, много знают о магии, разбираются в ее природе и возможностях и применяют ее максимально действенно и эффектно. Однако существует и четвертый тип, о котором Уотерхауз упоминает кратко, не останавливаясь для подробного описания. Это короли – мудрые правители, которые знают, как управлять всеми прочими.

Архитектор в некотором роде относится к «королевскому роду». Он должен хорошо понимать все остальные «расы» и позаботиться о том, чтобы в архитектуре для всех них были выделены подходящие роли. Архитектура, спроектированная с учетом только одной «расы», привлечет к проекту представителей именно этой «расы». И даже если в составе команды будут самые умелые «гномы», или самые талантливые «эльфы», или самые могущественные «волшебники», однобокий подход к решению задач будет серьезно ограничивать возможности команды.

---

[1] Нил Стивенсон «Криптономикон» (АСТ, 2006, 2007).

Хороший король ведет все «расы» подданных к великой цели и помогает им трудиться бок о бок, чтобы достичь ее. Без великой цели у команды не будет в дения, а ее работа в конечном итоге превратится в партизанские вылазки. Без представителей всех «рас» команда сможет решать проблемы только одного типа и остановится у первой же преграды, не соответствующей этому решению.

Архитектор указывает цель, принимая в расчет особенности всех «рас». Тогда архитектура становится руководством по поиску таких задач, которыми смогут заниматься члены команды из разных «рас», пока будут больше узнавать друг о друге. Когда проект столкнется со сложностями, команда уже будет знать, как подходить к их преодолению, потому что архитектура дала команде возможность стать одним целым.

Эван Кофски (Evan Cofsky) – программист, музыкант-любитель и заядлый мотоциклист. Он увлекся музыкой и компьютерами в колледже и продолжает увлеченно заниматься ими до сих пор. В настоящее время является штатным экспертом по языку Python в компании Virgin Charter, где в должности ведущего разработчика возглавляет команду исключительно талантливых и разносторонних людей.

Учитесь у архитекторов зданий

Кейт Брайтуэйт

> *Архитектура – социальный акт и материальный театр человеческой активности.*
>
> *Спиро Костоф (Spiro Kostof)*

Сколько найдется архитекторов программного обеспечения, считающих свою роль исключительно (или в первую очередь) технической? Разве не должны они в действительности быть посредниками и арбитрами для воюющих фракций среди заинтересованных в проекте сторон? Сколькие из них рассматривают свою работу с чисто интеллектуальной точки зрения, не уделяя должного внимания человеческому фактору?

> *Архитектор становится великим благодаря не столько уму, сколько чистому, чуткому сердцу.*
>
> *Фрэнк Ллойд Райт (Frank Lloyd Wright)*

Что в большей степени присуще архитекторам вашей организации: необузданная интеллектуальная мощь и способность помнить мельчайшие технические детали или хороший вкус, изысканность и душевная щедрость? В какой атмосфере предпочли бы работать вы сами?

> *Врач может похоронить свою ошибку, архитектор – разве что обсадить стены плющом.*
>
> *Фрэнк Ллойд Райт*

Не является ли «сопровождение унаследованных систем» лишь подрезкой этого плюща? Хватит ли вам как архитектору силы духа уничтожить собственное неудачное творение? Или вы предпочтете просто прикрыть его? Райт сказал также, что лучшим другом архитектора является кувалда. А что вы разбили ею за последнее время?

> *Архитекторы не только верят, что сидят по правую руку от Бога, но и считают, что если Бог встанет, то они займут его место.*
>
> *Карен Мойер (Karen Moyer)*

Замените «Бог» на «заказчик».

> *В архитектуре, как и во всех остальных прикладных видах искусства, конечная цель управляет действием. Конечная цель – хорошо построенное здание. Такое здание обладает тремя свойствами: Удобство, Прочность и Эстетичность.*
>
> *Генри Уоттон (Henry Watton)*

Когда вам в последний раз попадался на глаза программный продукт, архитектура которого доставила вам эстетическое удовольствие? А вы сами стремитесь к тому, чтобы ваши работы были эстетичными?

> *Человек, не являющийся великим скульптором или художником, не может быть архитектором. Если он не скульптор, не художник, то сможет быть только строителем.*
>
> *Джон Раскин (John Ruskin)*

Занимает ли эстетика достойное место в вашей архитектуре? Движет ли вами в ходе объединения компонентов при построении систем художественный интерес к формам и текстурам, скульптурное чувство баланса и стремление к передаче движения либо ощущение важности отрицательного пространства?

И наконец, высказывание, не требующее комментариев, – верное средство от самого разрушительного синдрома в работе архитектора:

> *Это выглядит фантастическим парадоксом, но тем не менее является важнейшей истиной: абсолютно совершенная архитектура не может быть действительно великой.*
>
> *Джон Раскин*

Биография автора приведена на стр. 35.

Боритесь с повторениями

Никлас Нильссон

Приходится ли вашим разработчикам выполнять однообразные задания, над которыми почти не нужно думать? Попадаются ли в коде почти одинаковые фрагменты? Замечаете ли вы код, написанный методом «скопировать-вставить-изменить»? Если ответы положительные, то ваша команда работает медленнее, чем могла бы, и, как ни странно, причиной тому можете быть именно вы.

Прежде чем пускаться в объяснения, давайте договоримся о двух истинах, касающихся разработки программного обеспечения:

- Дублирование – это зло.
- Повторяющаяся работа замедляет разработку.

Вы как архитектор задаете тон. Вы лучше всех представляете себе систему в целом, и, скорее всего, именно вы задали направление работы, создав полный вертикальный срез системы – пример, который к настоящему моменту был неоднократно скопирован. Каждая операция копирования (копирует ли разработчик несколько строк кода, XML-файл или класс) свидетельствует о том, что в системе что-то можно упростить или выкинуть. Чаще всего копируется не логика предметной области, а инфраструктурный код, обеспечивающий ее работу. По этой причине очень важно, чтобы вы четко представляли себе возможный эффект от своих примеров. Любые фрагменты кода, любые примеры конфигурации в ваших примерах станут основой для десятков, сотен и тысяч других частей системы. Следовательно, вы обязаны следить за тем, чтобы ваш код был понятным, хорошо передавал ваши намерения и не содержал ничего, кроме самого существенного – логики предметной области. Как архитектор вы должны быть крайне чувствительны к любым повторам, потому что все, что вы пишете, будет повторено.

В вашей системе такого не происходит, правда? А теперь взгляните на этот конфигурационный файл: что потребует изменения, если применить его к другой части системы, а что останется прежним? Или возьмите типичный метод уровня бизнес-логики: нет ли в нем шаблона, который проявляется и в других методах с такими операциями, как обработка транзакций, логирование, аутентификация или аудит? А как насчет уровня доступа к данным? Часто ли встречаются одинаковые фрагменты, отличающиеся только именами сущностей и полей? Смотрите на вещи шире. Попадаются ли вам две-три строки кода, которые всегда идут рядом и воспринимаются как одно целое, хотя и работают с разными объектами? Все это примеры повторения. Со временем разработчики учатся отсеивать и игнорировать повторения при чтении кода, обращая внимание лишь на интересные отличия. Но даже при наличии такого навыка чтение кода тормозится. Такой код подходит для выполнения компьютером, но не для чтения разработчиками.

Ваша прямая обязанность – избавиться от повторений. Возможно, для этого вам придется освоить новую инфраструктуру, создать более совершенные абстракции, обратиться к специалистам для создания аспектной инфраструктуры или написать несколько небольших генераторов кода. И все же повторения никуда не исчезнут, пока кто-то специально не позаботится об этом.

И этот кто-то – вы.

Биография автора приведена на стр. 45.

Добро пожаловать в реальный мир

Грегор Хоп

Технари любят точность – особенно программисты, живущие в мире нулей и единиц. Они привыкли работать с двоичными решениями: один/ноль, истина/ложь, да/нет. Всё четко и последовательно, от неожиданностей защищают ограничения внешних ключей, атомарные транзакции и контрольные суммы.

К сожалению, реальный мир не столь упорядочен. Клиенты оформляют заказы, а через секунду отменяют их. Чеки отклоняются, письма теряются, платежи задерживаются, а обещания нарушаются. Ошибки ввода данных происходят чаще, чем нам хотелось бы. Пользователи предпочитают «плоские» («shallow») интерфейсы, которые предоставляют одновременный доступ ко многим функциям, а не длинный и узкий коридор «процесса» ввода данных, который проще реализуется и кажется многим разработчикам более «логичным». Ходом выполнения программы управляет уже не стек вызовов, а пользователь.

Распределенные системы только усугубляют положение, потому что в игру вступает уйма новых несогласованностей. Службы бывают недоступны, изменяются без предварительного уведомления и не предоставляют транзакционных гарантий. При выполнении приложения на тысячах компьютеров вопрос уже не в том, произойдет *ли* сбой, а в том, *когда* он произойдет. Такие системы обладают слабой связанностью (loosely coupled), асинхронностью и параллелизмом и не соответствуют традиционной транзакционной семантике... Не желаете принять синюю таблетку?!

Дивный новый мир компьютерных технологий трещит по швам – так что же нам делать? Как это часто бывает, осознание проблемы становится первым важным шагом к ее решению.

Распрощайтесь со старой доброй детерминированной архитектурой стека вызовов, в которой вы сами определяли, что, когда и в каком порядке происходит. Вместо этого будьте готовы реагировать на события в любое время и в любом порядке, восстанавливая состояние системы по мере необходимости. Выдавайте асинхронные запросы параллельно вместо последовательного вызова методов. Чтобы избежать полного хаоса, моделируйте свое приложение, используя управляемые событиями цепочки процессов или модели состояний. Нейтрализуйте ошибки посредством коррекции, повторных попыток или тестовых операций.

Звучит слишком устрашающе? Вы на такое не рассчитывали? К счастью, в реальном мире с похожими проблемами имеют дело уже давно: задержки писем, нарушенные обещания, перепутанные сообщения, платежи не на те счета – примеров не счесть. Соответственно люди вынуждены посылать письма заново, списывать некорректные заказы и игнорировать напоминания об уже произведенных платежах. Не вините реальный мир в своих проблемах – лучше используйте его, чтобы подсмотреть решения. В конце концов, Starbucks тоже не использует протокол двухфазной фиксации[1]. Добро пожаловать в реальный мир!

Грегор Хоп (Gregor Hohpe) – архитектор ПО в компании Google, Inc. Грегор является общепризнанным экспертом в области асинхронных архитектур для систем обмена сообщениями и сервис-ориентированных архитектур. Он является соавтором книги «Enterprise Integration Patterns»[2] (Addison-Wesley Professional), регулярно выступает на технических конференциях по всему миру.

---

1 См. *http://www.eaipatterns.com/ramblings/18_starbucks.html*.

2 Хоп Г., Вульф Б. «Шаблоны интеграции корпоративных приложений». – М.: Вильямс, 2007.

Не контролируйте – наблюдайте

Грегор Хоп

Современные системы являются распределенными и слабо связанными (loosely coupled). Построение слабо связанных систем создает немало хлопот, так почему же мы идем на это? Потому что хотим, чтобы наши системы были гибкими и не разваливались при малейших изменениях. Это критическое свойство в современных средах, где мы контролируем лишь небольшую часть своего приложения, а все остальное существует в виде распределенных служб или пакетов, находящихся под контролем других отделов или внешних производителей.

Итак, стремление создать систему гибкую и способную развиваться со временем – дело хорошее. Но это означает также, что наша система будет постепенно изменяться. Другими словами, «сегодня система уже не та, какой была вчера». К сожалению, это заметно усложняет документирование системы. Все знают, что документация устаревает в момент печати, но в постоянно изменяющейся системе дела могут обстоять еще хуже. Более того, построение гибкой системы обычно усложняет архитектуру, и получить пресловутую «общую картину» становится еще труднее. Например, если все компоненты системы обмениваются информацией по логическим, настраиваемым каналам, то, чтобы получить представление о происходящем, необходимо взглянуть на конфигурацию каналов. Отправка сообщений в логическое пойди-туда-не-знаю-куда вряд ли приведет к ошибке компиляции, но наверняка огорчит пользователя, действие которого было запаковано в это сообщение.

Архитектор, помешанный на тотальном контроле, – фигура из прошлого; решения, которые он создает, являются сильно связанными и хрупкими. С другой стороны, полная и неограниченная свобода приложения – верный путь к хаосу. Ослабление контроля необходимо дополнить другими механизмами, чтобы «полет по приборам» не происходил без самих приборов. Но

какие приборы есть в нашем распоряжении? Вообще-то их более чем достаточно. Современные языки программирования поддерживают рефлексию (reflection), и почти все платформы предоставляют динамические метрики времени выполнения. По мере того как система становится более настраиваемой, ее текущая конфигурация сама становится отличным источником информации. Так как разобраться в слишком большом объеме низкоуровневых данных довольно трудно, создайте на их основе модель. Например, когда станет ясно, какие компоненты отправляют сообщения по тем или иным логическим каналам и какие компоненты ожидают поступления сообщений по этим каналам, вы сможете построить граф передачи данных между компонентами. Эту процедуру можно производить каждые несколько минут или часов, создавая точную и оперативную картину системы в ходе ее развития. Считайте это своего рода «MDA наоборот»[1]: вместо модели, управляющей архитектурой, вы строите гибкую архитектуру и формируете модель на основании текущего состояния системы.

Во многих случаях модель можно легко визуализировать, построив действительно «общую картину». Однако боритесь с соблазном заполнить квадратиками и линиями полотно 3×5 метров в стремлении изобразить все классы вашей системы. Такая картина сойдет за произведение современного искусства, но ее не назовешь полезной программной моделью. Вместо этого лучше использовать «вид с высоты 300 метров», как рекомендует Эрик Дорненбург, – тот уровень абстракции, который содержит действительно полезную информацию. Вдобавок вы сможете проследить за тем, чтобы в модели не было нарушений простейших правил корректности – циклических ссылок в графе зависимостей или отправки сообщений по непрослушиваемым логическим каналам.

Ослабление контроля пугает, особенно когда речь идет об архитектуре системы. Но в сочетании с тщательным наблюдением, построением моделей и проверкой корректности оно, вероятно, является единственным разумным подходом к построению архитектуры программного обеспечения в XXI веке.

Биография автора приведена на стр. 109.

[1] MDA (Model-Driven Architecture) – архитектура, управляемая моделями.

Архитектор Янус

Дэвид Бартлетт

У древних римлян Янус считался богом начала и завершения, дверей и проходов. Обычно его изображали с двумя головами, направленными в разные стороны; этот символ часто встречается на монетах и в кино. Янус символизирует переходы и изменения в жизни – рождение, вступление в брак, достижение зрелости – от прошлого к будущему, от юности к старости.

Способность Януса смотреть вперед и назад, видеть прошлое и будущее является исключительно ценным навыком как для архитектора программного обеспечения, так и для архитектора-строителя. Архитектор стремится объединить реальность со своим представлением о ней, прошлые достижения – с планами на будущее, ожидания руководства – с ограничениями разработки. Наведение таких мостов – важнейшая часть работы архитектора. Часто у архитектора возникает чувство, что в своем стремлении довести проект до завершения ему приходится преодолевать пропасти, которые созданы воздействием сил, направленных в противоположные стороны. Это могут быть, например, простота в использовании и безопасность или соответствие текущим бизнес-процессам и ориентация на перспективу, обрисованную руководством. У хорошего архитектора должно быть «две головы», способные удерживать две разные идеи или концепции, две разные цели или точки зрения, чтобы он мог создать продукт, удовлетворяющий все заинтересованные стороны проекта.

Обратите внимание: у Януса две головы, а не просто два лица. Иначе говоря, он обладает дополнительной парой ушей и глаз, что повышает его осведомленность и восприимчивость. Хороший IT-архитектор должен быть превосходным слушателем и аналитиком. Понимание причин, по которым осуществляются инвестиции, жизненно важно для определения целей и представлений руководства о будущем организации. Умение оценивать

технические навыки вашего персонала в области проектирования и используемых в проекте технологий поможет сформировать подходящие пары для обучения и работы. Знание о том, какие решения с открытым кодом можно использовать в сочетании с теми или иными коробочными коммерческими программами, заметно сократит бюджет и сроки реализации проекта. Хороший архитектор должен разбираться в этих и многих других составляющих процесса разработки и уметь ставить их себе на службу, чтобы обеспечить успех проекта в целом.

Некоторые начальники ожидают от своих архитекторов едва ли не божественных способностей, но это не то, что я имел в виду, сравнивая архитектора с божеством. Хороший архитектор открыт для новых идей, инструментов и способов проектирования, способствующих прогрессу проекта, развитию команды или профессиональному совершенствованию; он не желает тратить большую часть своего времени на совещания с руководством или собственноручное написание кода; он должен распознавать ценные идеи и создавать атмосферу, способствующую их созреванию. Успеха в проектировании может добиться только подлинно открытый ум – ум, способный в ходе выполнения проекта находить точку равновесия множества противоборствующих сил. Каждый архитектор стремится завершить свой проект и обеспечить успех своей команды. Лучшие из архитекторов создают системы, которые выдерживают проверку временем, поскольку способны сохранять работоспособность и развиваться по мере роста организации и изменений в технологиях. Такие архитекторы прислушиваются к мнениям других, анализируют и перерабатывают свои процессы, методы и подходы к проектированию; они прилагают особые усилия, чтобы их продукты устояли перед неизбежными изменениями и переходами.

К такому образу мышления должны стремиться мы, архитекторы. Это легче сказать, чем сделать. Подобно Янусу, архитектор должен стать хранителем дверей и проходов, прокладывать мосты между старым и новым; он должен объединять творческий подход с надежным техническим фундаментом, чтобы, выполняя сегодняшние требования, быть готовым к завтрашним переменам.

Биография автора приведена на стр. 55.

В центре внимания архитектора – границы и интерфейсы

Эйнар Ландре

С тех пор как лорд Нельсон уничтожил французский и испанский флоты при Трафальгаре в 1805 году, принцип «разделяй и властвуй» стал главной мантрой для решения сложных, комплексных задач. Другой, более знакомый термин с таким же смыслом – *разделение ответственности (separation of concern*). Разделение ответственности ведет к инкапсуляции, а инкапсуляция способствует выделению границ и интерфейсов.

Если смотреть на задачу глазами архитектора, наиболее сложная ее часть – поиск естественных мест для формирования границ и определение подходящих интерфейсов, нужных для создания работоспособной системы. Это особенно трудно сделать в крупных корпоративных системах, где естественных границ зачастую очень мало, а различные предметные области тесно переплетены. В подобных ситуациях отчасти помогают традиционные принципы вроде «Минимизируй связанность, увеличивай сцепление» (minimize coupling, maximize cohesion) и «Не разрезай там, где нужен интенсивный обмен информацией», однако они ничего не говорят о том, как простым путем донести информацию о задачах и потенциальных решениях до заинтересованных сторон.

Здесь на помощь приходит концепция ограниченных контекстов и карт контекстов, описанная Эриком Эвансом в книге «Domain-Driven Design» (Проектирование на основе предметной области) (Addison-Wesley Professional). *Ограниченным контекстом* (*bounded context*) называется область уникального определения модели или понятия; на диаграммах она изображается в виде «облачка» с содержательным именем, определяющим его роль или задачу в предметной области. Например, система транспортировки может

содержать такие контексты, как «Отгрузка», «Планирование отгрузки» и «Перемещения в пределах порта». В других предметных областях возникнут другие имена.

После того как вы выделили ограниченные контексты и нанесли их на доску, наступает следующий этап – обозначение связей между контекстами. Эти связи могут отражать организационные, функциональные или технические зависимости. В результате создается *карта контекстов* (*context map*) – совокупность контекстов и интерфейсов между ними.

Карта контекстов становится в руках архитектора полезным инструментом, который помогает сосредоточиться на том, какие компоненты следует держать вместе, а какие необходимо отделить друг от друга; другими словами, наглядно реализуется принцип «разделяй и властвуй». Эта техника может быть с легкостью использована как для документирования и анализа текущей ситуации, так и для перепроектирования с целью создания системы, обладающей слабой связанностью, высоким сцеплением и четко определенными интерфейсами.

Биография автора приведена на стр. 27.

Поддерживайте разработчиков

Тимоти Хай

Сказать обычно проще, чем сделать; уж что-что, а говорить архитекторы умеют. Чтобы ваши слова не превращались в пустое сотрясание воздуха (основной метод возведения воздушных замков), вам понадобится хорошая команда разработчиков. Как правило, роль архитектора состоит в том, чтобы накладывать ограничения, но у вас есть также возможность эти ограничения снимать. Сделайте все от вас зависящее, чтобы развязать руки разработчикам.

Удостоверьтесь, что у разработчиков есть все нужные инструменты. Инструментарий не должен быть навязан сверху – он должен быть вдумчиво выбран, с тем чтобы у разработчиков под рукой всегда было все необходимое. Рутинную работу следует по возможности автоматизировать. Кроме того, не жалейте средств на то, чтобы обеспечить разработчиков современными компьютерами, достаточно широкими каналами связи и доступом к программам, данным и информации, необходимым для выполнения работы.

Проследите за тем, чтобы разработчики обладали необходимыми навыками. Если разработчикам необходимо обучение, позаботьтесь о том, чтобы они могли его пройти. Покупайте книги и поощряйте активные обсуждения технологий. Трудовая жизнь разработчика должна быть занята практической деятельностью, но это не должно мешать активному повышению квалификации. Если вы располагаете достаточными средствами, отправляйте свою команду на технические презентации и конференции. Если нет – подключите команду к техническим спискам рассылки и следите за бесплатными мероприятиями в своем городе. По возможности участвуйте также в процессе отбора разработчиков. Ищите тех, кто жаждет учиться, у кого есть «искра» технической одаренности (но убедитесь в том, что они способны работать в команде...). Трудно добиться чего-то выдающегося от группы безликих работяг.

Позволяйте разработчикам принимать самостоятельные решения, если эти решения не вступают в противоречие с общей целью проекта системы. Но там, где ограничения действительно важны, вводите их – не только в заботах о качестве, но и для дополнительной поддержки разработчиков. Создавайте стандарты не только ради обеспечения согласованности действий, но и для сокращения количества хлопотных мелких решений, не касающихся основной задачи, которую решают разработчики. Четко определите, где должны размещаться исходные файлы, какие имена им следует присвоить, когда создавать новые версии и так далее. Это сэкономит разработчикам время.

Наконец, оградите разработчиков от несущественных аспектов их работы. Излишняя бумажная волокита и прочая офисная возня порождают непроизводительные затраты и снижают эффективность. Вы (как правило) не являетесь руководителем, но можете влиять на процессы, сопутствующие разработке. Какие бы процессы ни использовались в вашем случае, убедитесь в том, что они устраняют препятствия, а не создают их.

*Тимоти Хай (**Timothy High**) – **архитектор программного обеспечения, у которого** за плечами более чем 15-летний опыт разработки с использованием веб-технологий, создания многоуровневых клиент–серверных систем и применения технологий интеграции приложений. В настоящее время работает архитектором программного обеспечения в компании Sakonnet Technologies, которая является лидером в области создания программ для управления сделками и рисками в сфере энергетики (**ETRM – Energy Trading and Risk Management**).*

Записывайте свои обоснования

Тимоти Хай

В СООБЩЕСТВЕ РАЗРАБОТЧИКОВ СУЩЕСТВУЕТ НЕМАЛО РАЗНОГЛАСИЙ по поводу ценности документации, особенно в том, что касается архитектуры программного продукта. Разногласия эти обычно связаны с субъективными взглядами на ценность «тщательного предварительного проектирования» и теми сложностями, которые возникают при постоянном обновлении проектной документации в соответствии с изменениями в базе кода.

Одна из разновидностей документации, которая почти не устаревает, не требует особых усилий по подготовке и почти всегда окупается, – запись обоснований, стоящих за теми или иными архитектурными решениями. Как объясняет Марк Ричардс в этюде «Архитектурные компромиссы», создание архитектуры программного продукта по сути сводится к поиску разумных компромиссов между различными характеристиками качества, затратами, временем и другими факторами. Вам, руководству, разработчикам и другим заинтересованным в проекте сторонам должно быть абсолютно ясно, почему одному решению отдано предпочтение перед другим и к каким компромиссам это привело. (Вы пожертвовали горизонтальной масштабируемостью ради сокращения затрат на оборудование и лицензии? Проблемы безопасности настолько серьезны, что оправдывают увеличение времени отклика ради шифрования данных?)

Формат такой документации зависит от специфики проекта и может варьироваться от кратких заметок в текстовом документе, вики или блоге до формальных шаблонов, отражающих все аспекты каждого архитектурного решения. Независимо от вида и формата этот документ должен отвечать на следующие основные вопросы: «В чем суть принятого решения?» и «Почему мы приняли такое решение?». Еще один частый вопрос, ответ на который следует там отразить: «Какие альтернативные решения рассматривались

и почему они были отвергнуты?» (на самом деле чаще спрашивают: «Почему не сделали так, как предлагал я?»). Документация должна предусматривать возможность поиска, чтобы при необходимости можно было легко найти требуемую информацию.

Такая документация может быть полезна во многих ситуациях:

- Чтобы проинформировать разработчиков о важных архитектурных принципах, которые должны соблюдаться в работе.
- Чтобы создать у членов команды единое ви́дение проекта (и даже предотвратить бунт), когда разработчики ставят под сомнение логику архитектуры (а возможно, покорно принять критику, если решение действительно оказалось несостоятельным при ближайшем рассмотрении).
- Чтобы продемонстрировать руководству и заинтересованным сторонам те причины, в силу которых программный продукт строится именно так, а не иначе (например, почему необходимо какое-нибудь дорогостоящее оборудование или программный компонент).
- Чтобы передать проект новому архитектору (сколько раз, получая в наследство программный продукт, вы пытались понять, почему архитекторы поступили именно ТАК?).

Однако самые важные преимущества этой практики состоят в следующем:

- Она заставляет вас явным образом формулировать свои доводы, проверяя их надежность (см. следующий этюд «Сомневайтесь в допущениях – особенно в собственных»).
- Она дает отправную точку для переоценки решений в случае изменения условий, от которых эти решения зависят.

По затрате усилий на подготовку такая документация сопоставима с заметками в ходе собраний или тематических обсуждений. Но какой бы формат вы ни выбрали, эти усилия окупятся.

Биография автора приведена на стр. 117.

Сомневайтесь в допущениях – особенно в собственных

Тимоти Хай

Закон отложенных решений Уэзерна гласит: «Допущения – корень всех провалов». Конечно, формулировка не очень серьезная, но когда предположения обходятся вам в несколько тысяч (а то и миллионов) долларов, становится не до смеха.

Опыт архитекторов программного обеспечения свидетельствует о том, что следует документировать обоснования каждого принятого решения, особенно если решение подразумевает компромисс (между производительностью и удобством сопровождения, между стоимостью и сроком выпуска продукта на рынок и т. п.). При более формальном подходе вместе с каждым решением регистрируется контекст, в котором оно принято, включая факторы, обусловившие окончательный выбор. Такими факторами могут быть не только функциональные или нефункциональные требования, но и факты (или «фактоиды»[1] ...), которые показались важными лицу, принимающему решение (ограничения технологий, квалификация работников, политическая среда и т. д.).

Практика документирования решений полезна тем, что перечисление этих факторов помогает выделить неявные допущения, которые влияют на важные решения относительно проектируемого продукта. Очень часто в основе

---

[1] Информация или публикация, недостойная доверия, либо событие сомнительной истинности, повсеместно принимаемое за правду. Термин введен американским писателем Н. Мейлером в 1973 году. – *Примеч. перев.*

этих допущений лежат «исторические причины», субъективные мнения, предубеждения разработчиков, опасения и сомнения[1] и даже слухи:

- «Продукты с открытым исходным кодом ненадежны».
- «От индексов на основе битовых карт больше хлопот, чем пользы».
- «Заказчик никогда не примет страницу, которая грузится по пять секунд».
- «IT-директор согласится покупать продукты только у крупных вендоров».

Очень важно, чтобы такие допущения формулировались явно и четко (ради тех, кто придет нам на смену, а также для будущей переоценки). Но, пожалуй, еще важнее проследить за тем, чтобы любые предположения, не подкрепленные актуальными эмпирическими доказательствами (или подтверждениями от участников, если речь идет о политических факторах), были проверены до окончательного принятия решения. А вдруг заказчик согласится подождать пять секунд при построении важнейшего отчета, если вы добавите индикатор выполнения? В каком именно отношении и какой именно проект с открытым кодом ненадежен? А вы тестировали индексы на основе битовых карт на своих данных, используя запросы и транзакции своего приложения?

Обратите внимание на слово «актуальный». То, что было справедливо для старой версии вашей программы, может стать недействительным сегодня. Производительность индексов на основе битовых карт может различаться в разных версиях **Oracle. Дыры в безопасности старой версии библиотеки мо**гут быть уже исправлены в новой версии. Старый надежный производитель или поставщик может быть на последнем издыхании в финансовом отношении. Технологический ландшафт изменяется каждый день, меняются и люди. Кто знает, может, ваш IT-директор стал тайным поклонником Linux?

Факты и допущения – это сваи, на которых строится ваш программный продукт. Позаботьтесь о том, чтобы эти сваи стояли на прочном фундаменте.

Биография автора приведена на стр. 117.

---

[1] В оригинале FUD (Fear, Uncertainty, Doubts – опасение, неуверенность, сомнения) – используемое в электронной переписке сокращение, возникшее согласно легенде в стенах компании IBM, менеджерам которой рекомендовалось в переписке с клиентами не критиковать продукцию конкурентов явным образом, а «высказывать FUD». – *Примеч. ред.*

Делитесь знаниями и опытом

Пол У. Хомер

Из всех своих начинаний, как успешных, так и неудачных, мы выносим много полезного. В такой молодой отрасли, как разработка программного обеспечения, распространение опыта и знаний жизненно необходимо для движения вперед. То, что любая из команд узнала в своем крошечном уголке мира, может повлиять на работу их коллег по всему земному шару.

Если смотреть на вещи реалистично, наша база фундаментальных (то есть абсолютных и теоретически истинных) знаний о разработке программ мала по сравнению с тем, что необходимо для успешного развития проекта. Чтобы компенсировать эту нехватку знаний, мы строим догадки, полагаемся на интуицию, а порой даже просто выбираем наудачу. Таким образом, в ходе любого крупного проекта по разработке программного обеспечения возникают эмпирические данные о том, что работает, а что нет. Мы шаг за шагом накапливаем опыт изменений, который хотим применять к отрасли в целом.

На индивидуальном уровне каждый из нас стремится повысить свою квалификацию и понять, как строить все более и более крупные системы. Задачи, которые ставит перед нами сама наша профессия, будут непрерывно усложняться, и при их решении мы хотим руководствоваться своим прежним опытом. Но, чтобы извлечь из обретенного опыта максимум пользы, его часто необходимо вывести на рациональный уровень. Самый лучший и простой способ сделать это – попытаться изложить его другому человеку.

Процесс обсуждения всегда помогает выявить слабости обсуждаемого предмета. Нельзя сказать, что вы что-то понимаете, если вы не можете это «что-то» легко объяснить. Только когда мы излагаем свои объяснения и обсуждаем их с другими, наш опыт преобразуется в знания.

Кроме того, даже если мы получили некий опыт, сделанные из него выводы могут оказаться не совсем правильными в более широком контексте. Возможно, мы были не настолько успешны или не настолько умны, как нам хотелось бы. Конечно, проверять свои знания в реальных условиях страшно, особенно когда нечто ценное для нас оборачивается мифом, ошибкой или заблуждением; оказаться неправым всегда неприятно.

Все мы люди, поэтому то, что происходит в наших головах, не всегда правильно и не каждая возникающая у нас мысль разумна. Только когда мы осознаем свою неидеальность, перед нами открывается путь к совершенствованию. Старая поговорка «На ошибках учатся» по-прежнему верна. Если наши идеи и убеждения не выдерживают проверки обсуждением, об этом лучше узнать до того, как мы положим их в основу своих решений.

Делясь обретенными знаниями и опытом, мы способствуем движению нашей отрасли вперед; мы сознаем, что это также помогает нам лучше понимать происходящее и исправлять ошибки. С учетом того, в каком состоянии пребывает значительная часть наших программ, очень важно использовать любую возможность рассказать другим о том, что мы поняли, узнали или полагаем, что узнали. Если мы поможем тем, кто нас окружает, стать лучше, они помогут нам в полной мере раскрыть наш потенциал.

Пол У. Хомер (Paul W. Homer) – программист, писатель и фотограф-любитель. Занялся разработкой программного обеспечения несколько десятилетий назад и с тех пор неустанно работает над построением все более сложных систем.

Патология шаблонов

Чед Лавинь

Шаблоны проектирования – один из самых полезных инструментов в арсенале архитектора программного обеспечения. Использование шаблонов позволяет создавать типовые решения, которые проще понять и объяснить другим. Сама идея шаблонов напрямую ассоциируется с качественным проектированием. Из-за этого у нас часто возникает соблазн продемонстрировать свою искушенность, включив в проект большое количество шаблонов. Если вы поймали себя на том, что упорно втискиваете свои любимые шаблоны в пространство задачи, где они неуместны, то не исключено, что вы стали жертвой *патологии шаблонов*.

Многие проекты страдают от такого положения дел. Глядя на них, так и видишь, как архитектор проекта поднимает взгляд от последней страницы сборника шаблонов, потирает руки и произносит: «Так, с какого шаблона начнем?..». Этот образ действий отчасти сродни подходу разработчика, начинающего писать класс с мысли: «Хм, от какого бы класса мне унаследовать?..». Шаблоны проектирования – отличный инструмент снижения неотъемлемой сложности, но, как и любой другой инструмент, его можно использовать неправильно. Есть известная пословица: «Человек с молотком везде видит гвозди»; когда в роли такого молотка оказываются шаблоны проектирования, проблемы неизбежны. Следите за тем, чтобы ваша любовь к шаблонам не превратилась в манию, которая приведет к реализации более сложных решений, чем это действительно необходимо.

Шаблоны – не панацея, а их использование не всегда служит признаком хорошего дизайна. Они представляют собой всего лишь повторно применимые решения типичных задач, найденные и описанные другими разработчиками, чтобы нам было легче узнать «уже изобретенный велосипед», если он попадется нам на глаза. Наша задача – выявить те проблемы, для которых

создавались эти решения, и правильно применить подходящий шаблон проектирования. Не позволяйте своему желанию продемонстрировать мастерское владение шаблонами проектирования взять верх над прагматизмом. Направьте свои усилия на проектирование систем, обеспечивающих эффективное решение бизнес-задач, и используйте шаблоны для решения тех проблем, для которых они предназначены.

Чед Лавинь (Chad LaVigne) – архитектор программных решений и внештатный технический специалист фирмы TEKSystems, Inc. (Балтимор). Работает в основном в Миннеаполисе, занимается проектированием и реализацией решений на базе технологий Enterprise Java.

Не увлекайтесь архитектурными метафорами

Дэвид Инг

Архитекторы обожают иметь дело с метафорами. Метафоры позволяют придать осязаемость абстрактным, сложным и ускользающим от понимания темам. При общении с другими членами команды или в процессе обсуждения архитектуры с конечным пользователем возникает неодолимый соблазн подобрать выразительную, хорошо знакомую по реальному миру метафору, которая описала бы то, что вы пытаетесь построить.

Обычно все начинается хорошо: собеседники находят общий язык, и кажется, что все идет в нужном направлении. Метафора развивается и растет до тех пор, пока не начинает жить собственной жизнью. Люди относятся к ней благосклонно – прогресс налицо!

А затем обычно наступает момент, когда архитектурная метафора вдруг становится опасной и восстает против своих создателей. Вот как это может происходить:

- Ваша метафора начинает нравиться заказчику больше, чем сама разрабатываемая система, то есть все заинтересованные стороны начинают верить в красивую иллюзию, созданную метафорой, не желая разбираться, что за ней на самом деле скрывается.

Пример: «Мы строим транспортную систему по принципу перемещения корабля между несколькими причалами».

Вы представляете себе контейнеровоз, бороздящий просторы Тихого океана, – я же имел в виду гребную лодку в бассейне, притом с одним веслом.

- Команда разработчиков начинает считать метафору более весомой по сравнению с реальной бизнес-задачей. А вы из любви к своей метафоре начинаете оправдывать странные решения.

Пример: «Мы сказали, что система будет похожа на картотечный шкаф, поэтому нам придется выводить данные в алфавитном порядке. Я знаю, что этот шкаф имеет шесть измерений, бесконечную длину и встроенные часы, но мы уже сделали иконку – а картотека должна вести себя как картотека...».

- В системе попадаются названия из старых, давно забытых метафор – археологические реликты концепций, давно отправленных в утиль.

Пример: «А почему объект Billing Factory[1] создает Port Channel для системы Rowing boat?.. И он в самом деле должен возвращать для Hub Bus представление Pomegranate?.. Ну да, я действительно работаю здесь недавно, а что?»

Итак, не влюбляйтесь в метафору, представляющую вашу систему, используйте ее в целях исследований и разъяснения, но не попадайте под ее влияние.

Дэвид Инг (David Ing) – архитектор программного обеспечения, живущий и работающий в Ванкувере. Родился в Великобритании, но переехал подальше от дождей (хотя сейчас считает, что был обманут лживой литературой для туристов).

Подчиняясь требованиям моды, сейчас Дэвид работает в компании Taglocity, специализирующейся на Web 2.0; здесь он пытается привести системы электронной почты «к цивилизованному виду», а заодно понять, что же такое Web 2.0.

[1] Billing Factory – фабрика счетов («Фабрика» – один из шаблонов проектирования); port channel – агрегирование каналов (технология, позволяющая объединить несколько физических каналов в один логический, в данном случае – отвечающий за это программный объект); rowing boat – гребная лодка; hub bus – концентратор шины; pomegranate – гранат. (Поскольку программные объекты обычно имеют английские названия, то в российской компании подобный разговор происходил бы именно так – на смеси русского и английского. Поэтому мы сочли правильным привести переводы и дать пояснения в сноске, а не напрямую в тексте.) – *Примеч. ред.*

Уделяйте пристальное внимание поддержке и сопровождению

Мнчедизи Каспер

Поддержка и сопровождение приложений никогда и ни при каких обстоятельствах не должны отходить на второй план. Поскольку более 80% жизненного цикла приложения занимает стадия сопровождения, вы должны уже на стадии проектирования обратить пристальное внимание на проблемы поддержки и сопровождения. Стоит вам забыть об этом, и вскоре вы с ужасом будете взирать на то, как ваше приложение превращается из мечты архитектора в нечто отвратительное, гибнет в конвульсиях и навечно остается в памяти людей как ваша неудача.

В ходе проектирования приложения перед внутренним взором архитекторов находятся главным образом разработчики, вооруженные интегрированными средами и отладчиками. Если что-то идет не так, искусные программисты переключаются в режим отладки и находят ошибку. Такая картина вполне естественна, поскольку большинство архитекторов основную часть своей жизни проводят в роли разработчиков, а не администраторов системы. К сожалению, разработчик и специалист службы поддержки обладают разными навыками – аналогично тому, как среда разработки/тестирования и рабочая среда (production environment) служат разным целям.

Вот примеры проблем, с которыми сталкивается администратор системы:

- Администратор не может заново отправить запрос, чтобы воспроизвести проблему. В рабочей среде вы не можете выполнить финансовую транзакцию в «боевой» базе данных, чтобы просто посмотреть, где произошла ошибка.
- Когда приложение находится в рабочей среде, требования исправить ошибки исходят от пользователей и начальства, а не от руководителя проекта и группы тестирования – а разгневанное начальство гораздо опаснее.

- В рабочей среде нет отладчика.
- В рабочей среде процесс развертывания планируется и координируется заблаговременно. Никто не разрешит вам отключить приложение на несколько минут, чтобы протестировать исправление.
- В среде разработки журналы сообщений обычно содержат гораздо более подробную информацию, чем в рабочей среде.

Некоторые из симптомов недостаточного планирования поддержки выглядят так:

- Большинство проблем требует привлечения разработчика.
- Отношения между командой разработчиков и службой поддержки весьма напряженные; разработчики считают, что в службе поддержки собрали одних идиотов.
- Служба поддержки ненавидит новое приложение.
- Команды архитекторов и разработчиков проводят много времени в рабочей среде.
- Приложение часто перезапускается для решения возникших проблем.
- Администраторам вечно не хватает времени для нормальной настройки системы, потому что они постоянно занимаются «тушением пожаров».

Чтобы ваше приложение успешно работало после того, как выйдет из рук разработчиков, вы должны:

- Понять, что разработка и поддержка требуют разных навыков.
- Как можно раньше вовлечь в проект руководителя службы технической поддержки.
- Сделать руководителя службы технической поддержки одной из центральных фигур проектной команды.
- Привлечь руководителя службы технической поддержки к планированию поддержки приложения.

Проектируйте продукт так, чтобы обучение персонала службы поддержки проходило с максимальной скоростью. Трассируемость, аудит и журналы играют в сопровождении чрезвычайно важную роль. Когда довольны администраторы системы, все остальные тоже довольны (особенно ваш начальник).

> *Мнчедизи Каспер (**Mncedisi Kasper**) – **директор по технологии и стратегии в Open Xcellence ICT Solutions, южноафриканской компании, специализирующейся на интеграции корпоративных приложений и консультациях в области SAP (ABAP/XI).***

Приготовьтесь выбрать два из трех

Билл де Ора

Иногда ввод некоторого ограничения или отказ от какого-либо свойства улучшает архитектуру, делает ее более простой и экономичной. Желательные свойства обычно группируются по три, но попытки описать и построить систему, обладающую всеми тремя свойствами, как правило, дают результат, посредственный во всех трех отношениях.

Знаменитый пример такого рода – теорема Брюэра, которая гласит, что для распределенной системы желательными являются три свойства – целостность (Consistency), доступность (Availability) и устойчивость к разделению (Partitioning tolerance) – и что достичь всех трех целей сразу невозможно. Попытки получить «все сразу» кардинально повышают затраты на разработку и, как правило, влекут за собой рост сложности, не приводя при этом к желаемому эффекту и реальному достижению бизнес-цели. Если данные должны быть распределенными и постоянно доступными, обеспечение целостности обходится все дороже и в конечном счете становится нереальным. Аналогичным образом, если система должна быть распределенной и целостной, обеспечение целостности ведет сначала к задержкам и проблемам с производительностью и в конце концов – к недоступности в то время, когда система занята проверкой данных.

Нередко встречаются ситуации, когда одно или несколько свойств считаются неприкосновенными: данные не должны дублироваться, все операции записи должны быть транзакционными, система должна поддерживать 100-процентную доступность, вызовы должны быть асинхронными, в системе не должно быть единых точек отказа, все составные части должны быть расширяемыми и так далее. Такое фанатичное отношение к свойствам не только наивно, но и мешает нам думать о тех реальных задачах, которые стоят перед нами. От обсуждения главных принципов проектирования мы

уходим в обсуждение отклонений архитектуры от идеала и начинаем путать догматизм с качественным управлением. Вместо этого следует задать вопросы: Почему эти свойства так необходимы? Какие преимущества они приносят? Когда они желательны? Как разбить систему на части для достижения лучшего результата? В таких случаях всегда проявляйте здоровый скепсис, потому что архитектурные догмы способны сделать поставку продукта проблематичной. Принять неизбежность таких компромиссов – одна из самых трудных задач в разработке программного обеспечения, причем не только для архитекторов, но также для разработчиков и других заинтересованных лиц. И все же это гораздо лучше, чем иметь безграничную свободу выбора, – неизбежные компромиссы часто ведут к творческим, нестандартным результатам.

Билл де Ора (Bill de hÓra) – ***ведущий архитектор в компании NewBay Software, где он работает над крупномасштабными веб-системами и системами для мобильных устройств. Является соредактором Atom Publishing Protocol, ранее участвовал в работе группы W3C RDF Working Group. Признанный эксперт в области REST и архитектур на основе передачи сообщений, а также проектирования протоколов.***

Принципы, аксиомы и аналогии важнее личных мнений и предпочтений

Майкл Хармер

При создании архитектуры системы следует руководствоваться явно сформулированными принципами, аксиомами и аналогиями. Это дает ряд преимуществ, недоступных в том случае, если вы всецело полагаетесь на личный опыт, мнения и вкусы.

Прежде всего, упростится документирование архитектуры. Вы можете начать с описания принципов, использованных при проектировании. Это гораздо проще, чем пытаться изложить свое личное мнение и опыт. Описанные принципы станут удобной отправной точкой для тех, кому предстоит разобраться в вашей архитектуре и реализовать ее. Кроме того, такое описание окажет бесценную помощь начинающим архитекторам, которые будут работать с вашей архитектурой.

Четкое изложение принципов избавляет архитектора от необходимости лично участвовать во всех процессах, для которых архитектура системы является определяющим элементом. Это изложение расширяет возможности и влияние архитектора. Вам необязательно становиться вездесущим трудоголиком, чтобы другие без особых трудностей смогли:

- Реализовать и адаптировать архитектуру
- Расширить архитектуру на смежные предметные области
- Модифицировать архитектуру для реализации с использованием новых технологий
- Подробно проработать граничные случаи

Расхождения во мнениях и вкусах неизбежно переходят в политические споры, в которых побеждает тот, кто обладает властью. Напротив, несогласие с ясными основополагающими принципами ведет к конструктивной

дискуссии без переходов на личности. Более того, появляется возможность разрешить спорные моменты вообще без обращения к архитектору.

Принципы и аксиомы обеспечивают также согласованность архитектуры как в ходе реализации, так и в дальнейшем. Поддержание согласованности часто становится серьезной проблемой, особенно в крупных системах, охватывающих несколько технологий и существующих в течение многих лет. Четко сформулированные архитектурные принципы помогут человеку, незнакомому с конкретной технологией или компонентом, быстрее разобраться в новой области и понять ее.

Майкл Хармер (Michael Harmer) занимается разработкой программного обеспечения уже 16 лет. Он был рядовым разработчиком, руководителем группы, архитектором, ведущим специалистом и руководителем направления.

Начните с ходячего скелета

Клинт Шенк

Чрезвычайно полезная стратегия реализации, проверки и совершенствования архитектуры приложения – начать с того, что Алистер Коберн (Alistair Cockburn) называет *ходячим скелетом*. Речь идет о минимальной реализации системы «от начала до конца», связывающей воедино все основные архитектурные компоненты. Начав с минимума – **с рабочей системы, содержащей** все коммуникационные каналы, – вы можете быть уверены в том, что движетесь в правильном направлении.

Когда скелет будет готов, можно переходить к наращиванию плоти, то есть постепенному, пошаговому добавлению конечной функциональности. Ваша цель – сохранять работоспособность системы, пока скелет обрастает мясом.

Чем дольше существует система и чем больше ее размеры, тем труднее дается и дороже обходится внесение изменений. Ошибки желательно обнаруживать как можно раньше. Такой подход обеспечивает нас коротким циклом обратной связи, что позволяет быстрее адаптировать архитектуру путем итеративной работы над атрибутами времени выполнения системы в соответствии с приоритетами бизнеса. Все допущения, касающиеся архитектуры, также проверяются раньше. При этом развитие созданной архитектуры происходит более простым путем, потому что проблемы выявляются на ранней стадии, когда на реализацию потрачено меньше сил и средств.

Чем крупнее система, тем важнее использовать эту стратегию. В небольшом приложении всю функциональность от начала до конца может относительно быстро реализовать один разработчик, но в более крупных системах такой подход становится непрактичным. Достаточно часто встречается ситуация, когда в реализации заняты несколько разработчиков из одной команды (и даже нескольких распределенных команд). Соответственно необходима

более тесная координация. К тому же разработчики выдают результаты в разном темпе: одни успевают сделать много за небольшой промежуток времени, другие тратят уйму времени на небольшую задачу. Самые сложные и трудоемкие задачи должны выполняться на ранней стадии проекта.

Начните со скелета, заставьте его ходить, а затем постепенно наращивайте на него плоть.

Клинт Шенк (Clint Shank) – разработчик, консультант и преподаватель из ***Sphere of Influence, Inc., компании, предоставляющей услуги по проектированию*** *и производству программных продуктов коммерческим и правительственным организациям.*

В основе всего – данные

Пол У. Хомер

Мы, разработчики, изначально воспринимаем программное обеспечение как систему команд, функций и алгоритмов. Такое «командно-ориентированное» представление помогает нам освоить построение ПО, но оно же начинает мешать, когда мы пытаемся создавать более масштабные системы.

В конце концов, компьютер – не что иное, как хитроумный инструмент, который помогает нам получать и обрабатывать горы данных. Структура этих данных лежит в основе нашего понимания того, как справиться со сложностью крупномасштабной системы. Миллионы команд сложны по своей природе, однако за ними стоит небольшой набор базовых структур данных, который мы можем легко понять.

Например, если вы хотите разобраться в операционной системе UNIX, вряд ли вам сильно поможет построчный просмотр ее исходного кода. Но если вы прочитаете книгу, которая описывает основные внутренние структуры данных, обеспечивающих работу процессов, файловых систем и т. д., вы скорее поймете глубинные принципы работы UNIX. Концептуально структуры данных имеют гораздо меньший размер и существенно более просты по сравнению с кодом.

В ходе выполнения кода на компьютере состояние данных непрерывно изменяется. С абстрактной точки зрения любой алгоритм можно рассматривать как простое преобразование данных из одной версии в другую. Вся функциональность системы воспринимается как большой набор четко определенных преобразований, переводящих данные между разными состояниями.

Такое ориентированное на данные представление, когда система рассматривается исключительно через призму структуры данных, лежащих в ее основе, способно даже самую сложную систему свести к реально воспринимаемому

набору деталей. А снижение сложности позволяет понять, как построить сложную систему и работать с ней.

Данные находятся в центре большинства задач. Задачи, относящиеся к предметной области, проникают в код через данные. Большинство ключевых алгоритмов хорошо изучены и проанализированы, а вот структура данных и связи между ними изменяются часто. Проблемы уже работающих систем (такие как обновление) также существенно усложняются, если затрагивают данные. Изменение кода или поведения – не такая уж серьезная проблема: нужно просто выпустить новую версию системы; но изменение структур данных может потребовать огромных усилий по преобразованию старой версии данных в новую.

И конечно, многие фундаментальные проблемы архитектуры программного обеспечения в действительности связаны с данными. Собирает ли система правильные данные в правильное время? Кто должен иметь возможность просматривать или изменять эти данные? Если данные уже существуют, насколько они качественны и как быстро растет их объем? Если данных пока нет, то какова их структура, откуда они поступят и насколько надежен их источник? А когда данные окажутся в системе, останется еще один вопрос: есть ли способ просмотра и/или редактирования конкретных данных или его необходимо добавить в систему?

С точки зрения дизайна критическим вопросом для большинства систем является получение правильных данных в нужное время. Все те преобразования, которые применяются к данным с этого момента, нужны, чтобы обеспечить их доступность, реализацию функциональности и сохранение результатов. Большинству систем для нормальной работы не требуется высокая внутренняя сложность – они просто должны накапливать все большие и большие объемы данных. Пользователь видит прежде всего функциональность, но ядро каждой системы образуют данные.

Биография автора приведена на стр. 123.

Простое должно быть простым

Чед Лавинь

Архитекторы программного обеспечения решают множество очень сложных задач, но наряду с ними встречаются и относительно простые. А вот чего мы стремимся избежать, так это решения простых задач сложными методами. Каким бы очевидным ни казался этот совет, следовать ему порой нелегко. Проектировщики программного обеспечения – умные, очень умные люди. Однако весьма легко попасть в ловушку «простая задача – сложное решение», потому что все мы любим демонстрировать свои знания. Если вы почувствовали, что проектируете решение настолько умное, что оно со временем, того и гляди, проявит искру самосознания, остановитесь и подумайте. Соответствует ли такое решение поставленной задаче? При отрицательном ответе рассмотрите заново варианты дизайна системы. Простое должно быть простым. У вас будет масса возможностей продемонстрировать свой талант, когда вы столкнетесь со сложными задачами, а это непременно случится.

Конечно, это вовсе не означает, что наши решения не должны быть элегантными. Речь о том, что если вам поручают спроектировать систему для поддержки складского хранения и продаж единственного типа продукции, то, вероятно, не стоит закладывать в систему возможность динамически настраивать иерархию продуктов.

Затраты, обусловленные излишней сложностью решения, могут показаться небольшими, но они, скорее всего, превысят ваши изначальные оценки. На архитектурном уровне чрезмерная сложность решений создает в целом те же проблемы, что и на уровне разработки, однако отрицательные эффекты имеют склонность умножаться. Неудачные решения, принятые на уровне проектирования, сложны в реализации, сопровождении и, что хуже всего, в отмене. Прежде чем выбрать архитектурное решение, выходящее за рамки

требований, спросите себя, насколько сложно будет отказаться от этого решения после его реализации.

Впрочем, затраты не ограничиваются реализацией и сопровождением упомянутого решения. Если вы потратите на простую задачу больше времени, чем необходимо, у вас останется меньше времени для решения действительно сложных проблем, когда они возникнут. И вдруг обнаруживается, что из-за ваших архитектурных решений границы проекта начали расползаться, создавая неоправданный риск для проекта. Ваше время можно было бы потратить куда более эффективно.

Часто возникает сильное желание оправдать решения предполагаемой выгодой или прогнозируемыми требованиями. Запомните: когда вы пытаетесь угадать будущие требования, в 50% случаев вы ошибаетесь, а в 49% – очень-очень сильно ошибаетесь. Решайте проблемы по мере их поступления. Сдайте приложение вовремя и дождитесь обратной связи, чтобы сформулировать реальные требования. Созданная вами простая архитектура намного упростит реализацию новых требований в случае их поступления. А если вы все же угадаете и спрогнозированные вами требования станут реальными к следующей версии, то у вас уже будет в голове готовое решение. Отличие в том, что теперь вы сможете выделить для него должное время, потому что оно действительно необходимо. И вы с вашей командой опомниться не успеете, как завоюете репутацию профессионалов, способных хорошо оценить задачу и справиться со своей работой в срок.

Биография автора приведена на стр. 125.

Архитектор – прежде всего разработчик

Майк Браун

Вы когда-нибудь слышали о судье, который не был бы юристом, или о заведующем хирургическим отделением, который не был бы хирургом? Даже достигнув того, что может считаться венцом их карьеры, люди, занимающие такие должности, продолжают осваивать новые разработки в своей области. Мы, архитекторы программного обеспечения, обязаны соответствовать тем же стандартам.

Как бы качественно ни было спроектировано решение, успех его реализации в существенной степени определяется тем, сможете ли вы привлечь на свою сторону разработчиков. А самый быстрый способ сделать это – завоевать их уважение и доверие. Всем известно, как проще всего завоевать доверие разработчика: *ваш код – ваша «визитная карточка»*. Если разработчики будут знать, что вы не какой-то абстрактный мечтатель, не способный написать ни строчки кода, они будут меньше ворчать по поводу ухищрений, на которые вы «заставляете» их идти для отображения данных на странице, ведь вы способны с полным на то основанием сказать: «Можно сделать проще, всего лишь привязав датасет к гриду», – и им будет нечего возразить.

И хотя это не является моей непосредственной работой, я часто беру на себя некоторые наиболее сложные задачи. Во-первых, это интересно и помогает мне поддерживать на уровне свои навыки программирования; во-вторых, этим я наглядно показываю своим разработчикам, что мои слова не пустое сотрясание воздуха.

Архитектор должен стремиться к созданию решения осуществимого, удобного в сопровождении и, конечно, отвечающего сути задачи. Одним из критериев осуществимости решения является возможность адекватно оценить объем работы и усилия, необходимые для реализации элементов решения. Поэтому я считаю, что *если вы проектируете систему, вы должны быть способны ее реализовать*.

*Майк Браун (**Mike Brown**) – **ведущий разработчик в компании Software Engineering Professionals, Inc. (http://www.sep.com). Обладает 13-летним опытом** работы в сфере информационных технологий, включая 8 **лет разработки** корпоративных решений для разнообразных вертикальных рынков. Является учредителем группы пользователей **Indianapolis Alt.NET, соучредителем** WPF Disciples, а также организатором группы профессиональных пользователей Indy Arc.*

Окупаемость как фактор проектирования

Джордж Маламидис

Все решения, которые мы принимаем в наших проектах – технические, организационные, кадровые, – можно рассматривать как своего рода инвестиции. С любыми инвестициями связаны определенные затраты (выраженные в денежной или какой-либо иной форме), которые, как предполагается, со временем окупятся. Наши работодатели платят нам деньги в надежде, что эти инвестиции положительно отразятся на результатах деятельности их предприятия. Мы выбираем определенную методологию разработки в надежде, что она повысит результативность работы нашей команды. Мы решаем потратить целый месяц на переработку физической архитектуры приложения, ожидая, что это принесет выгоду в долгосрочной перспективе.

Одним из показателей успешности инвестиций является *окупаемость* (ROI, Return on Investment). Например: мы предполагаем, что дополнительные затраты времени на создание тестов уменьшат количество ошибок в следующем выпуске приложения. Размер инвестиций в этом случае определяется временем, потраченным на написание тестов, а прибыль – это время, сэкономленное на исправлении ошибок в будущем, плюс удовлетворение клиентов, работающих с более качественной программой. Допустим, в настоящее время 10 из 40 рабочих часов в неделю тратятся на исправление ошибок. По нашим оценкам, выделив 4 часа в неделю на тестирование, мы сократим затраты времени на исправление ошибок до 2 часов в неделю, фактически получив 8 свободных часов, которые можно вложить во что-то другое. Прогнозируемая окупаемость составляет 200%[1] (8 часов, сэкономленных на исправлении ошибок, делим на 4 часа, потраченных на тестирование).

---

[1] В действительности эффект будет не настолько сильным: переключение между задачами само по себе отнимает у разработчика существенное время, тем самым снижая его продуктивность. – *Примеч. науч. ред.*

Не все следует сводить к выгоде в денежном эквиваленте, однако наши инвестиции должны обеспечивать добавленную стоимость. Если в текущем проекте срок выпуска на рынок критичен для заинтересованных сторон, возможно, «пуленепробиваемая» архитектура, требующая долгого предварительного проектирования, не обеспечит такой привлекательной окупаемости, как быстрый выпуск альфа-версии. Быстрый выпуск позволит изучить реакцию аудитории, что может стать определяющим фактором выбора будущего направления и успеха проекта; в то же время планирование на скорую руку может обернуться лишними затратами из-за трудностей с масштабированием приложения, если такая необходимость возникнет. Окупаемость каждого варианта можно определить путем анализа затрат и предполагаемых прибылей, а затем использовать ее как критерий выбора при наличии нескольких альтернатив.

Рассматривайте архитектурные решения как инвестиции и анализируйте связанную с ними окупаемость; это полезный метод оценки реалистичности и практичности имеющихся вариантов.

Джордж Маламидис (George Malamidis) – разработчик программного обеспечения в компании TrafficBroker (Лондон). До этого был ведущим консультантом и ведущим техническим специалистом в ThoughtWorks. Участвовал в разработке и внедрении критических приложений в разных областях, от сетевых и финансовых приложений до Web 2.0.

Ваша система станет унаследованной – учитывайте это при проектировании

Дейв Андерсон

Даже если вы создаете сверхсовременную систему на базе новейших технологий, для вашего преемника она станет унаследованной (legacy). С этим ничего не поделаешь! Современному программному обеспечению свойственно быстро устаревать. Если вы ожидаете, что ваша система будет запущена в реальную эксплуатацию и просуществует хотя бы несколько месяцев, смиритесь с тем, что сопровождающим ее разработчикам придется вносить в нее исправления. Отсюда вытекает ряд требований к системе:

- Ясность: должно быть сразу понятно, какую роль играет тот или иной компонент или класс.
- Удобство тестирования: насколько легко проверить вашу систему?
- Корректность: работает ли система так, как было задумано? Исключите исправления, вносимые на скорую руку.
- Трассируемость: сможет ли специалист-«пожарник», никогда прежде не видевший ваш код, диагностировать ошибку в работающей системе и исправить ее? Или же на то, чтобы войти в суть дела, ему потребуются два месяца?

Попробуйте представить себе, что будет, если какая-то другая команда откроет вашу базу кода и попытается в нем разобраться. Это наиважнейший аспект хорошей архитектуры. Систему необязательно чрезмерно упрощать или документировать до мельчайших подробностей; хороший дизайн во многих отношениях документирует сам себя. Кроме того, дизайн может проявлять себя в поведении системы во время эксплуатации. Например, система с «разросшейся» архитектурой, опутанной уродливыми зависимостями, в эксплуатации часто ведет себя, как зверь в клетке. Подумайте о других

(обычно менее опытных) разработчиках, которым придется исправлять ошибки в вашей программе и отлаживать код.

В кругах разработчиков не любят термин «унаследованный», но на самом деле этот ярлык несет на себе любая программная система. И в этом нет ничего плохого, потому что такое определение может означать лишь то, что ваша система долговечна, соответствует ожиданиям пользователей и обладает коммерческой ценностью. Если программную систему никогда не называли унаследованной, ее, скорее всего, отправили пылиться на полке, так и не запустив, а это вряд ли можно считать свидетельством удачной архитектуры.

Дейв Андерсон (Dave Anderson) – главный специалист по разработке ПО в компании Liberty IT (Белфаст), которая поставляет IT-решения для компании Liberty Mutual, входящей в список Fortune 100. Дейв обладает более чем 10-летним опытом разработки ПО для многих ведущих IT-компаний в разных отраслях и странах.

Когда видите единственное решение, спросите других

Тимоти Хай

Вероятно, вам уже доводилось слышать это высказывание. Каждый опытный архитектор знает: если он видит только одно решение задачи, это плохой признак.

Создание архитектуры программного обеспечения сводится к поиску наилучшего решения задачи при некоторых заданных ограничениях. Редко когда удается обеспечить выполнение всех требований и уложиться во все ограничения с первым же решением, которое пришло вам в голову. Обычно приходится идти на компромиссы, выбирая решение, лучше всего удовлетворяющее тем требованиям, которые определяются важнейшими приоритетами.

Если вы видите только одно решение текущей проблемы, это означает, что у вас отсутствует свобода выбора для поиска таких компромиссов. Вполне возможно, что это единственное решение окажется неудовлетворительным для некоторых заинтересованных в проекте сторон. Это означает также, что при смене приоритетов, обусловленной изменениями бизнес-окружения, у вашей системы не будет возможности адаптироваться к новым требованиям.

Такая ситуация крайне редко возникает из-за реального отсутствия альтернатив. Гораздо более вероятное объяснение – неопытность архитектора в предметной области конкретной задачи. Если вы знаете, что это относится к вам, не постесняйтесь обратиться за советом к кому-нибудь из более опытных в этом вопросе коллег.

Другое, более коварное проявление этой проблемы встречается при проектировании архитектуры «по привычке». Архитектор может иметь богатый опыт применения одного архитектурного стиля (например, трехуровневая система типа клиент–сервер), но ему не хватает знаний, чтобы определить, когда этот стиль неуместен. Если вы оказались в ситуации, когда решение

известно вам сходу, без сопоставления с другими возможными вариантами, задержитесь, сделайте шаг назад и попытайтесь придумать другой способ решения той же задачи. Если это не получается, возможно, стоит обратиться за помощью.

Один мой друг работал техническим руководителем в небольшой, но быстро развивающейся интернет-компании. По мере роста числа пользователей начала возрастать и нагрузка на систему. Это привело к резкому падению производительности, и компания начала терять с таким трудом обретенных пользователей.

Начальник спросил его:

– Что мы можем сделать для улучшения производительности?

У моего друга уже был готов ответ:

– Купить более мощную машину!

– А что еще?

– М-м-м... Насколько я знаю, ничего.

Моего друга немедленно уволили. Разумеется, начальник был прав.

Биография автора приведена на стр. 117.

Осознавайте последствия изменений

Дуг Кроуфорд

Хороший архитектор снижает сложность до минимума и может спроектировать решение, абстракции которого, с одной стороны, обеспечивают надежный фундамент, а с другой – достаточно прагматичны, чтобы пережить изменения.

Выдающийся архитектор понимает последствия изменений – не только в отдельных программных модулях, но также в отношениях между людьми и между системами.

Изменения могут проявляться в разных формах:

- Изменения функциональных требований
- Ужесточение требований к масштабируемости
- Модификация интерфейсов системы
- Приход и уход членов команды
- и так далее...

Масштаб и сложность изменений в программном проекте невозможно определить заранее. Бесполезно пытаться объехать каждую выбоину на дороге до того, как вы на нее наткнетесь. Но переживет проект очередное препятствие или нет, в большой степени зависит от архитектора.

Задача архитектора – не столько управлять изменениями, сколько позаботиться об их управляемости.

Для примера возьмем распределенное решение, в котором задействовано несколько приложений, объединенных различным промежуточным программным обеспечением (middleware). Если набор зависимостей неверно определен или неточно представлен в визуальной модели, изменения

бизнес-процесса могут привести к настоящему хаосу. Последствия изменений оказываются особенно серьезными, если изменения затрагивают модель данных или нарушают имеющиеся интерфейсы, а существующие долговыполняемые транзакции с отслеживанием состояния должны быть успешно завершены в старой версии процесса.

Пример может показаться излишне радикальным, но решения с высокой степенью интеграции в наше время применяются повсеместно. Это очевидно по ассортименту предлагаемых стандартов интеграции, инфраструктур и шаблонов. Чтобы обеспечить вашим клиентам нормальный уровень поддержки, крайне важно понимать последствия изменений в таких системах, частично лежащих за пределами вашей досягаемости.

К счастью, существует целый ряд инструментов и методов, позволяющих смягчить воздействие изменений:

- Вносите небольшие, поэтапные изменения
- Создавайте воспроизводимые тестовые сценарии и часто запускайте их
- Упрощайте построение тестовых сценариев
- Отслеживайте зависимости
- Действуйте и реагируйте систематично
- Автоматизируйте повторяющиеся задачи

Архитектор должен оценить воздействие изменений на различные аспекты масштаба проекта, времени и бюджета и быть готовым уделить больше внимания тем областям, которые способны оказать наиболее сильное влияние, став той самой «выбоиной на дороге». Оценка рисков – полезный инструмент для принятия решения о том, на что следует расходовать ваше драгоценное время.

Снижение сложности – важная задача, но низкая сложность не равнозначна простоте. Понимание возможных изменений и их влияния на решение оказывает неоценимую пользу в среднесрочной и долгосрочной перспективах.

Дуг Кроуфорд (Doug Crawford) руководит командой разработчиков промежуточного программного обеспечения для телекоммуникационной компании в Южной Африке. Последние 10 лет он с успехом совмещал несовместимое и принимал самое непосредственное участие в проектах по интеграции приложений в разнообразных отраслях: рекламе, банковском обслуживании крупного бизнеса, страховании и образовании.

Архитектор должен разбираться в оборудовании

Камал Викраманаяке

Для многих архитекторов тема планирования мощностей оборудования выходит за рамки их «зоны комфорта», однако она остается важной частью работы архитектора. Причины, по которым архитекторы часто забывают уделить должное внимание оборудованию, весьма разнообразны, но все они связаны большей частью с недопониманием и нечеткостью требований.

Главная причина заключается в том, что мы полностью концентрируемся на программной стороне и игнорируем аппаратные требования. Вдобавок языки высокого уровня и программные инфраструктуры (software frameworks) естественным образом изолируют нас от оборудования.

Существенным фактором являются и нечеткие требования, поскольку они могут быть неправильно поняты или подвержены значительным изменениям. По мере развития архитектуры ее аппаратные аспекты тоже изменяются. К тому же может оказаться, что клиент не осознает или не может оценить размер контингента пользователей либо спрогнозировать динамику использования системы. Наконец, оборудование постоянно совершенствуется. То, что мы знаем о нем по прошлому опыту, может быть неактуальным сегодня.

Без знания оборудования прогнозирование аппаратной конфигурации разрабатываемых систем – занятие чрезвычайно ненадежное. В качестве компенсации некоторые архитекторы используют большие значения запаса производительности, которые обычно не опираются на какие-либо объективные оценки или методологические рекомендации. В большинстве случаев это приводит к избыточной производительности оборудования, которая не будет задействована даже в периоды пиковой нагрузки. В результате деньги клиента расходуются на оборудование более мощное, чем реально необходимо системе.

Лучшая защита от некачественного аппаратного планирования – тесное взаимодействие с архитектором аппаратной инфраструктуры. В отличие от архитекторов программного обеспечения, архитекторы аппаратной инфраструктуры хорошо разбираются в планировании вычислительных мощностей; они должны быть частью вашей команды. Впрочем, не каждому архитектору программного обеспечения доступна такая роскошь, как возможность работать с архитектором аппаратной инфраструктуры. В таких случаях архитектор системы может принять некоторые меры, снижающие вероятность ошибок при планировании оборудования.

Например, вам может пригодиться прошлый опыт. Если вы прежде уже занимались реализацией систем, то у вас имеется некоторое представление о планировании аппаратной мощности – хотя бы на основании ретроспективного анализа. Вы можете также обсудить эту тему со своим клиентом и убедить его выделить средства на планирование мощности оборудования. Финансирование такой деятельности часто оказывается намного выгоднее покупки оборудования сверх реально необходимого. В этом случае ключевая роль отводится горизонтальной масштабируемости[1]: оборудование добавляется по мере надобности вместо избыточных закупок в самом начале. Чтобы «горизонтальная стратегия» работала, архитектор ПО должен постоянно проводить измерения вычислительной мощности и изолировать программные компоненты для запуска в среде с прогнозируемыми показателями производительности.

Планирование вычислительной мощности не менее важно, чем архитектура; уделяйте первостепенное внимание этому вопросу независимо от того, имеется ли у вас под рукой специалист по аппаратной инфраструктуре или нет. Подобно тому как архитектор программного обеспечения отвечает за установление связей между потребностями и программным решением, архитектор аппаратной инфраструктуры отвечает за установление связей между аппаратной и программной частями.

Камал Викраманаяке (Kamal Wickramanayake) – архитектор аппаратного и программного обеспечения, живет на Шри-Ланке. Является обладателем сертификата TOGAF от The Open Group.

[1] Горизонтальная масштабируемость – разбиение системы на более мелкие структурные компоненты и разнесение их по отдельным физическим машинам либо увеличение количества серверов, параллельно выполняющих одну и ту же функцию. Вертикальная масштабируемость – увеличение производительности каждого компонента системы с целью повышения общей производительности. (См. *http://ru.wikipedia.org/wiki/Масштабируемость*.) – *Примеч. ред.*

«Срезание углов» сейчас обойдется слишком дорого потом

Скот Макфи

При создании архитектуры важно помнить, что на сопровождение системы в долгосрочной перспективе расходуется больше ресурсов, чем собственно на разработку. «Срезание углов» на фазе разработки проекта может вылиться в существенные затраты на этапе сопровождения.

Допустим, кто-то сказал вам, что модульные тесты не приносят непосредственной пользы, и вы даете своим разработчикам указание не углубляться в их создание. Впоследствии это значительно затруднит модификацию готовой системы и породит чувство неуверенности во внесенных изменениях. Относительно небольшие изменения потребуют гораздо большего объема ручного тестирования, что приведет к нестабильности и росту затрат на сопровождение, а получившийся дизайн нельзя будет назвать полностью тестируемым (не говоря уже о соответствии принципу «опережающего тестирования»[1]).

Серьезной архитектурной ошибкой является и попытка приспособить существующую систему к тем целям, для которых она не предназначалась, под тем предлогом, что использование существующей системы может каким-то образом снизить затраты. Например, архитектурные компоненты BPEL[2] в сочетании с триггерами баз данных можно приспособить для реализации

---

[1] Здесь подразумевается одна из гибких методик разработки, получившая название TDD (Test-Drived Design, проектирование, направляемое тестами). В этой методике отправной точкой процесса проектирования является не моделирование диаграмм, а написание тестов. – *Примеч. науч. ред.*

[2] BPEL (Business Process Execution Language) – основанный на XML язык формального описания бизнес-процессов и протоколов их взаимодействия (см. *http://ru.wikipedia.org/wiki/BPEL*). – *Примеч. ред.*

системы на основе асинхронной передачи сообщений. Такие решения обычно возникают из соображений удобства либо потому, что эта архитектура известна вам или клиенту. Однако действительным основанием для такого выбора могут послужить только четко сформулированные требования – это обязательное условие. Неудачные решения на ранней стадии проекта обходятся очень дорого, когда архитектуру системы приходится изменять в соответствии с новыми требованиями.

В начальной фазе разработки важно не только избегать «срезания углов», но и сразу исправлять неудачные проектировочные решения по мере их обнаружения. Отдельные плохо спроектированные аспекты системы могут лечь в основу других аспектов, что сделает последующие исправления еще более затратными.

Например, если вы поняли, что выбранные библиотеки плохо подходят для какой-то части функциональности системы, замените их как можно скорее. В противном случае усилия, направленные на то, чтобы приспособить их к изменяющимся требованиям, приведут к появлению дополнительных уровней абстракции, маскирующих несоответствия предыдущего уровня. Архитектура запутывается в сплошной клубок противоречий, и с каждым новым уровнем распутывать его становится все труднее. В итоге получается система, сопротивляющаяся любым изменениям.

Столкнувшись с архитектурной проблемой или дефектом проектирования, вы как архитектор должны настоять на том, чтобы их исправили немедленно, пока это обойдется еще не так дорого. Чем дольше вы будете откладывать, тем дороже придется платить.

Скот Макфи (Scot Mcphee) – австралийский разработчик и архитектор с 15-летним опытом программирования и проектирования приложений. Последние 8 лет работал главным образом с технологиями семейства J2EE.

Лучшее – враг хорошего

Грег Найберг

Проектировщики программного обеспечения (и особенно архитекторы) склонны оценивать решения по тому, насколько элегантно и оптимально они решают конкретную задачу. Словно судьи на конкурсе красоты, мы смотрим на дизайн или реализацию и немедленно видим незначительные дефекты или недостатки, которые можно легко устранить всего несколькими дополнительными изменениями или итерациями рефакторинга. Модель предметной области буквально умоляет выполнить еще один проход для поиска общих атрибутов или функций, которые можно было бы переместить в базовые классы. Службы, продублированные в нескольких реализациях, стенают о преобразовании их в веб-службы. Запросы требуют внимания, жалуясь на буферизацию и неуникальные индексы.

Мой совет: не поддавайтесь искушению довести дизайн или реализацию системы до совершенства! Ориентируйтесь на отметку «достаточно хорошо» и остановитесь, когда вы доберетесь до нее.

«Что именно означает “достаточно хорошо”?» – спросите вы. Это означает, что оставшиеся недочеты не оказывают сколько-нибудь заметного влияния на функциональность, удобство сопровождения или производительность системы. Архитектура и дизайн идут рука об руку. Реализованная система работает и соответствует требованиям к производительности. Программный код понятен, лаконичен и хорошо документирован. Можно ли сделать лучше? Конечно, но и так достаточно хорошо – поэтому остановитесь. Заявите о своей победе и переходите к следующей задаче.

Я считаю, что стремление к идеальному дизайну и идеальной реализации приводит к излишне усложненным и запутанным решениям, которые в конечном итоге затрудняют сопровождение системы.

Некоторые этюды в этой книге предостерегают проектировщиков от излишних абстракций и избыточной сложности. Почему мы не довольствуемся простыми решениями? Потому что мы стремимся к идеальному решению! Зачем еще архитектору вводить сложность в работоспособное решение, если не для устранения субъективных несовершенств в более простом варианте?

Помните, что разработка приложений – не конкурс красоты. Перестаньте выискивать недочеты и тратить время на поиски совершенства.

Грег Найберг (Greg Nyberg) – независимый консультант в области J2EE с 18-летним опытом проектирования, построения, тестирования и развертывания крупномасштабных транзакционных приложений, таких как системы оформления предварительных заказов, центры приема звонков и потребительские веб-сайты. Является автором справочника «WebLogic 6.1 Server Workbook for Enterprise JavaBeans, 3-е издание» (O'Reilly) и ведущим автором книги «Mastering WebLogic Server» (Wiley).

Остерегайтесь «хороших идей»

Грег Найберг

Хорошие идеи убивают проекты. Иногда смерть наступает быстро, но чаще это медленное, мучительное умирание, причиной которого служат сорванные сроки и лавины программных ошибок.

Вы знаете, о каких хороших идеях я говорю: соблазнительные, очевидные, абсолютно безвредные на первый взгляд – «ничего-страшного-не-будет-если-мы-попробуем». Обычно они приходят в голову кому-либо в команде где-то в середине жизненного цикла проекта, когда все вроде бы идет хорошо. Работа движется в бодром темпе, начальное тестирование проходит как положено, дата выпуска выглядит непоколебимой – жизнь прекрасна.

Тут у кого-то появляется «хорошая идея», вы с ней соглашаетесь – и вот вы уже переделываете проект под свежую версию Hibernate, чтобы воспользоваться ее новейшими возможностями, или используете AJAX на некоторых веб-страницах, потому что разработчик показал пользователям, как круто это смотрится, или пересматриваете архитектуру базы данных, чтобы задействовать те возможности по работе с XML, которые предлагает СУБД. Вы говорите руководителю проекта, что для реализации этой «хорошей идеи» понадобится еще несколько недель, однако изменения затрагивают больший объем кода, чем предполагалось, и график начинает трещать по швам. Вдобавок, приняв первую «хорошую идею», вы, как в поговорке, «выпустили джинна из бутылки»: вскоре на свет появляются новые «хорошие идеи», а вам уже гораздо труднее отказать (а джинн тем временем уже выглядывает из всех щелей).

Самая коварная особенность «хороших идей» состоит в том, что они «хороши». «Плохую» идею распознает и отвергнет кто угодно – в проект проникают именно «хорошие» идеи, раздувая его масштаб и повышая сложность,

а также требуя лишних усилий на включение в приложение того, что не нужно для достижения бизнес-целей.

Вот несколько ключевых фраз, свидетельствующих об опасности:

- «Разве не круто будет, если...». На самом деле сигналом тревоги может быть любое предложение со словом «круто».
- «Только что вышла версия XXX библиотеки YYY. Нам надо перейти на новую версию!»
- «Знаешь, раз уж мы работаем над ZZZ, нам стоит заодно переработать XXX...»
- «XXX – действительно мощная технология! Возможно, мы сможем применить ее в...»
- «Послушай, *<здесь_ваше_имя>*, я тут размышлял о дизайне нашей системы – и мне пришла в голову мысль...»

Хорошо, хорошо – возможно, в последнем пункте я перегибаю палку. Но вы все равно должны остерегаться «хороших идей», которые способны убить ваш проект.

Биография автора приведена на стр. 155.

Хороший контент порождает хорошие системы

Зубин Вадья

Я ВИДЕЛ ВЕЛИКОЕ МНОЖЕСТВО ИНИЦИАТИВ, в которых внимание было сосредоточено на требованиях, дизайне, разработке, безопасности, сопровождении, но только не на сущности системы – данных. Такая ситуация особенно часто встречается в контентных системах (content-based systems), где данные – это информация, доставляемая потребителю в виде неструктурированного или слабо структурированного контента. Именно качество контента часто отличает актуальную систему от бесполезной.

Контент – король. Контент – сеть. Контент – интерфейс. В современном мире, пронизанном многочисленными информационными связями, качество контента все чаще определяет успех или неудачу. FaceBook против Orkut, Google против Cuil, NetFlix против BlockbusterOnline... список сражений, выигранных и проигранных на поле контента, можно продолжать до бесконечности. Кто-то может возразить, что аспекты, касающиеся контента, не относятся к проблематике архитектора ПО, но я считаю, что следующее десятилетие докажет обратное.

Оценка контента должна стать частью процесса проектирования новой системы. Простого проектирования эффективной модели предметной области/объектов/данных недостаточно.

Проанализируйте весь доступный контент и оцените его значимость по следующим критериям:

- Достаточно ли доступного количества контента? Если нет, как получить «критическую массу»?
- Достаточно ли актуальна содержащаяся в нем информация? Если нет, как улучшить скорость поступления?

- Все ли возможные каналы распространения контента изучены? RSS-трансляции, электронная почта, бумажные бланки – все это является возможными каналами.
- Созданы ли эффективные входные потоки, упрощающие непрерывное поступление контента в систему? Одно дело – выявить ценный контент, и совсем другое – организовать его регулярное получение.

Несомненно, успех системы зависит от ее контента. Уделите в процессе проектирования достаточное внимание анализу ценности контента. Если результаты анализа окажутся неудовлетворительными, это тревожный признак, о котором следует посоветоваться с заинтересованными сторонами проекта. Я видел много систем, которые выполняли все обязательства по договору, соответствовали всем требованиям – но все равно потерпели неудачу, потому что этот очевидный аспект был проигнорирован. Хороший контент порождает хорошие системы.

Зубин Вадья (Zubin Wadia) – генеральный директор RedRock IT Solutions и технический директор ImageWork Technologies. Обладает разносторонней квалификацией в области программирования, владеет языками Basic, C, C++, Perl, Java, JSP, JSF, JavaScript, Erlang, Scala, Eiffel и Ruby. Специализируется на разработке решений из области автоматизации бизнес-процессов для компаний из списка Fortune Global 500 и правительственных учреждений США.

Бизнес и недовольный архитектор

Чед Лавинь

В КАРЬЕРЕ ЛЮБОГО АРХИТЕКТОРА НАСТУПАЕТ МОМЕНТ, когда становится ясно, что многие вопросы, с которыми он имеет дело, уже встречались на его пути раньше. Сменяются проекты и области, но проблемы остаются прежними. На этой стадии мы можем опереться на свой опыт, чтобы создавать решения быстрее и оставлять максимум времени для более интересных задач. Мы уверены в своих решениях, мы выдаем их в полном соответствии со своими обещаниями. Наступает своего рода гомеостаз. Именно в такие моменты легко совершить огромную ошибку – решить, что вы знаете достаточно много для того, чтобы отныне говорить больше, чем слушать. Это ошибочное решение обычно сопровождается цинизмом, нетерпимостью и гневом по отношению к тем «низшим умам», которые смеют оспаривать ваше выдающееся понимание всех вопросов – технических и прочих.

В худшем своем проявлении самонадеянность просачивается в сферу деловых отношений. Это отличная возможность навсегда загубить свою карьеру. Именно бизнес оправдывает само наше существование. Возможно, нам несколько неприятно сознавать этот факт, но все же не стоит упускать его из виду. Мы живем для того, чтобы служить потребностям бизнеса, а не наоборот. Умение слушать представителей бизнеса, которые нанимают нас для решения своих задач, и понимать эти задачи – один из самых важных наших навыков. Вам когда-нибудь доводилось нетерпеливо дожидаться, пока бизнес-аналитик закончит говорить, чтобы приступить к изложению своей позиции? Скорее всего, его точку зрения вы при этом не восприняли. Относитесь к экспертам в предметной области с тем уважением, которое хотели бы видеть по отношению к себе; крайне нежелательно, чтобы они считали вас непробиваемым упрямцем. Если они начнут избегать вас, вы станете катализатором нарушения информационного обмена и по сути выроете могилу

собственному проекту. Помните: когда вы говорите, то сможете услышать только то, что уже знаете. Никогда не считайте себя умным настолько, что другие уже не могут сообщить вам ничего стоящего.

Слушая, мы часто не соглашаемся с услышанным по поводу ведения бизнеса. Это нормально. Мы можем вносить рационализаторские предложения и определенно должны это делать. Но если по итогам дня ситуация не изменилась и вы по-прежнему не согласны с тем, как работает бизнес, дело плохо. Не позволяйте себе превратиться в раздражительного гения, который тратит все свое время на попытки поразить других остроумными снисходительными замечаниями о том, как безобразно управляется эта компания. Вы не произведете ни на кого впечатления. Ваши собеседники уже встречались с такими личностями раньше и относятся к ним с неприязнью. Одним из важнейших качеств хорошего архитектора является страстное отношение к своей работе, но эту страсть не следует разменивать на отрицательные эмоции. Научитесь принимать разногласия и двигаться дальше. Если разногласия оказываются слишком большими и вам приходится постоянно пререкаться с представителями бизнеса, найдите компанию, с которой вам будет проще поладить, и работайте на нее. Так или иначе, постарайтесь установить хорошие отношения с бизнесом и берегите их, не позволяйте своему эго вредить им. Это сделает вашу работу более приятной и эффективной.

Биография автора приведена на стр. 125.

Проверяйте решения на прочность по ключевым характеристикам

Стивен Джонс

Изначально архитектура приложения формируется на основании заданных бизнес-требований, выбранных или уже применяемых технологий, диапазона производительности, ожидаемых объемов данных и финансовых ресурсов, выделенных для построения, развертывания и управления системой. Решение, каким бы оно ни было, должно соответствовать требованиям из этого набора либо превосходить их – и при этом успешно работать в современных условиях (иначе это попросту не решение).

Теперь возьмите свое решение и посмотрите, какие проблемы появятся, если «растянуть» ключевые характеристики.

Анализ такого рода выявляет архитектурные ограничения, которые проявятся в том случае, если система станет, например, чрезвычайно популярной и количество ее пользователей превысит начальные ожидания, или увеличится ежедневное количество транзакций, или потребуется хранить данные в течение полугода вместо изначально предусмотренной недели. «Растягивайте» ключевые характеристики сначала по отдельности, а затем в сочетаниях друг с другом, чтобы выявить те невидимые ограничения, которые скрыты в исходной архитектуре.

«Растяжение» ключевых характеристик поможет вам:

- Понять, выдержит ли запланированная инфраструктура такие изменения и где именно заложены ограничения. Если работа инфраструктуры будет нарушена, этот процесс позволяет узнать, где именно произойдут нарушения. Далее можно сообщить полученную информацию владельцу приложения или же учесть возможность обновления при покупке инфраструктуры.

- Убедиться, что в сутках хватит часов для обработки данных с заданной пропускной способностью с запасом для пиковых нагрузок и компенсации простоев. Решение, не способное завершить дневной объем работы за день и вынужденное переносить работу на выходные дни, в которые нагрузка снижается, лишено будущего.
- Убедиться в том, что механизмы доступа к данным останутся работоспособными при масштабировании системы. Решение, работающее с недельным объемом данных, может не справиться с объемом, накопленным за полгода.
- Проверить, как при повышении рабочей нагрузки приложение распределит ее на дополнительное оборудование (если оно потребуется), и проанализировать пути перехода при повышении нагрузки. Анализ переходных путей перед развертыванием приложения может повлиять на механизмы хранения данных и их структуру.
- Убедиться, что приложение сможет нормально восстанавливаться после сбоев при возрастании объема данных и/или разделении данных в пределах расширенной инфраструктуры.

На основании этого анализа выявляются проблемные элементы архитектуры системы, требующие доработки. Доработка обойдется дешевле, пока дизайн остается виртуальным, технические решения еще не зафиксированы, а репозитории не заполнены рабочими данными.

Стивен Джонс (Stephen Jones) проектирует решения для ***Tier-1Telco Billing*** *и сопутствующие крупномасштабные процессы для таких компаний, как Telstra и Optus (Австралия), а также AT&T (США). В частности, он занимался исходной реализацией систем тарификации* ***Telco, перепроектированием*** *системы опротестования счетов и борьбы с фальсификациями и более двух лет управлял службой круглосуточной поддержки Telstra.*

Проектируйте только то, что можете запрограммировать

Майк Браун

Архитекторов часто подстерегает искушение создать изощренные абстракции и дизайн для элегантного решения текущей задачи. Еще более соблазнительно выглядит включение в проект новых технологий. Но в конечном итоге кому-то придется реализовывать ваши идеи, и архитектурная акробатика, на которую вы обрекаете разработчиков, отразится на ходе проекта.

Обдумывая архитектуру для своих проектов, вы должны представлять себе объем работы, необходимый для реализации каждого элемента вашего дизайна. Если какой-то элемент вы уже разрабатывали ранее, вам будет намного проще оценить объем работы.

Не применяйте при проектировании шаблоны, которые вы не использовали прежде. Не полагайтесь на инфраструктуры, с помощью которых вы никогда ничего не разрабатывали. Не используйте сервер, который вам не доводилось настраивать. Если архитектура зависит от элементов, с которыми вы никогда не имели дела лично, возникает целый ряд отрицательных побочных эффектов:

- Вы не представляете себе кривую обучения, которую предстоит одолеть вашим разработчикам. Не зная, сколько времени потребуется для изучения новой технологии, вы не сможете достоверно оценить время реализации.
- Вы не знаете, какие «ловушки» могут подстерегать вас при использовании этих элементов. На практике все обязательно пойдет не так гладко, как на демонстрации, которую проводил эксперт в этой технологии. Если прежде вы никогда не работали с технологией, вас неизбежно ждут неприятные сюрпризы.

- Вы лишитесь доверия своих разработчиков. Если вы не можете уверенно ответить на вопросы, относящиеся к вашему дизайну, разработчики быстро перестанут доверять вам и вашим творениям.
- Вы подвергаетесь излишнему риску; нехватка знаний ставит жирный вопросительный знак на ключевых элементах решения. Никто не захочет начинать проект, имея в багаже огромный ненужный риск.

Как же архитектор должен подходить к освоению новых инфраструктур, шаблонов и серверных платформ? Об этом вам расскажет этюд «Архитектор –прежде всего разработчик».

Биография автора приведена на стр. 141.

«Что значит имя?», или Как роза превращается в капусту

Сэм Гардинер

Я СЛУЧАЙНО ПОДСЛУШАЛ РАЗГОВОР АРХИТЕКТОРОВ, которые принимали решение о необходимости дополнительных уровней в их архитектуре. Они были правы, но, как это нередко случается, подошли к делу немного не с той стороны. Они пытались создать инфраструктуру, которая включала бы в себя бизнес-логику. Вместо того чтобы решать конкретную задачу, они задумали создать инфраструктуру, которая инкапсулирует базу данных и создает объекты. И при этом она должна была использовать объектно-реляционные отображения. И сообщения. И веб-службы. И должна была делать массу других Классных Штук.

К сожалению, еще не было точно известно, какие Классные Штуки она должна вытворять, поэтому архитекторы не знали, как ее назвать. Им пришлось затеять небольшой конкурс на лучшее имя. В подобный момент вы должны понять, что столкнулись с проблемой: если вы не знаете, как что-то назвать, то вы не знаете, что это такое. А если вы не знаете, что это такое, то вы не сможете сесть и написать программный код.

В нашем конкретном случае быстрый просмотр истории версий исходного кода выявил всю глубину проблемы. Конечно, в нем оказалось множество пустых «реализаций» интерфейсов! Но самое смешное, что даже без нормального кода имена уже менялись трижды. Сначала было выбрано имя ClientAPI (причем под «клиентом» имелся в виду заказчик, а не клиент в контексте модели «клиент–сервер»), а последняя версия называлась ClientBusinessObjects. Воистину замечательное имя: туманное, предельно широкое и сбивающее с толку.

Конечно, имя – это всего-навсего указатель. Как только все участники будут знать, что имя – это просто имя, а не архитектурная метафора, вы сможете

двигаться дальше. Но если вы не можете договориться о выборе имени, которое было бы достаточно конкретным, чтобы вы сразу могли понять, подходит оно или нет, то вам будет сложно даже приступить к решению задачи. Все проектирование направлено на реализацию намерений (например, быстрота, гибкость, экономичность), а имена отражают эти намерения.

Если вы не в состоянии что-либо назвать, то не сможете это и реализовать. Если вы меняете имя уже в третий раз, приостановите свою деятельность и попытайтесь понять, что же вы собираетесь построить.

После продолжительных развлечений с компьютерами (от написания игр на Basic на компьютере BBC до применения Pascal, Mathematica и LabVIEW для обработки экспериментальных данных, которые хранились в «базе данных» из скрепленных скотчем папок) Сэм Гардинер (Sam Gardiner) перешел к профессиональной разработке ***ПО. Он работает в сфере программирования на протяжении шести лет.***

Четко определенные задачи решаются качественно

Сэм Гардинер

Реальная практика программирования представляет собой отнюдь не решение задач, которые дал вам кто-то другой. Конечно, в учебном классе вы должны были решать задачу двоичной сортировки, полученную от преподавателя. Но в реальном мире лучшие архитекторы заняты не решением сложных задач, а поиском для них обходных путей. Их искусство проявляется в умении проложить границы между разнотипными и расплывчатыми задачами, чтобы сделать их четко определенными и автономными.

Архитектор вникает в мешанину концепций, данных и процессов и разбивает их на меньшие фрагменты. Важнейшим качеством таких фрагментов является четкая определенность задачи, что позволяет решить их законченными и четко определенными фрагментами системы. Основные свойства фрагментов задачи таковы:

- Внутренняя связность: фрагмент является концептуально цельным, то есть все его задачи, данные и функциональные возможности связаны друг с другом.
- Изолированность: фрагменты концептуально нормализованы, то есть практически не перекрываются друг с другом.

Подобно человеку с хорошей ориентацией на местности, который подсознательно знает, где находится в данный момент, человек, в совершенстве владеющий методом фрагментирования задачи, может даже не осознавать, что использует его, – ему просто кажется разумным разбить задачи, данные и функциональные возможности так, чтобы сформировать удобную логическую границу или интерфейс. (Естественно, речь идет не об интерфейсах из объектно-ориентированных языков, а о границах системы.)

Например, реляционная СУБД имеет очень хорошую границу. Она работает практически с любыми данными, которые могут быть преобразованы в поток байтов, обеспечивая структурирование, поиск и выборку этих данных. Все просто.

Здесь интересно то, что решение четко определенной задачи неизменно. Возможно, через пять лет к нему будет приделан веб-интерфейс, а через пятьдесят – телепатический, но ядро системы изменять не понадобится. Система стабильна, потому что четко определена решаемая с ее помощью задача.

Конечно, код сам по себе должен быть простым и понятным, но при решении хорошо формализованной задачи этого проще добиться благодаря отсутствию всевозможных «исключительных случаев». Аккуратный код хорош прежде всего тем, что его проще тестировать и анализировать, а это естественным образом ведет к весьма высокому качеству реализации. Если у вас нет проблем с кодом, вы сможете направить усилия на то, что обычно остается невидимым для пользователя, например на реализацию надежного механизма передачи сообщений, распределенные транзакции или повышение производительности за счет применения многопоточности (и даже низкоуровневых языков, например, ассемблерных вставок). Так как задача остается неизменной, вы можете сосредоточиться на совершенствовании реализации до того уровня, на котором качество станет отличительной чертой вашей системы.

Стабильность задачи позволяет вам создать систему со стабильным дизайном; стабильность дизайна позволяет создавать приложения высочайшего качества.

Биография автора приведена на стр. 167.

Необходимо усердие

Брайан Харт

Работу архитектора часто изображают как деятельность, основанную исключительно на изобретательности и творческом подходе к решению задач. Изобретательность – важнейшее свойство хорошего архитектора, но не менее важным качеством является усердие. Оно может проявлять себя по-разному, но в конечном счете сводится к упорству в достижении цели, а также к тому, чтобы уделять должное внимание каждой задаче и каждому архитектурному аспекту системы.

Усердие идет рука об руку с практичностью. Успешная практическая работа архитектора во многом обыденна. Эффективные архитекторы часто ведут ежедневные и еженедельные контрольные списки, которые напоминают им то, что они и так знают теоретически, но еще не делают по привычке. Без таких контрольных списков и напоминаний проект может быстро застопориться, потому что нехватка усердия позволяет архитектору обойти или нарушить известные теоретические принципы. В ходе ретроспективного анализа неудачных проектов важно понять, что в большинстве случаев крах произошел не из-за некомпетентности, а из-за недостатка усердия и чувства срочности.

Кроме того, быть усердным означает для архитектора преуспеть в обманчиво простом деле принятия и выполнения обязательств. Такие обязательства часто оказываются весьма разнородными и охватывают широкий диапазон ограничений и ожиданий. Примеры:

- Соблюдение финансовых и временных ограничений заказчика.
- Выполнение всей работы, которая делает архитектора эффективным (а не только той, которая ему нравится).
- Использование конкретных процессов/методологий.
- Принятие ответственности.

Вот что говорит Атул Гаванде (Atul Gawande) в своей замечательной книге «Better: A Surgeon's Notes on Performance» (Быть лучше: записки хирурга о профессиональном мастерстве) (Metropolitan Books) об усердии в медицинском сообществе:

> «Добиться истинного успеха в медицине нелегко. Это требует силы воли, внимания к мелочам и творческого подхода. Но из своего пребывания в Индии я вынес урок: это может сделать кто угодно и где угодно. Найдется очень мало мест с еще более трудными условиями. Но и там встречались поразительные успехи... Я понял, что стать лучше всегда возможно. Для этого не обязательна гениальность. Необходимо усердие. Необходимы моральные принципы. Необходима изобретательность. И самое главное – желание попытаться.»

Брайан Харт (Brian Hart) – ведущий консультант по ***CGI, специалист в области*** *информационных технологий и бизнес-процессов. Брайан участвовал в проектировании приложений J2EE, главным образом в государственном секторе и на уровне местных властей. В сфере разработки программного обеспечения работает с 1997 года.*

Отвечайте за свои решения

И Чжоу

Архитекторы программного обеспечения должны отвечать за свои решения, так как они влияют на ход проекта гораздо сильнее, чем большинство других сотрудников организации. Анализ программных проектов показывает, что более двух третей из них заканчиваются либо полным крахом, либо выпуском продукта с нарушениями (перенос срока выпуска, перерасход бюджета, недовольство заказчика). Глубинные причины неудач чаще всего связаны с неверными решениями, принятыми архитектором, или с неверным осуществлением правильных архитектурных решений.

Как стать ответственным архитектором, принимающим эффективные архитектурные решения?

Во-первых, вы должны хорошо знать процесс принятия решений независимо от того, относится ли он к гибким или традиционным методологиям. Нельзя сказать, что архитектурное решение принято, пока не выполнены следующие два условия:

- Решение изложено в письменном виде, поскольку архитектурные решения редко бывают тривиальными. Они должны быть четко обоснованными и отслеживаемыми вплоть до источника.
- Информация о решении передана исполнителям, а также тем людям, которых оно затронет (прямо или косвенно). Передача информации формирует единое для всех понимание сути решения.

Во-вторых, регулярно возвращайтесь к анализу своих архитектурных решений. Сопоставляйте результаты своих решений с исходными ожиданиями. Определяйте, какие архитектурные решения доказали свою правильность, а какие нет.

В-третьих, обеспечьте выполнение своих архитектурных решений. Во многих программных проектах архитектор участвует только в фазе проектирования, а затем переходит на другой проект (или же контракт на оказание консультационных услуг подходит к концу). Как в такой ситуации он может проследить за правильностью реализации своих тщательно проработанных архитектурных решений? Без авторского контроля за исполнением решения в лучшем случае останутся благими намерениями.

Наконец, доверьте принятие некоторых решений другим специалистам в предметной области задачи. Многие архитекторы ошибочно полагают, что они должны принимать все архитектурные решения без исключения, то есть играют роль экспертов «в чем угодно». В действительности универсальных технических гениев не бывает. У каждого архитектора есть области, в которых он хорошо разбирается, области, в которых он что-то знает, и области, в которых он попросту некомпетентен. Грамотный архитектор делегирует другим принятие решений в тех областях, в которых плохо ориентируется сам.

И Чжоу (Yi Zhou) в настоящее время работает главным архитектором программного обеспечения в широко известной биотехнологической компании, где проектирует программные платформы для медицинских устройств и персонализации управления ходом заболевания. Он обладает почти 20-летним опытом, охватывающим все стадии цикла разработки ПО, и специализируется на согласовании бизнеса и технологий, стратегическом планировании, совершенствовании процессов, проектировании архитектур и инфраструктур, создании проектных команд и управлении ими, а также на консультировании.

Не мудрствуйте

Эбен Хьюит

Интеллект, изобретательность, вдумчивость, широта и глубина познаний, любовь к точности – качества, похвальные для любого человека и особенно ценные для архитекторов.

Однако изощренность ума имеет побочный смысловой оттенок. Да, она подразумевает умение моментально найти решение, которое выведет вас из сложной ситуации, но в конечном счете это решение опирается на фокус, ловкость рук, трюк сродни игре в «наперсток». Вспомните умных спорщиков, с которыми вы вместе учились, – они всегда умели поиграть на семантике или использовать логические слабости в позиции оппонента, чтобы победить в споре.

Умные программы обходятся дорого, сложны в сопровождении и ненадежны. Не мудрствуйте. Постарайтесь быть как можно примитивнее, но создайте при этом подходящий дизайн. Самый подходящий дизайн никогда не бывает умным. Если вам кажется, что без ухищрений не обойтись, значит, задача неправильно структурирована, – проанализируйте ее заново. Пересматривайте постановку задачи до тех пор, пока вы не сможете снова действовать примитивно. Работайте на уровне грубых набросков; придерживайтесь общих решений. Забудьте о новомодных веяниях. Умный архитектор должен действовать как можно примитивнее.

Именно изощренность ума позволяет нам «обмануть» нежизнеспособную систему и заставить ее работать. Не становитесь адвокатом, который своим красноречием «спасает» программу на техническом суде. Вы не Руб Голдберг

и не Макгайвер[1], готовый в любой момент построить умопомрачительную конструкцию из скрепок, жевательной резинки и динамитной шашки. Выкиньте все лишнее из головы; подойдите к решению задачи без своих обширных познаний в области замыканий, обобщений и управления поколениями объектов в «куче». Иногда, конечно, все перечисленное действительно нужно для решения задачи, но намного реже, чем кажется на первый взгляд.

Чем примитивнее решение, тем больше разработчиков справятся с его реализацией и сопровождением. В примитивных решениях каждый компонент выполняет ровно одну функцию. Они быстрее создаются и легче модифицируются в будущем. Они наследуют оптимизацию от структурных блоков, которые используются при их построении. Такие решения уже на бумаге выглядят жизнеспособными, и при первом взгляде на них ощущается их элегантность и простота. В то же время умный, изощренный дизайн запутывает общую картину в хитросплетении деталей и разваливается от малейшего прикосновения.

*Эбен Хьюит (Eben Hewitt) возглавляет группу архитекторов в национальной компании розничной торговли с миллиардным оборотом, где в настоящее время занимается проектированием и реализацией сервис-ориентированной архитектуры (SOA). Он является автором книги «**Java SOA Cookbook**», которая будет скоро опубликована издательством O'Reilly.*

[1] Руб Голдберг (Rube Goldberg) – американский карикатурист, скульптор, писатель и изобретатель, широко известный серией комиксов, где изображались очень сложные устройства для выполнения простых задач (получившие название «машины Руба Голдберга»). Ангус Макгайвер (Angus MacGyver) – персонаж одноименного американского телесериала, секретный агент, отличительная черта которого – умение применять обширные научные познания для изобретательного использования обыденных вещей в критической ситуации. – *Примеч. ред.*

Выбирайте оружие тщательно и не спешите его менять

Чед Лавинь

Любой архитектор, будучи закаленным ветераном проектирования и реализации, обладает целым арсеналом «оружия», которое он раз за разом успешно применяет в своей работе. По тем или иным причинам эти технологии завоевали наше расположение и сумели подняться на первые позиции в нашем личном списке предпочтений. Скорее всего, они заслужили свое место в арсенале, победив в ходе ожесточенной конкуренции. Однако, несмотря на это, их положению постоянно угрожают многочисленные новые технологии. У нас часто возникает соблазн отложить свое испытанное оружие, чтобы опробовать новые альтернативы... но не стоит торопиться. Отказываться от проверенных инструментов в пользу других технологий, которые еще не прошли аналогичной проверки, – дело весьма рискованное.

Это вовсе не означает, что технологии, попавшие в список избранных, получили пожизненное содержание; и конечно, не означает, что вы можете зарыть голову в песок и не обращать внимания на новые достижения в области разработки ПО. Для каждой технологии наступает время, когда ее приходится заменять. Технологии развиваются быстро, и более совершенные решения появляются постоянно. Мы как архитекторы должны идти в ногу со временем, но не обязаны первыми хвататься за каждую недозревшую технологию. Первопроходцы новых технологий обычно не только не получают особых преимуществ, но часто сталкиваются с рядом препятствий.

Чтобы риск, связанный с выбором новой технологии, был оправданным, она должна предлагать преимущества, поднимающие ее на качественно иную высоту по отношению к предшественницам. Многие новые технологии обещают подобный прорыв, но лишь в редких случаях действительно могут его обеспечить. Взглянуть на новую технологию и увидеть ее технические преимущества легко, но убедить в них заинтересованные стороны проекта

зачастую весьма непросто. Прежде чем вы решите прокладывать дорогу, используя новую технологию, спросите себя, какую пользу это принесет бизнесу. Если с точки зрения бизнеса выгода настолько мала, что ее никто не заметит, пересмотрите свое решение.

Еще одним важным фактором являются затраты, обусловленные недостатками новой технологии. Эти затраты могут быть довольно высокими и плохо прогнозируемыми. Работая со знакомыми технологиями, вы хорошо представляете себе их больные места. Наивно было бы полагать, что новая технология избавлена от «ловушек». Возникновение проблем, с которыми вы ранее не сталкивались, разрушит все ваши оценки. О затратах, связанных с реализацией решений на базе знакомых технологий, вы осведомлены гораздо лучше.

Наконец, необходимо также учесть перспективность новой технологии. Было бы здорово уметь запросто выявлять и отбирать самые перспективные технологии, однако это не так легко. Хорошие технологии не всегда побеждают. Попытки выявить победителей на ранней стадии – азартная игра, которая не приносит большого дохода. Подождите, пока шумиха уляжется, и посмотрите, докажет ли новая технология свою полезность. Вы увидите, что многие новинки попросту уходят в небытие. Не ставьте под угрозу свой проект применением технологии, у которой, возможно, нет будущего.

Выбор технологии, используемой для решения задачи, – важная часть работы архитектора программного обеспечения. Тщательно выбирайте свое оружие и не спешите его менять. Дайте вашим прежним успехам лечь в основу будущих достижений и расширяйте свой арсенал технологий постепенно и разумно.

Биография автора приведена на стр. 125.

Ваш клиент – не ваш клиент

Эбен Хьюит

Принимая участие во встречах по сбору требований в ходе проектирования программных продуктов, представьте себе, что вашим клиентом является не ваш клиент. На самом деле это совсем несложно, потому что это правда.

Ваш клиент – не ваш клиент. Вашим клиентом является клиент вашего клиента. Если выигрывает клиент вашего клиента, то выигрывает и ваш клиент. А это означает, что выигрываете вы.

Если вы пишете приложение электронной коммерции, позаботьтесь о том, что потребуется посетителям, которые будут совершать покупки на вашем сайте. Им потребуется безопасная передача данных. Им потребуется шифрование хранимых данных. Ваш клиент, возможно, даже не упомянет об этих требованиях. Если вы знаете, что ваш клиент упустил что-то, что нужно его клиентам, займитесь этой проблемой и объясните ему, почему вы это сделали.

Если ваш клиент намеренно и осознанно игнорирует важные аспекты, нужные его клиенту (а это время от времени случается), возможно, от проекта стоит отказаться. Если ваш клиент не желает ежегодно платить за SSL и предпочитает хранить данные кредитных карт в текстовом виде, потому что это удешевит разработку системы, будет ошибкой просто согласиться. Соглашаясь работать над реализацией заведомо плохих требований, вы уничтожаете клиентов своего клиента.

Встречи по сбору требований – это не диспуты о реализации. Не позволяйте своему клиенту углубляться в вопросы реализации, если только речь не идет об абсолютно необходимых или хорошо понятных задачах. Пусть ваш клиент опишет свой идеал, свою концепцию и цели, вместо того чтобы диктовать решение (и вообще использовать технические термины).

Как обеспечить такую дисциплину на встречах с клиентом? На самом деле эта задача только выглядит сложной. Помните, что вы должны заботиться о клиенте своего клиента. Хотя чек выписывает ваш клиент, вы должны четко дать ему понять, что будете следовать лучшим рекомендациям и можете сделать то, что действительно необходимо клиенту, а не то, что он считает таковым. Конечно, это повлечет за собой длительные споры и потребует умения четко формулировать, что именно вы делаете и почему.

Как и многое в жизни, сказанное лучше всего иллюстрируется стихами. В 1649 году Ричард Лавлейс написал стихотворение «Лукасте по поводу ухода на войну». Оно заканчивается строками: «Ведь если бы я предал честь, я предал бы тебя».

Мы предаем наших клиентов, когда игнорируем интересы их клиентов.

Биография автора приведена на стр. 175.

Все будет не так, как задумано

Питер Гиллард-Мосс

Все будет не так, как задумано. Очень легко попасть в ловушку и преисполниться уверенности в том, что реализация будет полностью соответствовать вашим планам, после того как вы потратили массу времени на проектирование. Детальное проектирование создает иллюзию, что вы предусмотрели все до мельчайших подробностей. Чем подробнее детали, чем глубже ваши исследования, тем больше вы доверяете своему дизайну. Но это впечатление обманчиво: все получится не так, как задумано.

Реальность такова, что, как бы глубоко вы ни прорабатывали дизайн, как бы тщательно ни обдумывали детали, результат все равно будет отличаться от того, каким вы его себе представляли. Что-нибудь непременно произойдет – и в дизайн вмешается какой-либо внешний фактор: неверная информация, ограничение, странное поведение чужого кода... А может быть, вы где-то ошибетесь: упущение, неверное предположение, какой-то пропущенный нюанс... Или что-нибудь изменится – требования, технология, – или кто-то найдет лучшее решение™.

Мелкие изменения в дизайне накапливаются, и вскоре выясняется, что необходимо внести одно большое изменение. И вот ваша исходная концепция разбивается вдребезги, и вам приходится снова браться за карандаш и бумагу. Вы решаете, что проблема требует более тщательного, более подробного проектирования, усердно трудитесь – и добиваетесь более четкого, более глубокого и более совершенного ви́дения.

А затем повторяется та же история. Снова там и сям появляются изменения, подрывающие ваш замысел; разработчики накладывают все новые заплатки, пытаясь хоть как-то удержать расползающийся по швам дизайн, но

в конечном итоге только разваливают его окончательно. И вы восклицаете: «Конечно же, ошибки будут – ведь система для этого не предназначалась!»

Проектирование – это исследовательский процесс; в ходе реализации обнаруживается новая информация, которую часто бывает невозможно предсказать заранее. Принимая как факт, что проектирование – это эмпирический процесс в постоянно изменяющемся мире, мы понимаем, что процесс проектирования должен быть гибким и непрерывным. Если вы упорно цепляетесь за исходный дизайн и пытаетесь втиснуть действительность в его рамки, итог предопределен. Вам следует осознать, что *все всегда получается не так, как задумано.*

Питер Гиллард-Мосс (Peter Gillard-Moss) – сотрудник ***ThoughtWorks и ме****меолог широкого профиля из США. Работает в сфере информационных технологий с 2000 года; участвовал во многих проектах, от общедоступных медиасайтов и электронной коммерции до банковских приложений и корпоративных интрасетей.*

Выбирайте инфраструктуры, хорошо сочетающиеся с другими

Эрик Готорн

При выборе программных инфраструктур, которые будут положены в основу вашей системы, необходимо учитывать не только качество и возможности каждой инфраструктуры (framework), но и то, как они будут взаимодействовать друг с другом и насколько легко будет адаптировать их к новым программным элементам, которые вам, возможно, придется добавить в ходе эволюции системы. Из этого следует, что выбранные инфраструктуры должны быть неперекрывающимися, простыми, компактными и специализированными.

Лучше всего, если каждая инфраструктура или сторонняя библиотека будет относиться к отдельной логической области и решать обособленную задачу, не вторгаясь в область других задействованных инфраструктур.

Обязательно изучите перекрытия логических областей и функциональности инфраструктур-кандидатов. Если понадобится, нарисуйте диаграмму Венна. Две модели данных, существенно перекрывающиеся в предметной области, или две реализации, решающие очень похожие задачи немного различающимися способами, порождают излишнюю сложность: придется находить соответствия между концептуальными различиями или представлениями либо приделывать заплатки из неуклюжего связующего кода. В итоге вы, скорее всего, получите не только громоздкий связующий код, но и сведете функциональные и репрезентативные возможности системы к наименьшему общему знаменателю двух инфраструктур.

Чтобы снизить вероятность функционального перекрытия, выбирайте инфраструктуры с высоким отношением полезной нагрузки к балласту в контексте требований вашей системы. Полезная нагрузка – это функциональность или возможности представления данных, необходимые вашему проекту,

а балласт – то, насколько инфраструктура стремится охватить все что можно и решить все задачи на свете. Требует ли инфраструктура смешивать управление данными и их представление? Как сильно ее модель данных или набор пакетов/классов выходят за пределы того, что необходимо вашей системе? Придется ли вам присягнуть ей на верность и впредь ограничить выбор инфраструктур теми, что соответствуют ее рамкам? Ограничивает ли ее избыточная сложность возможности взаимодействия? Если уж инфраструктура отягощена большим количеством балласта, она должна обеспечивать хотя бы 75% необходимой функциональности проекта.

Система должна состоять из неперекрывающихся инфраструктур – блистательных в своей области, но при этом простых, компактных и гибких.

Эрик Готорн (Eric Hawthorne) профессионально занимается созданием архитектуры, проектированием и разработкой объектно-ориентированных программ и распределенных систем с 1998 года. Первые 10 лет его карьеры прошли в Macdonald Dettwiler – канадской системотехнической компании, где среди прочего он имел возможность поучиться архитектурным тонкостям у самого Филиппа Крачтена (Philippe Kruchten).

Подготовьте убедительное экономическое обоснование

И Чжоу

Приходилось ли вам как архитектору программного обеспечения сталкиваться с трудностями финансирования архитектурных проектов? Преимущества того или иного архитектурного решения очевидны для архитекторов, но для заинтересованных в проекте сторон это зачастую не так. Психология массового сознания говорит, что большинство людей склонно верить лишь тому, что видит собственными глазами. Однако на ранней стадии проекта трудно продемонстрировать заинтересованным сторонам что-то, что может стать убедительным доводом в пользу качественной проработки программной архитектуры. Еще сложнее дело обстоит в отраслях, не связанных с программированием, где заинтересованные стороны обычно очень слабо разбираются в программных технологиях.

Психология массового сознания говорит также, что большинство людей считает воспринимаемое реальностью. А значит, если вы сможете управлять тем, как люди воспринимают предложенный вами подход к решению тех или иных архитектурных вопросов, вы практически гарантированно сможете управлять и их реакцией на ваше предложение. Как управлять восприятием заинтересованных сторон? Подготовьте для своей архитектуры надежное экономическое обоснование. Люди, которые финансируют ваши идеи, почти всегда руководствуются деловыми соображениями.

На протяжении своей карьеры я неоднократно применял следующий процесс из пяти шагов для успешного продвижения своих архитектурных решений:

1. *Сформулируйте ценностное предложение.* Составьте пояснительную записку относительно того, почему деловые интересы вашей организации требуют применения определенной программной архитектуры. Для

этого необходимо сравнить предлагаемые вами архитектурные решения с имеющимися решениями или возможными альтернативами. Основное внимание следует уделить возможности повышения производительности и эффективности бизнеса, а не замечательным свойствам технологий.

2. *Предложите количественные метрики.* Польза от предлагаемых вами решений должна быть измеримой (до разумной степени). Чем больше количественных показателей вы предъявите, тем убедительнее сможете обосновать свое утверждение о том, что предлагаемая архитектура обеспечит существенную отдачу. Чем раньше будут заданы метрики, тем проще вам будет управлять восприятием заинтересованных сторон – а это поможет вам убедить их в преимуществах своей архитектуры.
3. *Увяжите свои данные с традиционными бизнес-метриками.* Будет превосходно, если вы сумеете напрямую представить результаты технического анализа в денежном выражении. В конце концов, единственным неизменным параметром в традиционных бизнес-метриках являются деньги. Если вы недостаточно хорошо разбираетесь в финансовых тонкостях, возьмите себе в помощники бизнес-аналитика.
4. *Знайте, где остановиться.* Для этого необходимо подготовить план, в котором будут отражены ваши представления обо всех ключевых этапах работ и о том, какую ценность для бизнеса представляет каждый из них. Пусть заинтересованные стороны проекта сами решат, где следует остановиться. Если ценность каждого этапа для бизнеса будет значительной, вам, скорее всего, удастся добиться финансирования в полном объеме.
5. *Выберите подходящий момент.* Даже если вы выполните четыре предыдущих рекомендации и подготовите хорошее экономическое обоснование, ваша идея может провалиться из-за неудачного выбора момента ее подачи. Помню, одно из моих предложений долгое время не получало поддержки – до тех пор, пока другой проект не завершился полным крахом из-за плохой архитектуры. Разумно подходите к выбору момента.

Биография автора приведена на стр. 173.

Управляйте не только кодом, но и данными

Чед Лавинь

Управление исходным кодом и непрерывная интеграция – замечательные инструменты для управления процессами сборки и развертывания приложений. Однако изменения в схеме и в данных часто являются существенной частью этого процесса наравне с изменениями в исходном коде и требуют аналогичного управления. Если ваш процесс сборки и развертывания системы содержит список сложных операций, необходимых для обновления данных, берегитесь. Подобные списки всегда являются тревожным знаком. Выглядят они примерно так:

1. Вы создаете список сценариев, которые необходимо выполнить в заданном порядке.
2. Вы отправляете сценарии конкретному администратору базы данных по электронной почте.
3. Администратор копирует сценарии в каталог, из которого они должны запускаться заданием *cron*.
4. Вы проверяете журнал выполнения сценариев и молитесь, чтобы все сценарии выполнились успешно, потому что не знаете точно, что произойдет при их повторном выполнении.
5. Вы запускаете сценарии проверки и осуществляете выборочную проверку данных.
6. Вы проводите регрессионное тестирование приложения и смотрите, что перестало работать.
7. Вы пишете сценарии для вставки отсутствующих данных и исправления ошибок.
8. Вы повторяете шаги с 1 по 7.

Конечно, я преувеличиваю, но лишь слегка. Во многих проектах приходится выполнять подобные «акробатические трюки» для успешной миграции базы данных. Возникает такое чувство, что при создании плана миграции архитекторы по какой-то причине напрочь забывают о данных. В результате появляется ненадежный, выполняемый вручную процесс, состряпанный задним числом.

Сложная, запутанная процедура открывает много возможностей для сбоев. Ситуация усугубляется тем, что ошибки, вызванные изменениями схем или данных, не всегда обнаруживаются модульными тестами, которые являются частью ночной сборки. Эти ошибки любят заявлять о себе громко и эффектно сразу же после миграции. Решения проблем в базах данных сложнее проверить на правильность, а ручной «откат» базы данных – операция весьма трудоемкая. Ценность полностью автоматизированного процесса сборки, способного вернуть базу данных в известное состояние, становится особенно очевидной, когда вы используете его для исправления очень явных ошибок. Если вы не можете удалить базу данных и восстановить ее состояние, совместимое с конкретной сборкой приложения, вы сталкиваетесь с теми же проблемами, что и при невозможности быстрого восстановления кода.

Изменения в базе данных не должны разрывать «пространственно-временной континуум» сборки. Вы должны собирать все приложение, включая базу данных, как единое целое. Сделайте управление данными и схемой неотъемлемой частью автоматизированного процесса сборки и тестирования с самого начала и предусмотрите возможность отмены – все это окупится с лихвой. В лучшем случае это избавит вас от долгих часов тягостной, напряженной работы после случайной ошибки, допущенной поздним вечером. В худшем – позволит вашей команде уверенно продвигаться вперед при рефакторинге уровня доступа к данным.

Биография автора приведена на стр. 125.

Расплатитесь по техническим кредитам

Беркхардт Хафнагель

В ЖИЗНИ ЛЮБОГО ПРОЕКТА, НАХОДЯЩЕГОСЯ В СТАДИИ ЭКСПЛУАТАЦИИ (то есть имеющего активных пользователей), наступает момент, когда требуется внести изменения – исправить ошибку либо добавить новую функцию. Возможны два пути: либо вы не пожалеете времени и сделаете все «как положено», либо начнете «срезать углы», чтобы решить задачу побыстрее.

В общем случае представители бизнеса (отдел продаж/маркетинга и клиенты) хотят, чтобы изменения были внесены как можно быстрее, а разработчики и тестировщики больше заинтересованы в том, чтобы потратить нужное количество времени на проектирование, реализацию и тестирование изменений, прежде чем продукт будет отправлен клиентам.

Вам как архитектору проекта предстоит решить, какой путь более разумен, а затем убедить лиц, принимающих решения, последовать вашему совету; как в большинстве архитектурных вопросов, здесь необходим разумный компромисс. Если вы считаете, что система достаточно стабильна, возможно, стоит поработать «на скорую руку» и внести изменения как можно быстрее. В принципе, это нормально, но учтите, что при этом проект берет «технический кредит», который позже придется выплачивать. В нашем случае «выплата» означает, что вам все равно придется вернуться и внести изменения так, как вы это сделали бы, располагая необходимым временем и ресурсами.

Тогда почему мы беспокоимся о том, когда следует внести добросовестные изменения – сейчас или потом? Потому что изменения «на скорую руку» сопряжены со скрытыми расходами. В финансовой сфере такие скрытые расходы на кредит называются процентами; любой владелец кредитной карты знает, как дорого может обходиться простая выплата процентов по кредиту.

Для технических долгов «проценты» принимают обличие нестабильности системы и повышенных затрат на сопровождение, обусловленных топорным внесением изменений и отсутствием должного проектирования, документации и/или тестов. Причем, как и в финансовом случае, проценты на техническую задолженность выплачиваются регулярно до тех пор, пока кредит не будет погашен.

Теперь, когда вы лучше представляете себе истинную стоимость технических кредитов, возможно, вы решите, что цена слишком высока. Но когда приходится выбирать между максимально быстрым исправлением ошибки и серьезными финансовыми потерями, обычно имеет смысл выбрать первое. Итак, примите на себя удар и исправьте продукт как можно скорее, но только не останавливайтесь на этом.

Как только исправление попадет в работающую систему, добейтесь того, чтобы разработчики вернулись и внесли полноценные исправления, которые можно включить в следующий запланированный выпуск. Такой подход можно сравнить с выплатой по кредитной карте и погашением долга в конце месяца, чтобы не пришлось платить проценты. Это позволяет вносить быстрые изменения, требуемые бизнесом, но одновременно защищает ваш проект от попадания в «долговую яму».

Берк Хафнагель (Burk Hufnagel) создает пользовательские приложения с 1978 года. В настоящее время работает ведущим архитектором ПО в LexisNexis.

Не спешите решать задачи

Эбен Хьюит

Практически все архитекторы когда-то были разработчиками. Разработчикам платят за решение задач из области программирования, менее масштабных по сравнению с архитектурными задачами. Как правило, это мелкие каверзные алгоритмические задачи. Они часто представлены в книгах и учебных курсах так, словно существуют в отрыве от реальности, а их каверзность выглядит весьма вызывающе и соблазнительно. Со временем мы начинаем воспринимать такие задачи как данность – мы не спрашиваем, насколько осмысленна задача, интересна ли она, полезна, этична и так далее. Нам не платят за анализ роли задачи в более широком контексте. Мы приучены концентрироваться исключительно на самом решении; дело усугубляется тем, что решать сложные задачи действительно трудно. Мы привыкли брать быка за рога на собеседованиях, где перед нами по сути вываливают груду цветных леденцов, требуя рассортировать их в соответствии с некоторым набором ограничений. Нас приучают не сомневаться в этих ограничениях: они – **учебный инструмент, при помощи которого мы должны самостоятельно** открыть то, что уже известно учителю или экзаменатору.

Архитекторы и разработчики привыкают немедленно переключаться в режим решения задачи. Но в некоторых ситуациях лучшее решение – это не браться за решение совсем. Многие программные задачи вообще не нуждаются в решении. Мы воспринимаем их как проблемы лишь потому, что видим симптомы, но не сами явления.

Возьмем в качестве примера автоматическое управление памятью. На платформах с автоматическим управлением памятью разработчики не занимаются решением задач по управлению памятью; большинство из них не сможет это сделать, даже если потребуется, – само решение подразумевает, что они практически полностью избавлены от подобных проблем.

Или рассмотрим сложную систему сборки с множеством взаимосвязанных сценариев, требующих соблюдения уймы стандартов и соглашений. Да, вы в состоянии решить эту задачу – и было бы так здорово применить всю мощь своих навыков написания сценариев и весь свой опыт, чтобы ощутить законную гордость, когда система заработает!.. Это произведет впечатление на ваших коллег. А если вы не займетесь решением задачи, никто не впечатлится. Однако если вы сумеете сделать шаг назад и понять, что в данном случае имеете дело не с задачей организации сборки, а с задачей автоматизации и обеспечения переносимости, возможно, вы найдете инструмент, который избавит вас от необходимости ее решать.

Из-за того что архитекторы склонны немедленно входить в режим решения задач, они часто забывают (а то и вовсе не умеют) подвергать анализу саму задачу. Архитектор должен научиться, подобно линзе телеобъектива, увеличивать и уменьшать масштаб, чтобы убедиться в том, что задача действительно очерчена правильно и он не просто бездумно принимает то, что дали сверху. Не превращайтесь в простую машинку, которая глотает требования и выдает умные решения, словно автомат по продаже леденцов.

Вместо того чтобы немедленно приступать к решению задачи в том виде, в котором вы ее получили, подумайте, нельзя ли изменить саму задачу. Спросите себя: как бы выглядела архитектура, если бы этой задачи просто не было? Такой подход может дать в итоге более элегантное и сбалансированное решение. Бизнес-задачу все равно придется решать, но, возможно, не так, как казалось изначально.

Преодолейте свое пристрастие к задачам. Мы любим иметь с ними дело; мы видим себя в роли тайного агента, который только что получил самоуничтожающийся коричневый конверт с описанием сверхсекретного задания. Прежде чем обдумывать свой ответ на задачу, подумайте, как бы выглядел мир, если бы этой задачи просто не было.

Биография автора приведена на стр. 175.

Стройте zuhanden-системы

Кейт Брайтуэйт

Мы создаем инструменты. Создаваемые нами системы служат единственной цели (не считая того, что нам за них платят) – помогать кому-то (обычно кому-то другому) что-либо делать.

Мартин Хайдеггер (Martin Heidegger), известный немецкий философ XX века, исследовал, как люди воспринимают инструменты (и вообще «оборудование»). Человек использует инструмент для достижения определенной цели, причем сам инструмент является всего лишь вспомогательным средством.

Для успешного применения инструмент должен быть *zuhanden* («подручным», то есть «ложащимся в руку»). Иначе говоря, такой инструмент воспринимается непосредственно, он используется инстинктивно, без размышлений и полемики. Мы просто берем инструмент и используем его для достижения нашей цели. В ходе такого использования инструмент исчезает, он фактически становится продолжением нашего тела и не воспринимается как нечто отдельное. Одним из признаков *zuhanden*-инструмента является то, что он становится как бы невидимым, неосязаемым, незаметным.

Допустим, вы бьете молотком по гвоздю или пишете ручкой. Представьте себе всю непосредственность этих действий. Подумайте о том, каким образом инструмент становится естественным продолжением вашего тела.

Возможна и другая ситуация (обычно она свидетельствует о возникших проблемах): пользователь воспринимает инструмент как *vorhanden* («наличествующий», то есть «находящийся в руке»). Инструмент отделяется от цели; он находится перед вами, требуя вашего внимания. Он становится объектом отдельного исследования. Пользователь уже не может двигаться к цели; он должен разобраться с инструментом, прежде чем тот сможет хоть как-то приблизить его к ней. Нам как техническим специалистам

свойственно воспринимать создаваемые нами для пользователей системы как *vorhanden*-инструменты – и в ходе работы над ними, и позднее, когда мы получаем сообщения о дефектах. Инструмент вполне закономерно является для нас объектом рассмотрения, предметом размышлений, исследовательской задачей. Это нечто требующее изучения.

Однако (и это очень важно!) пользователи смогут добиться успеха, применяя наш продукт, только в том случае, если для них это *zuhanden*-инструмент. Спроектированы ли ваши системы так, чтобы становиться «невидимыми» при использовании? Насколько естественно «ложится в руку» пользовательский интерфейс? Или же ваша система постоянно требует внимания, отвлекая пользователя от его цели?

Биография автора приведена на стр. 35.

Найдите и удерживайте энтузиастов

Чед Лавинь

Формирование команды незаурядных разработчиков – одна из самых важных задач, решение которых обеспечивает успех программного проекта. О сохранении существующей команды говорят значительно реже, но эта задача не менее важна. Итак, вы должны тщательно отбирать команду разработчиков и старательно оберегать ее в ходе дальнейшей деятельности.

Вероятно, большинство читателей согласится с тем, что поиск первоклассных разработчиков требует проведения основательного технического собеседования. Но что следует понимать под «основательным»? Это вовсе не означает, что кандидат должен ответить на трудные вопросы о малоизвестных технических нюансах. Разумеется, проверка конкретных технических навыков является частью процесса, но превращение собеседования в сертификационный экзамен не гарантирует успеха. Вам нужны разработчики-энтузиасты, обладающие навыком поиска решений. Инструменты, которые вы используете сейчас, наверняка изменятся; вам нужны люди, способные успешно пойти на штурм задачи независимо от доступных технологий. Даже если человек может наизусть перечислить все методы API, это практически ничего не скажет вам о его склонностях и о его стремлении решать задачи.

С другой стороны, если вы попросите человека объяснить свой подход к диагностике проблем быстродействия, вы получите ясное представление о его методах решения задач. Хотите узнать, умеет ли разработчик извлекать опыт из полученных уроков? Спросите, что он изменил бы в своем последнем проекте, если бы получил возможность начать его заново. Хорошие разработчики относятся к своей работе с энтузиазмом. Этот энтузиазм непременно проявится, когда они будут рассказывать о своих прошлых проектах, – и вы получите такую информацию, которую невозможно извлечь из правильных ответов на технические вопросы.

Потратив немало сил на формирование сильной команды, сделайте все, что в ваших силах, для ее сохранения. Возможно, некоторые удерживающие факторы (например, оплата) вам неподвластны, но позаботьтесь по крайней мере о тех мелочах, которые помогают сохранить здоровую рабочую среду. Хороших разработчиков часто стимулирует признание их заслуг. Используйте это обстоятельство и отмечайте выдающиеся достижения. Найти хорошего разработчика трудно; показать человеку, что его ценят, легко. Не упускайте простые возможности поднять дух и повысить производительность команды.

Будьте осторожны с порицанием. Избыток негатива подавляет творческое мышление, снижает производительность и, что еще хуже, разобщает команду. Хорошие разработчики умны и знают, что не могут все время ошибаться. Постоянно выискивая в их работе мелкие огрехи, вы потеряете их уважение. Ограничивайтесь конструктивной критикой и не требуйте, чтобы каждое решение выглядело так, будто вышло из ваших рук.

Важность правильного подбора команды разработчиков трудно переоценить. В любой команде есть «тяжеловесы», которым поручаются самые трудные задачи, но, когда дело доходит до оценок, все участники рассматриваются на равных. Позаботьтесь о сплоченности стартового состава, а когда вы успешно сформируете команду победителей, не пожалейте усилий на то, чтобы сохранить ее.

Биография автора приведена на стр. 125.

Программы на самом деле не существуют

Чед Лавинь

Разработку программного обеспечения часто сравнивают с такими устоявшимися дисциплинами, как гражданское строительство. Однако у подобных аналогий имеется серьезный недостаток: в отличие от материальных продуктов, создаваемых в ходе строительства, программы в реальности не существуют – по крайней мере, в традиционном смысле слова. Обитатели мира нулей и единиц не связаны физическими законами, которые накладывают ограничения на классические инженерные парадигмы. Хотя в фазе проектирования ПО применение инженерных принципов дает достаточно хороший результат, предположение о том, что вы сможете воплотить созданный дизайн таким же образом, как это делается в традиционных инженерных подходах, выглядит малореалистичным.

Как бизнес, так и программное обеспечение – живые, динамичные сущности. Бизнес-требования быстро меняются под влиянием таких факторов, как появление новых деловых партнеров и маркетинговых стратегий. По этой причине очень сложно использовать в программном проекте те же подходы, что и в традиционном конструировании (например, строительстве мостов). Вряд ли вас попросят изменить местоположение моста в разгар строительства. В то же время с появлением нового делового партнера вполне может оказаться, что в приложение придется включить поддержку управления контентом на уровне организации. Мы часто говорим, что программные архитектурные решения трудно изменять, но все же это намного проще, чем изменять объекты, воплощенные в камне (как в буквальном, так и в переносном смысле).

Если вы изначально знаете, что создаваемые вами продукты пластичны, а предъявляемые к ним требования могут измениться, вы находитесь в совершенно ином положении, чем инженер, создающий статичный объект.

В инженерных проектах физического характера гораздо проще реализовать принцип «сначала план работ, потом работы по плану». К построению программных продуктов обычно приходится подходить по принципу «сначала план работ, потом подгонки плана».

Эти различия не обязательно ставят нас в невыгодное положение – иногда из них можно извлечь пользу. Например, вы не связаны требованием строить компоненты программных систем в определенном порядке, а значит, можете начать работу с самых рискованных вопросов. Это радикально отличает программный проект от строительства моста, где порядок выполнения задач обусловлен множеством жестких физических ограничений.

Тем не менее гибкость процесса разработки ПО сопровождается также рядом специфических проблем, источником которых часто являются они же сами. Мы, архитекторы, очень хорошо сознаем изменчивую природу своей профессии и любим решать задачи. Ситуацию усугубляет то, что владельцы бизнеса тоже что-то слышали об этом и без особых сомнений инициируют большие изменения. Не торопитесь соглашаться с крупными архитектурными изменениями только потому, что вас подталкивает к этому ваша природа «решателя задач». Подобные решения могут разрушить в целом жизнеспособный проект.

Помните, что формулировка требований – не технический чертеж, а программы не существуют в реальности. Создаваемые нами виртуальные объекты изменять проще, чем их физические аналоги, и это хорошо, потому что такая необходимость возникает довольно часто. Совершенно нормально планировать работу так, будто вы строите объект недвижимости, – просто не удивляйтесь, когда кто-либо потребует его передвинуть.

Биография автора приведена на стр. 125.

Освойте новый язык

Беркхардт Хафнагель

Чтобы добиться успеха в роли архитектора, вы должны говорить понятно для людей, не владеющих вашим родным языком. Нет, я не предлагаю изучать эсперанто или клингонский язык, но вы, по крайней мере, должны говорить на простейших диалектах делового языка и языка тестирования. А если вы не владеете свободно программистским языком, его освоение должно стать для вас делом первостепенной важности.

Если вы не видите особого смысла в изучении других языков, взгляните на такой сценарий развития событий: у владельцев бизнеса есть желание внести изменения в существующую систему, и они организуют встречу с архитектором и программистами для обсуждения этих изменений. К сожалению, никто в технической группе не говорит на языке бизнесменов, а никто из бизнесменов не владеет языком программистов. Скорее всего, встреча будет проходить примерно так:

- Бизнесмен около минуты говорит о необходимости относительно простого усовершенствования существующего продукта. Он объясняет, каким образом эти изменения позволят отделу продаж увеличить долю рынка и повысить популярность продукта.
- Пока он говорит, архитектор начинает рисовать в блокноте какие-то мистические символы и вступает в тихий спор с одним из программистов на таинственном, понятном только им языке.
- Наконец бизнесмен завершает свою речь и выжидательно смотрит на архитектора.
- Архитектор заканчивает перешептываться, выходит к доске и принимается рисовать сложные диаграммы с несколькими представлениями существующей системы, попутно объясняя (в сложных технических

терминах), почему требуемое усовершенствование практически невозможно реализовать без радикальных изменений в системе и почему оно требует полного перепроектирования и переписывания всей системы.

- Бизнесмены (которые мало что поняли в диаграммах и еще меньше в объяснениях) явно озадачены. Им трудно поверить, что такой пустяк требует столь серьезных изменений. Они пытаются понять, говорит ли архитектор серьезно или просто придумывает отговорки, чтобы вообще избежать изменений.
- Тем временем архитектор и программисты в свою очередь поражаются, почему до бизнесменов не доходит, что их «мелкая задачка» требует принципиального изменения базовой функциональности системы.

В этом и состоит суть проблемы. Ни одна из сторон не понимает, что думает другая и что означает добрая половина используемых ею слов. Все это порождает непонимание и недоверие. Срабатывает простейший психологический принцип: люди более комфортно чувствуют себя в общении с теми, кто похож на них, а не с теми, кто от них отличается.

Представьте себе, как изменится описанная ситуация, если архитектор сможет объяснить бизнесменам возникающие проблемы на понятном им деловом языке, а потом переведет суть проблем бизнеса на язык, понятный программистам. Вместо удивления и взаимного недоверия они придут к согласию и компромиссу.

Я не утверждаю, что изучение нескольких языков решит все ваши проблемы. И все же оно поможет предотвратить недопонимание, порождающее проблемы.

Тем читателям, кто находит эту точку зрения разумной, я желаю успеха на их пути. Или, как говорят клингоны, – *Qapla!*[1]

Биография автора приведена на стр. 189.

[1] Клингоны – вымышленная цивилизация из сериала «Star Trek» («Звездный путь»). Слово «qapla» в клингонском языке является пожеланием успеха. – *Примеч. ред.*

Не создавайте решения «на перспективу»

Ричард Монсон-Хейфел

Сегодняшнее решение становится завтрашней проблемой

Никто не может предсказать будущее. Если вы согласны с этой всеобщей истиной, возникает вопрос: что следует считать будущим? Десятилетие? Два года? Двадцать минут? Если будущее невозможно предсказать, значит, вы не можете прогнозировать вообще ничего. Текущий момент и то, что ему предшествовало, – вот и все, что вы знаете, пока не наступит следующий момент. Собственно, из-за этого происходят автомобильные аварии: если бы вы знали, что во вторник попадете в катастрофу, то, скорее всего, остались бы дома.

И все же некоторые архитекторы проектируют системы «на перспективу», пытаясь, так сказать, застраховать их от будущего. Однако это попросту невозможно. Какое бы архитектурное решение вы ни приняли сейчас, рано или поздно оно устареет. Новомодный язык программирования, который вы примените, завтра станет таким же ископаемым, как COBOL. Сегодняшняя распределенная инфраструктура завтра будет выглядеть такой же несовершенной, как DCOM сегодня. Короче говоря, сегодняшнее решение неизбежно превратится в завтрашнюю проблему.

Если вы примете тот факт, что решения, принятые сегодня, заведомо станут ошибочными в будущем, это избавит вас от напрасных попыток обеспечить своим архитектурам долгую жизнь. Раз любое сегодняшнее решение все равно окажется плохим завтра, можно не беспокоиться о том, что его ждет впереди, – выбирайте то решение, которое лучше всего соответствует вашим сегодняшним потребностям.

Современные архитекторы часто сталкиваются с проблемой «аналитического паралича». В немалой степени она обусловлена желанием угадать лучшую технологию на будущее. Однако даже выбор технологии, правильной на текущий момент, – уже достаточно трудная задача, а потому попытки выбрать то, что будет актуально в будущем, обречены на неудачу. Подумайте, что нужно вашему бизнесу прямо сейчас. Изучите текущие предложения технологического рынка. Выберите то решение, которое лучше всего соответствует вашим сегодняшним потребностям, потому что любой другой выбор будет неверным не только завтра, но и сегодня.

Ричард Монсон-Хейфел (Richard Monson-Haefel) – независимый разработчик ПО, соавтор всех пяти изданий «Enterprise JavaBeans» и обоих изданий «Java Message Service» (все книги опубликованы издательством O'Reilly). Занимается проектированием и разработкой ***multitouch-интерфейсов, является*** *ведущим специалистом в области корпоративной обработки данных.*

Проблема пользовательского признания

Норман Карновейл

Люди не всегда рады появлению новых систем или крупным обновлениям. Это может поставить под угрозу успешное завершение проекта.

Решение о реализации новой системы нередко принимается в штыки – особенно на ранней стадии. Это вполне естественно, и причины хорошо известны. Тем не менее начальная реакция на новую систему создает меньше проблем, чем незатухающая отрицательная реакция.

Вы как архитектор должны знать о проблеме с признанием системы пользователями, понимать текущий уровень ее опасности и принимать меры к ее устранению. Для этого вам необходимо понимать суть проблемы и причины, ее порождающие. Вот наиболее распространенные причины:

- Люди могут иметь свои соображения о необходимости внедрения новой системы (и сопутствующего вывода из эксплуатации старой). В частности, они могут опасаться утратить ценную функциональность или потерять свое влияние в компании из-за перераспределения ролей.
- Люди боятся новых (непроверенных) технологий.
- Люди стараются избежать дополнительных затрат.
- Люди просто не любят изменения.

Каждая из причин требует своего подхода; с одними справиться можно, с другими нет. Вы должны понять различия и быстро устранить те проблемы, повлиять на которые в ваших силах. Как можно раньше начните обсуждать с конечными пользователями новую систему, ее реальные и кажущиеся преимущества и недостатки. Самая эффективная долгосрочная мера – использовать архитектуру системы, чтобы удовлетворить интерес пользователей.

К числу других эффективных мер относится обучение, запланированные демонстрации системы (на ранних стадиях жизненного цикла проекта) и распространение информации о том, какую пользу принесет новая система.

Наличие у проекта сторонников также помогает решить проблему признания. В идеале это должен быть человек, представляющий группу пользователей или заинтересованных лиц. Иногда самого этого человека поначалу приходится убеждать в преимуществах системы. Если подходящего кандидата нет, начинайте поиски с самого начала работы над проектом, а когда найдете, окажите ему всю возможную помощь и поддержку.

Биография автора приведена на стр. 57.

О важности консоме

Эбен Хьюит

Консоме – осветленный бульон, который обычно готовится из говядины или телятины и подается как деликатесный суп. Правильно приготовленный консоме идеально прозрачен. Его приготовление считается сложным и трудоемким делом, потому что существует только один способ удаления жира и других примесей и достижения абсолютной прозрачности, необходимой для этого блюда: многократно, раз за разом тщательно процеживать бульон. Усердная очистка путем многократной фильтрации отвара создает чрезвычайно богатый вкус. Пробуя консоме, вы как будто ощущаете саму сущность бульона. Собственно, для этого и делается такое блюдо.

В американских кулинарных школах среди начинающих шеф-поваров, готовящих консоме, проводится простое испытание: преподаватель бросает в янтарный бульон десятицентовую монетку. Если можно прочитать дату чеканки на монетке, лежащей на дне миски, испытание пройдено. Если нет – попытка считается неудачной.

Архитектор программного обеспечения должен постоянно избавлять мысли от примесей, многократно фильтровать свои идеи, пока не будет выявлена сущность каждого требования к системе. Словно Гамлет, держащий череп Йорика, мы спрашиваем себя: что представляет собой данная возможность? Каковы ее свойства? Ее связи? Мы проясняем свои концепции, чтобы отношения в архитектуре поддавались проверке и были внутренне согласованными.

Многие пропущенные требования и ошибки в программных продуктах возникают из-за размытости, неоднозначности языка. Многократно задавайте клиентам, разработчикам, аналитикам и пользователям одни и те же вопросы, пока они не начнут зевать от скуки. Теперь замаскируйте свой вопрос, чтобы придать ему новый облик. Словно прокурор, выискивающий изъян

в алиби, подмечайте все новое, любые различия и противоречия. Фильтруйте снова и снова.

Постарайтесь понять, какие составляющие можно исключить из концепций, представленных в архитектуре, для обнажения их сущности. С хирургической точностью формулируйте требования, избавляясь от всякой неоднозначности, общих слов, необоснованных предположений или многословия. Это сделает ваши концепции более мощными и устойчивыми. Сокращайте снова и снова.

Проверяйте утверждения, спрашивая: «Останется ли это утверждение справедливым, если прибавить к нему “везде, всегда и при любых обстоятельствах”?» Люди стараются воздерживаться от подобных безапелляционных формулировок и поэтому начинают более тщательно подбирать слова. Просеивайте представления концепций через лингвистическое сито, чтобы очистить их от примесей. Повторяйте до тех пор, пока у вас не останется только исчерпывающий список простых, поддающихся проверке утверждений, описывающий истинную сущность системы.

В какой момент работу над архитектурой можно считать завершенной? Когда она станет абсолютно прозрачной и вы сможете прочитать дату на монетке.

Биография автора приведена на стр. 175.

Для пользователя интерфейс – это и есть система

Винаяк Хеджд

Слишком много хороших продуктов скрывается за плохими пользовательскими интерфейсами. Конечный пользователь работает с системой через интерфейс пользователя. Если опыт взаимодействия пользователя с вашим продуктом окажется негативным, пострадает и впечатление пользователя от всего продукта, каким бы технологически совершенным и новаторским он ни был.

Пользовательский интерфейс – важный архитектурный компонент, которым часто пренебрегают. Архитектор должен прибегнуть к услугам специалистов – проектировщиков опыта взаимодействия и экспертов по юзабилити. Специалисты по взаимодействию с пользователями совместно с архитектором управляют как архитектурой интерфейса, так и его связями с внутренними механизмами. Привлечение специалистов по интерфейсам на ранней стадии и их участие во всех последующих фазах разработки продукта гарантирует, что итоговый продукт будет отшлифован до блеска, а пользовательский интерфейс будет безукоризненно интегрирован с продуктом. Архитектор должен также проследить за взаимодействием реальных конечных пользователей с продуктом в ходе бета-тестирования и учесть полученные данные в окончательной версии системы.

Интерфейс продукта часто изменяется со временем по мере изменений в технологиях и добавления новых функций. Архитектор должен позаботиться о том, чтобы изменения в интерфейсе пользователя и в архитектуре отражали ожидания пользователей.

Взаимодействия с пользователями должны быть одной из приоритетных целей архитектуры продукта в целом. По сути дела, взаимодействия с пользователем должны стать такой же неотъемлемой частью процесса принятия

решений при проработке архитектурных компромиссов и внутренней документации продукта, как надежность и производительность. Изменения в логике взаимодействия с пользователем должны фиксироваться во времени, как и исходный код. Это особенно справедливо для тех продуктов, в которых пользовательский интерфейс пишется не на том языке программирования, на котором создается вся остальная система.

Архитектор отвечает за то, чтобы основные пути взаимодействия пользователя с продуктом стали не только простыми и удобными, но и приятными. Хороший интерфейс создает у пользователя позитивное впечатление от работы с продуктом, что делает его работу более продуктивной. Если же ваш продукт помогает пользователям повысить продуктивность, это наверняка отразится на экономических показателях вашей организации.

Винаяк Хеджд (Vinayak Hegde) – технофил, интересующийся компьютерами, фотографией и предпринимательством. В настоящее время работает архитектором в компании Akamai Technologies.

Лучшие программы не строят – их выращивают

Билл де Ора

В ОБЯЗАННОСТИ АРХИТЕКТОРА входит формирование исходной структуры и облика программной системы, которая будет расти и изменяться со временем, будет перерабатываться и взаимодействовать с другими системами – причем эти изменения почти всегда происходят не так, как прогнозирует сам архитектор или другие участники проекта. Хотя мы называемся *архитекторами* и многие метафоры в нашей работе позаимствованы из строительства и инженерного искусства, по-настоящему выдающиеся программы не строят – их выращивают.

Самым верным предвестником провала программного продукта является его размер. Если задуматься, нет никакого смысла начинать с проектирования крупномасштабной системы. Тем не менее в какой-то момент у каждого из нас возникает соблазн действовать именно так. Помимо неизбежной побочной сложности и инерции, проектирование системы крупного размера с самого начала увеличивает масштаб проекта, что повышает вероятность его провала, затрудняет тестирование, делает проект менее стабильным, приводит к появлению лишних и бесполезных частей, обычно повышает затраты на реализацию и часто вводит в него негативную политическую составляющую.

Поэтому сопротивляйтесь искушению спроектировать крупную завершенную систему, которая «с запасом» удовлетворяет имеющимся ожиданиям и поставленным требованиям, как бы соблазнительно ни выглядел такой подход. Крупномасштабным должно быть ваше видение системы, а не ее дизайн. И вы, и ваша система должны научиться адаптироваться к неизбежному изменению ситуации и требований.

Как этого добиться? Лучший метод снабдить программную систему способностью развиваться и адаптироваться – заставить ее развиваться и адапти-

роваться с самого начала. Вы начинаете с небольшой работоспособной системы, рабочего подмножества предполагаемой архитектуры – простейшей конфигурации, какая только может работать. Такой зародыш системы обладает многими полезными свойствами и может рассказать вам о заложенной архитектуре столько, сколько никогда не расскажет большая система (и тем более подборка документов с описанием архитектуры). Вы с большей вероятностью будете участвовать в его реализации. Миниатюрность упрощает тестирование и снижает уровень связанности. Для создания такого зародыша потребуется группа меньшего размера, что сократит затраты на координацию проекта. За его свойствами будет проще наблюдать. Его будет проще развертывать. По нему вы и ваша группа сможете как можно раньше понять, какие решения работают, а какие нет. Он покажет вам, в каких местах развитие системы затруднено, где она наверняка кристаллизуется и станет хрупкой. Где она может сломаться. И, что самое важное, вы сможете с самого начала предъявить нечто понятное и осязаемое заинтересованным сторонам проекта, помогая им как можно раньше вникнуть в общий дизайн системы.

Спроектируйте настолько маленькую систему, насколько сможете, помогите в ее создании и предоставьте ей возможность развиваться по направлению к основной цели. Хотя на первый взгляд может показаться, что вы теряете контроль над проектом и даже уклоняетесь от выполнения своих прямых обязанностей, в конечном итоге участники проекта поблагодарят вас. Только не путайте эволюционный подход с игнорированием требований, бездумной разбивкой на этапы и созданием системы для мусорной корзины.

Биография автора приведена на стр. 131.

Алфавитный указатель

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Я