ГЛАВА 2

Проектирование успеха

Как проект отстает на год?
Сначала он отстает на один день.
Фред Брукс (Fred Brooks)

Эффективный проект программного обеспечения — это проект, результатом которого является работоспособное решение, удовлетворяющее основные потребности бизнеса. Эффективное программное обеспечение — это программная система, являющаяся результатом эффективного проектирования на основе повторного использования готового кода и (по возможности) инфраструктуры с учетом доступных технологий и передовых достижений.

В настоящее время использование эффективного программного обеспечения является очень важным для бизнеса любого вида и размера, но еще важнее избежать использования неэффективного программного обеспечения. Плохое программное обеспечение может принести организации убытки. Причинами могут быть, например, медленно реагирующая страница, отталкивающая пользователей от вашего веб-сайта; запутанный пользовательский интерфейс, вызывающий задержки и порождающий очереди пользователей ваших услуг, и даже необработанное исключение, запускающее цепную реакцию неконтролируемых операций и приводящее к непредсказуемому результату.

Проекты программного обеспечения редко соответствуют ожиданиям. Мы предполагаем, что все, кто держит в руках эту книгу, хорошо это понимают. Что же может препятствовать достижению успеха в проектировании программного обеспечения? Если бы мы попробовали добраться до основных причин, мешающих проектам программного обеспечения полностью оправдывать надежды, то обнаружили бы явление, которое называется БОЛЬШИМ КОМКОМ ГРЯЗИ (BIG BALL OF MUD — BBM).

Аббревиатура BBM — это элегантный эвфемизм, обозначающий программную катастрофу.

Программная катастрофа — это неправильно и неконтролируемо растущая система, не поддающаяся ремонту. Иногда неудачно спроектированные системы удается исправить с помощью заплаток и оставить в наследство будущим программистам. Разработчики должны всегда стремиться к созданию эффективного программного обеспечения, хотя, как показывает история, иногда все можно исправить с помощью скотча. На сайте Space.com (по адресу http://bit.ly/1fJW6my) описан поучительный пример: Юрий Гагарин — человек, совершивший первый полет в космос в

1961 году — прямо перед стартом получил инструкцию оторвать кусочек пластыря и починить неполадку в оборудовании[1].

Термин БОЛЬШОЙ КОМОК ГРЯЗИ появился много лет назад и обозначает плохо структурированную систему, содержащую множество скрытых зависимостей между ее компонентами, большое количество данных и дубликатов кода и не имеющую явно идентифицированных уровней и понятий, в общем "каша из макаронов". Этот термин был изобретен Брайаном Футом (Brian Foote) и Джозефом Йодером (Joseph Yoder) из Иллинойского университета (University of Illinois) и проанализирован в их статье, размещенной по адресу http://www.laputan.org/pub/foote/mud.pdf.

В своей статье авторы термина BBM не клеймят его позором; они просто рекомендуют архитекторам и разработчикам быть готовым к этой опасности и учат их, как с ней справиться. Иначе говоря, угроза BBM носит постоянный характер и может появиться в любом проекте программного обеспечения, размер которого превышает некоторую критическую массу. Единственный способ избежать программной катастрофы — научиться распознать и устранить BBM.

Причины появления БОЛЬШОГО КОМКА ГРЯЗИ

Существует несколько основных причин появления BBM в проекте программного обеспечения. Главная причина заключается в том, что BBM не появляется внезапно и вначале не имеет большого размера. Вторая причина состоит в том, что ни один отдельно взятый разработчик не может создать BBM с нуля. BBM — всегда результат коллективного творчества.

Для того чтобы докопаться до корней этого явления, мы идентифицировали несколько основных причин, которые могут привести к появлению BBM.

Невозможность удовлетворить все потребности клиента

Архитекторы и разработчики создают программное обеспечение, особенно промышленное, для достижения конкретных целей. Цель программного обеспечения, выраженная в общих словах, — выяснить и решить проблемы заказчика. Существует целая отрасль программной инженерии, посвященная анализу и классификации требований, предъявляемых к программному обеспечению на разных уровнях, — требований бизнеса, требований заинтересованных сторон, функциональных требований, требований к тестированию и т.д.

Все дело в том, как вы переходите от длинного списка требований, выраженного на естественном языке, к конкретным свойствам программы, написанной на языке программирования.

---

[1] Расшифровка стенограммы разговоров Ю.А. Гагарина с центром управления полетами перед стартом приведена на веб-сайте http://www.cosmoworld.ru/spaceencyclopedia/gagarin/index.shtml?doc10.html. — *Примеч. ред.*

В главе 1, “Современные архитекторы и архитектура”, среди основных обязанностей архитектора мы указали признание требований. Требования часто поступают из разных источников и относятся к разным точкам зрения на одну и ту же систему. Поэтому не удивительно, что некоторые требования противоречат друг другу, а их релевантность варьируется в зависимости от конкретной заинтересованной стороны. Анализ требований и выявление тех из них, которые будут воплощены в системе, относятся лишь к первому этапу работы архитектора.

Второй этап начинается с проверки списка выбранных свойств. Предложенный список свойств должен соответствовать полному списку требований заинтересованных сторон. Впрочем, некоторые требования отдельных заинтересованных сторон можно просто отбросить.

Да, это возможно, если вы можете объяснить и обосновать причины, по которым эти требования были проигнорированы.

Для того чтобы спроектировать систему, решающую проблему, вы должны полностью понять проблему и соответствующую предметную область. Сделать это, просто прочитав список требований, практически невозможно. Иногда вы должны сказать “Нет”. Однако обычно вы обязаны выяснить причины появления аргументов за и против нового свойства, чтобы обосновать данный набор требований.

Урок, который все мы извлекли за минувшие годы, — то, что первый запуск программы, которую можно было написать в соответствии с частичным и неполным пониманием потребностей, помогает много больше, чем долгий заблаговременный поиск решения. Таким образом, гибкий подход к разработке программ в этом отношении больше основывается на здравом смысле, чем на методологии.

Признание требований требует общения и коммуникативных способностей. Впрочем, иногда общение просто не помогает. Иногда обе стороны верят в неправильные факты и обе в конечном счете пытаются спасти свою репутацию. Таким образом, разработчикам легко жаловаться на недостаток подробностей, а бизнесмены возражают, что все детали описаны в документах.

В основе коммуникационных проблем лежит тот факт, что бизнесмены и разработчики используют разные словари, а также используют и ожидают разные уровни точности в описании. Кроме того, почти все — кроме разработчиков — склонны полагать, что программировать намного легче, чем на самом деле. Добавление нового требования для бизнесменов или торгового персонала столь же легко, как дописать новую строчку в документе. В действительности необходима определенная адаптация к системе, которая связана с накладными расходами.

Поскольку адаптация программного обеспечения, которая следует из новых или измененных требований, действительно вызывает дополнительные расходы — и никто не хочет их оплачивать,— приходится удалять другие свойства или, что более вероятно, уменьшать объем рефакторинга, тестирования, отладки и документации. Когда появляется такой сценарий (что происходит слишком часто), возникает БОЛЬШОЙ КОМОК ГРЯЗИ.

Важное замечание

В предыдущей главе мы рассмотрели типичный процесс обработки требований. Здесь мы хотим лишь кратко повторить сказанное. По существу, мы группируем предварительные требования по категориям и в качестве отправной точки используем стандарт International Organization for Standards/International Electrotechnical Commission (ISO/IEC). Для этого мы просто записываем требования из каждой категории на отдельных листах рабочей книги Microsoft Office Excel. Затем открываем листы и добавляем или удаляем требования, руководствуясь здравым смыслом. Мы внимательно следим за листами, которые остались пустыми, и пытаемся выяснить, чем это вызвано. В конце концов этот процесс позволяет выявить новую информацию и сделать более ясной существующую на основе известных или полученных данных.

Применение быстрой разработки приложений при увеличении размера системы

В начале проект кажется простым. Заказчик утверждает, что ему не нужна большая и сложная система. Следовательно, можно применить некоторые разновидности быстрой разработки приложений (Rapid Application Development — RAD) и уделять меньше внимания аспектам проектирования, связанным с масштабированием системы.

Если же окажется, что размер системы увеличивается, подход RAD оказывается ограничивающим фактором.

Несмотря на то что подход RAD вполне обоснован для небольших и простых приложений, ориентированных на обработку данных (таких, как приложения CRUD), он оказывается опасным при проектировании крупных систем, учитывающих множество правил, специфичных для предметной области.

Неточность оценок

Бизнесмены всегда хотят точно знать, что они получат, прежде чем начнут тратить свои финансовые ресурсы. Однако они рассуждают с точки зрения высокоуровневых функциональных возможностей и поведения. Как только они сказали, что хотят иметь веб-сайт, доступный только авторизованным пользователям, они полагают, что на этом их функция выполнена. Они не чувствуют потребности уточнить, что пользователи также должны иметь возможность зарегистрироваться в длинном списке социальных сетей. Если этот вопрос будет поднят позже, они поклянутся, что все это было написано в документах.

Со своей стороны, разработчики хотят точно знать, что именно они должны создать, чтобы сделать приемлемые оценки. На этом уровне разработчики исследуют детали основных элементов системы и разделяют функциональные возможности на фрагменты. Проблема состоит в том, что эти оценки не могут быть точными, пока все требования не будут ясно определены и признаны. Если же требования изменяются, то оценка тоже подвергается изменениям.

Возникает царство неуверенности. Типичный сценарий выглядит следующим образом.

- Бизнесмены и команда разработчиков достигают первичного соглашения о функциях системы и календарном плане, и все счастливы. На первом этапе команда разработчиков должна разработать прототип.
- Команда разработчиков создает прототип и организовывает демонстрационную сессию.
- В ходе демонстрационной версии возникают более точные представления о функциях системы. Бизнесмены уточняют свое представление о системе и просят внести некоторые изменения.
- Команда разработчиков охотно включает в проект новые модули, но просит дополнительное время. Следовательно, затраты на разработку системы увеличиваются.
- Однако с точки зрения бизнесменов система осталась прежней, и они не видят причины, почему он должна стоить дороже. Ведь они всего лишь уточнили свои требования!

Когда вы делаете оценку, чрезвычайно важно выяснить источники неопределенности.

Нехватка времени для тестирования

В процессе разработки программного обеспечения тестирование выполняется на разных уровнях.

- *Модульное тестирование*, позволяющее определить, соответствуют ли отдельные компоненты программного обеспечения функциональным требованиям. Модульное тестирование важно для выявления любого регресса функциональных возможностей при рефакторинге кода.
- *Комплексное тестирование*, позволяющее определить, соответствует ли программное обеспечение окружению и инфраструктуре, а также правильно ли взаимодействуют несколько компонентов.
- *Приемочное тестирование*, позволяющее определить, соответствует ли законченная система и все ее части всем требованиям заказчика.

Модульное и комплексное тестирование относятся к сфере компетенции команды разработчиков и должно убедить команду в качестве программного обеспечения. Результаты тестирования сообщат команде, хорошо ли она работает и находится ли она на правильном пути.

Критически важным аспектом модульного и комплексного тестирования является разработка и выполнение тестов.

По отношению к модульному тестированию существует общее убеждение, что тесты надо разрабатывать одновременно с программой и объединять испытания с процессом сборки. Однако модульное тестирование занимает меньше времени, чем комплексное. Настройка комплексного тестирования также представляет собой намного более долгий процесс, который должен выполняться перед каждым запуском программы.

В проекте, предусматривающем объединение компонентов, созданных отдельными разработчиками или командами, существует опасность того, что компоненты поначалу будут плохо совмещаться друг с другом. Поэтому рекомендуется, чтобы затраты на интеграцию были распределены так, чтобы проблемы обнаруживались как можно скорее. Откладывание комплексного тестирования на конец проекта крайне опасно, потому что оно сокращает запас времени, которое вы имеете для внесения исправлений, не прибегая к заплаткам и другим практическим хитростям.

Нечеткая принадлежность проекта

Как показывает пример сайта `healthcare.gov`, в контракте на создание системы может участвовать много поставщиков, и необходимо точно определить право собственности на проект.

Поставщик или лицо, выполняющее проект, несут ответственность за общее качество и, помимо прочего, обязаны гарантировать, что каждый компонент имеет самое высокое качество и соответствует остальным частям системы. Он легко может обеспечить своевременное тестирование, чтобы своевременно обнаружить проблемы интеграции. В то же время он может добиться согласования графиков и потребностей между командами и заинтересованными сторонами так, чтобы каждый поставщик не наносил вреда другому.

Если лидер проекта определен нечетко или, как в случае с сайтом `healthcare.gov`, определен, но проявил необязательность, беспокойство за проект остается доброй волей отдельных поставщиков — за исключением ситуаций, в которых у каждого из них нет причины следить за положением дел за пределами их контракта. Работая под давлением, легко оказаться в ситуации, когда вы сосредоточены лишь на обеспечении работоспособности компонента, не интересуясь аспектами его дизайна и интеграции.

В этом случае через несколько итераций возникнет “макаронный код”.

Игнорирование степени кризиса

Технические трудности являются нормой для проекта программного обеспечения.

Если возникают трудности, не имеет смысла напускать тумана и заверять клиента, что все в порядке. Сокрытие проблем не добавит вам славы, если проект все же будет успешно завершен. Если же проект потерпит крах, то сокрытие трудностей станет источником многих неприятностей.

Как архитектор, вы должны быть таким же открытым и прозрачным, будто разрабатываете программное обеспечение с открытым кодом.

Если вы признаете своего рода кризис, то сообщите другим об этом и скажите им, что вы делаете и планируете сделать. Выработка подробного плана исправлений является самой трудной частью работы. Однако заинтересованным сторонам просто нужно сообщить о том, что работа продолжается, и уверить их, что команды двигаются в правильном направлении. Иногда подробного плана обновлений и отчетов о сделанной работе более чем достаточно, чтобы избежать давления, которое может выйти за допустимые пределы.

Как распознать природу кризиса, связанного с БОЛЬШИМ КОМКОМ ГРЯЗИ и эффективными программами?

Это вопрос здравого смысла. Если вы не скрываете проблемы, то скорее всего они будут решены раньше.

Симптомы большого комка грязи

Должно быть очевидно, что как архитектор, менеджер проектов или и тот и другой вы должны приложить все усилия, чтобы избежать BBM. Однако существуют ли ясные и однозначные признаки, которые указывают, что на вас катится комок грязи?

Попробуем определить несколько признаков, которые должны вызывать тревогу о том, что проект становится на сколькую дорожку.

Жесткий, значит хрупкий

Можете ли вы согнуть кусок дерева? Насколько этот кусок способен сопротивляться вашим усилиям?

Кусок дерева обычно — это жесткая и твердая вещь, сопротивляющаяся деформации. Если применить достаточно большую силу, деформация становится постоянной, и дерево ломается.

Что можно сказать о жестком программном обеспечении?

Жесткое программное обеспечение оказывает сопротивление изменениям. Степень сопротивления измеряется степенью регрессии. Вы можете изменить всего один класс, но это эхом прокатится по всему списку зависимых классов. В результате становится трудно предсказать, как долго будут вноситься изменения, даже самые простые.

Если вы бьете стекло или любой другой хрупкий материал, вы разламываете его на несколько частей. Аналогично, если вы вносите изменение в программное обеспечение и в результате повреждаете его в различных местах, значит, программное обеспечение определенно хрупкое.

Как и в реальной жизни, хрупкость и жесткость в программном обеспечении идут рука об руку. Если изменение класса разрушает (многие) другие классы из-за наличия скрытых зависимостей, то вы наблюдаете симптом плохого проекта и должны исправить его как можно скорее.

Легче сделать заново, чем использовать повторно

Представьте себе, что определенная часть программного обеспечения хорошо работает в одном проекте и вы хотите повторно использовать его в другом проекте, но копирование класса или установка связей в сборке нового проекта просто не дают результата.

Почему это происходит?

Если вы переносите некий код из одного проекта в другой и он оказывается неработоспособным, это обычно объясняется тем, что существуют зависимости или код не был предназначен для совместного использования. Из этих двух причин первая представляет собой самую большую проблему.

Реальной проблемой являются не сами зависимости, а их количество и глубина. Риск заключается в том, что для обеспечения функциональной возможности в другом проекте вам потребуется импортировать намного большее множество функций. В результате вы можете осознать, что код проще написать заново, чем использовать повторно.

Для проекта это плохой признак. Это свидетельствует о недостаточной *переносимости* проекта.

Проще обойти проблему, чем решить ее

Внося изменения в класс, вы часто понимаете, что можете сделать это несколькими способами. Чаще всего существует остроумный, элегантный и логичный способ, который очень трудно и сложно реализовать. В то же время можно быстро и без проблем написать грубый и неэффективный код, решающий ту же проблему.

Что делать?

На самом деле вы можете выбрать любой из этих путей. Все зависит от ситуации и решения менеджера.

В принципе, если обходное решение кажется намного проще и быстрее, чем правильное, значит, в проекте сложилась неидеальная ситуация. Иногда дополнительный труд отпугивает, и вы боитесь выбрать трудный прямой, а не легкий обходной путь. Регрессия, вот что на самом деле должно вас пугать. Если у вас есть хорошие и многочисленные модульные тесты, то, по крайней мере, вы можете быть уверены, что возникшая регрессия будет быстро обнаружена. Но если вы начинаете думать: “Ну да, модульные тесты не дают результата, как бы их обойти?”, то открываете двери для регрессии.

Функциональная возможность, которую проще взломать, чем исправить, тоже не говорит в пользу вашего проекта. Она просто означает, что между классами существует слишком много ненужных зависимостей и что ваши классы не образуют тесно связную массу кода. Этого достаточно, чтобы отпугнуть вас от применения правильного решения, которое требует более глубокого уровня рефакторинга, чем обходное решение.

Это отрицательное свойство проекта часто называют *вязкостью*.

Использование показателей для обнаружения BBM

В литературе часто используется другое название для процесса создания систем с интенсивным использованием программного обеспечения на основе плохого кода — *технический долг* (technical debt — TDBT).

Эта метафора была предложена Уодом Каннингемом (Ward Cunningham) и является особенно удачной, потому что именно плохой код, подобно финансовому долгу, со временем может увеличиваться и накапливать проценты, подлежащие выплате. В долгосрочной перспективе технический долг становится слишком большим и иногда мешает предпринимать правильные действия. Перефразируя известную цитату сэра Уинстона Черчилля, можно сказать: “Группа разработчиков, имеющих технический долг, напоминает человека, стоящего в ведре и пытающегося поднять его рукой”.

Технический долг — это абстрактная идея, и призывать к созданию инструментов для его измерения или даже устранения значит призывать вас творить чудеса. Тем не менее тщательное наблюдение фактов и их анализ позволяют предложить несколько показателей. В результате вы не получите чистое золото, но как минимум повысите свой уровень.

Итак, рассмотрим несколько показателей и инструментов, позволяющих обнаружить приближение BBM.

Статический анализ кода

В принципе члены команды знают, где кроются основные проблемы, но иногда необходимо предъявлять доказательства наличия проблем, связанных с кодом, а не просто перечислять их. Инструменты для статического анализа сканируют и зондируют код и генерируют полезный отчет, который может стать предметом для обсуждения.

В среде Microsoft Visual Studio 2013 существует собственный статический анализатор кода, оценивающий такие показатели, как *покрытие кода* (code coverage) и *цикломатическая сложность* (cyclomatic complexity). Другие аналогичные средства встроены в продукты CodeRush и ReSharper (мы обсудим их позднее в этой главе) или поставляются как отдельные продукты теми же поставщиками.

Представляет интерес инструмент для анализа статического кода NDepend (`http://ndepend.com`), который может создавать граф зависимостей, чтобы вы могли видеть количество и расположение проблемных мест.

Важное замечание

И все же инструменты для статического анализа кода не скажут вам ничего о том, что породило технический долг и что необходимо сделать, чтобы уменьшить его. Они лишь предоставляют информацию к размышлению, но не принимают решение за вас.

Силосная башня знаний

Другим интересным множеством показателей, которые можно использовать для идентификации технического долга, является количество узкоспециализированных навыков, которыми обладает команда. Если в команде есть только один участник, владеющий определенными навыками или знаниями о подсистеме или модуле, то его уход из компании или болезнь станет большой проблемой.

Термин *носители знаний* (knowledge silo) или *информации* (information silo) обычно относится к ситуации, в которой знаниями о коде владеют только некоторые участники команды. Для отдельного разработчика это не составляет трудностей, но для команды это определенно проблема.

Терминология

Мы использовали несколько терминов, смысл которых может зависеть от команды. Это термины *модуль* и *подсистема*. В данном контексте мы не придаем этим терминам особого смысла и рассматриваем их просто как блоки кода. Подсистемой мы обычно называем часть общей системы, а модулем — часть подсистемы.

Механика проектов программного обеспечения

Если вы спросите, почему проекты терпят неудачу, то, вероятно, получите самый распространенный ответ, который ссылается на проблемы, связанные с бизнесом,

такие как пропущенные требования, неправильное управление проектами, неправильные сметы, отсутствие коммуникации и даже несовместимость людей в различных командах. Вы вряд ли услышите ссылку на плохой код.

С учетом сказанного, мы считаем, что нераспознанный BBM может серьезно навредить проекту программного обеспечения, но BBM, оставленный без внимания, может его просто уничтожить.

В конце концов, именно люди и реальное взаимодействие между ними действительно определяют успех или крах проектов программного обеспечения. Однако структура организации и общая культура внутри нее также влияют на конечный результат.

Культура организации

В эпоху Стива Джобса (Steve Jobs) компания Apple выглядела как театр одного актера. Один человек выдвигал идеи и предлагал стратегию, а все команды поддерживали его и реализовывали эту стратегию. Удачная идея и правильная стратегия приносят успех.

Модель компании Microsoft, объявленная Стивом Балмером летом 2013 года (ее критический анализ можно найти на сайте http://www.mondaynote.com/2013/07/14/microsoft-reorg-the-missing-answer), основана на подразделениях, которые часто дополняют друг друга и мало общаются между собой.

Возможно, вы помните, что в 2008 году две группы в компании Microsoft создали две практически эквивалентные платформы — Linq-to-SQL и Entity Framework. Для постороннего наблюдателя было трудно понять, как это могло произойти.

Замечание

В биографии Стива Джобса, написанной Уолтером Айзексоном (Walter Isaacson), можно прочитать интересное мнение о компаниях, состоящих из подразделений. Стив Джобс рассказал, почему именно Apple, а не, скажем, Sony, достигла успеха, выдвинув идею iPod. Для того чтобы претворить эту идею в реальность, компания Apple должна была сначала создать аппаратное и программное обеспечение, а затем вступить в переговоры об авторских правах на музыкальные произведения. В области разработки аппаратного и программного обеспечения такие компании, как Sony, имели примерно такой же опыт, как и компания Apple, и к тому же уже имели доступ к авторским правам на музыкальные произведения и фильмы. Так почему же компания Sony не создала бизнеса, основанного на устройствах iPod? По мнению Джобса, все дело заключалось в культуре принятия решений компании Sony, состоящей из подразделений, каждое из которых имеет свой счет прибылей и убытков.

Вероятно, подразделение, приносящее прибыль благодаря авторским правам на музыкальные произведения, рассматривало подразделение, получающее прибыль за счет MP3-плееров, как угрозу. Эти два подразделения могли бороться друг с другом, вместо того чтобы совместно работать на благо компании (Steve Jobs by Walter Isaacson, Simon & Schuster, 2011)[2].

[2] Русский перевод: Айзексон У. Стив Джобс. — М.: Астрель, 2012. — *Примеч. ред.*

Команды и игроки

Есть старый анекдот об итальянской и немецкой командах, участвующих в гонке в классе восьмерок по академической гребле. Немецкая команда, состоящая из одного рулевого и семи гребцов, легко выиграла гонку. Итальянцы изучили причины поражения и выяснили, что их команда состояли из семи рулевых и одного молодого гребца.

Команда — это группа игроков, опыт и знания которых должны дополнять друг друга в рамках скоординированных усилий.

Замечание

Анекдот об академической гребле продолжается рассказом о том, как итальянская команда решила победить в гонке, но это уже не относится к нашей теме. Для интересующихся все же расскажем, что итальянская команда уволила молодого гребца и наняла четырех рулевых, двух руководителей для рулевых, одного главнокомандующего и одного старого гребца.

В проекте программной системы менеджеры и разработчики стремятся к одной и той же цели. Менеджеры чаще выбирают более агрессивную стратегию, чем разработчики. Разработчики отчитываются перед менеджерами и поэтому иногда склонны соглашаться на любые задачи и сроки. Не следует недооценивать желание большинства разработчиков быть героями. Разработчики стремятся быть супергероями, которые спускаются на землю и спасают мир.

В качестве примера предположим, что менеджер планирует показать потенциальному клиенту демонстрационный пример в понедельник утром. Он хочет удостовериться, что у него есть впечатляющий демонстрационный пример. Менеджер приходит в группу разработчиков и просит, чтобы демонстрационный пример был готов в понедельник. Вероятно, это нанесет повреждения текущему спринту или изменит запланированное действие. Как лучше разрешить этот конфликт интересов между компанией и командой? Перечислим несколько возможных сценариев.

- Команда разработчиков принимает новые сроки сдачи программы, тем самым признавая действия менеджера законными.
- Команда разработчиков не принимает новые сроки сдачи программы и продолжает придерживаться старого календарного плана, что противоречит желаниям менеджера осчастливить клиента.
- Команда разработчиков и менеджер совместно определяют разумную цель и не мешают друг другу достигать своих целей.

Наш опыт показывает, что первый сценарий является самым распространенным, а последний — самым желательным. Говоря о склонности разработчиков соглашаться на любые изменения календарного плана, мы ссылаемся на мнение Дядюшки Боба (Uncle Bob)[3]. Соглашаясь на новый срок сдачи продукта, команда разработчиков не-

[3] Прозвище Роберта Сесила Мартина (Robert Cecil Martin), известного консультанта по разработке программного обеспечения. — *Примеч. ред.*

явно признает, что притормаживала работу в прошлом. Разработчики как бы говорят: “Хорошо, мы можем сделать больше, хотя по некоторым причинам не делали этого; теперь настало время поработать в полную силу”. Однако если команда до сих пор не сдерживала себя на самом деле, то, пытаясь уложиться в новые сроки, она заставит своих членов работать дополнительно, пожертвовав своим личным временем. Это нечестно и по отношению к разработчикам, имеющим право на личную жизнь, и по отношению к менеджерам, которые просто слышат ложь.

Сколько раз мы лично соглашались на более напряженное расписание? Если мы и поступали так, то, как правило, делали это, чтобы избежать открытой конфронтации. Хороший командный игрок должен быть хорошим коммуникатором, всегда быть честным и никогда не лгать, — это единственное правило.

Ключевое слово здесь — *согласование*. Его цель — совместное достижение разумной цели и поиск разумного компромисса между имеющимися потребностями и расписаниями. В предыдущем примере хороший компромисс мог бы заключаться в отмене некоторых функций и создании оперативной сборки, дополненной фиктивным кодом и работающей как демонстрационный пример. Это не отдаляло бы вас от основного пути, не задерживало бы текущий спринт и не требовало, чтобы члены команды работали сверхурочно или сокращали функциональные возможности.

Пожарные Scrum

Каждое непредвиденное событие — потенциальный кризис (особенно в среде гибкого проектирования), и поэтому оно должно немедленно обрабатываться.

В среде Scrum, как правило, одного или нескольких членов называют *пожарными* (firefighters).

Пожарный в среде Scrum обязан выполнять любую дополнительную работу, ответвляющуюся от основной работы, но необходимую для поддержки усилий остальных членов команды. Как и настоящие пожарные, пожарные в среде Scrum иногда простаивают или принимают минимальное участие в текущей итерации проекта.

Поскольку быть пожарным в среде Scrum бывает скучно, эту роль поочередно выполняют все члены команды. Наш опыт показывает, что пожарным следует поручать не более 20% объема всех работ, иначе может снизиться производительность труда и вместо выигрыша вы получите проигрыш.

Лидер или босс

Все мы знаем, что затраты — больной вопрос проектов программного обеспечения. Затраты зависят от количества часов, требуемых для кодирования всех функций, включая тестирование, отладку, документирование и много других действий. Лидер группы разработчиков заботится об этом, обычно отчитываясь перед менеджером проекта.

Иногда между этими фигурами отсутствует взаимопонимание: менеджер думает, что группа разработчиков работает вполсилы, а группа разработчиков думает, что менеджер просто хочет получить больше, заплатив меньше.

Совершенно очевидно, что лидерские качества — это чрезвычайно важное качество.

Довольно часто можно встретить менеджеров, которые занижают свои оценки вдвое, а потом жалуются, что проект опаздывает. Потом они идут к своим боссам, обвиняют группу разработчиков и просят больше средств. Таким образом, они воплощают на практике закон Брукса: “Добавление рабочей силы к опаздывающему проекту программного обеспечения только задерживает его”.

Между лидером и боссом лежит пропасть.

Прежде всего, боссы ожидают, что команда будет служить им, а не они будут служить команде. Боссы сидят на вершине деловой иерархии и приказывают другим решать задачи, которые они сами не готовы или неспособны решить. Лидер, наоборот, сосредоточивается на бизнесе и руководит разработкой “из окопа”. Это различие можно сформулировать так: “Босс создает последователей, а лидер создает других лидеров”.

Помощь команде в разработке хорошего кода

Как известно, слишком многие разработчики считают, что плохой код — это, в конце концов, не так уж и страшно.

Если подсчитать количество проектов, которые, судя по сообщениям, потерпели неудачу из-за проблем, связанных с кодом, то оно окажется не таких уж огромным. Однако вовсе не нужно вызывать истинное бедствие, чтобы потерять большую сумму денег на проекте программного обеспечения.

Как архитектор может помочь команде разработать более качественный код?

Плохой код дороже, чем плохой

Мы уверены, что процесс разработки плохого кода дороже, чем хорошего. По крайней мере, мы полагаем, что это дороже по отношению к проектам со значительной продолжительностью жизни и влиянием на бизнес.

Проще говоря, плохой код уничтожает проект, когда затраты на работу с этим кодом (т.е. создание, тестирование и эксплуатацию) превышают расходы, которые может понести бизнес. К тому же ни один проект не потерпел бы неудачу из-за проблем, связанных с кодом, если бы компаниям удалось удержать его стоимость на достаточно низком уровне.

Это — больной вопрос.

Как определить аспекты, влияющие на окончательный счет за разработку кода? Какие действия подразумевает “разработка кода”: кодирование, сборку, отладку? Следует ли рассматривать тестирование как дополнительную, необязательную возможность? А что насчет документации? А исправление ошибок?

Иногда менеджеры слишком упрощают проблему и сводят ее к сокращению затрат на разработку, нанимая дешевых разработчиков или отказываясь от тестирования и документации.

К сожалению, эти менеджеры не понимают, что они лишь снижают затраты на производство программ, которые могут просто оказаться неработоспособными. Производство работоспособной программы — лишь один из аспектов проблемы. В настоящее время требования часто изменяются, сложность возрастает и, что хуже всего, проявляется только в процессе роботы. В рамках этого сценария производство

кода — всего лишь одна из статей сметы. Самая большая статья расходов — эксплуатация и развитие кода.

Каждый хороший архитектор знает, что решить проблему эксплуатации кода можно только при условии, что код написан правильно, с учетом принципов программного обеспечения и языковых особенностей, на основе хороших шаблонов и методов тестирования. Таким образом, кодирование дороже, чем простое программирование работоспособной программы, но намного дешевле эксплуатации и развития программы, от которой требуется лишь ее работоспособность.

Использование вспомогательных инструментов

Мы думаем, что успех проекта зависит от двух факторов: признания менеджерами роли лидерства и понимания разработчиками важности качества программирования.

У разработчиков, как правило, нет времени, чтобы сначала написать всего лишь работоспособную программу, а потом исправить и очистить ее. Каждый разработчик может поклясться, что второй этап никогда на наступит, а если он наступает, то выполняется не так глубоко и осмысленно, как это должно быть.

Первую помощь при программировании хороших программ оказывают вспомогательные инструменты. Как правило, эти инструменты интегрированы в среду разработки, упрощают решение общих задач развития, ускоряют процесс кодирования и предоставляют программистам шанс написать более качественную программу. В любом случае за счет ускорения программирования у разработчика останется время для второго этапа.

Такие службы, как автодополнение, подсказки по идиоматическому дизайну (т.е. помощь в разработке программ в соответствии с языком или идиомой, предложенной платформой), проверки кода, стандартные фрагменты кода, связанные с комбинациями клавиш, а также стандартные шаблоны, ускоряют разработку и гарантируют логическую согласованность и качество программы.

Вспомогательные инструменты облегчают программирование и значительно улучшают качество кода, который вы создаете с помощью всего лишь нескольких дополнительных щелчков мышью. Эти инструменты могут выявить дублированный и неиспользованный код, выполнить рефакторинг, упростить навигацию и контроль, а также установить шаблоны.

Например, все разработчики соглашаются в принципе, что хорошо продуманное соглашение об именах имеет огромное влияние на читабельность и качество кода (см. главу 4). Однако, понимая, что вы должны изменить пространство имен или имя метода, вы вынуждены делать это во всей кодовой базе. Это трудная работа, которую за короткое время могут выполнить за вас вспомогательные инструменты.

ReSharper — самый популярный доступный вспомогательный инструмент для кодирования. Дополнительную информацию о нем можно получить на веб-сайте `http://www.jetbrains.com/resharper`. Другие аналогичные инструменты — CodeRush от компании DevExpress (`http://www.devexpress.com/Products/CodeRush`) и JustCode от компании Telerik (`http://www.telerik.com/products/justcode.aspx`).

Однако помните, что вспомогательные инструменты для кодирования не волшебные, и все, что они делают, — это помогают вам написать более качественный код по более низкой цене и с меньшим усилием. Все остальное — ваше дело, ведь именно вы управляете инструментами в процессе рефакторинга и редактирования кода.

Как сказать людям, что их код плохой

Допустим, вы заметили, что некоторые члены команды написали плохой код. Как им сказать об этом?

Здесь есть психологические аспекты. Вы не хотите быть резким и причинять кому-либо боль; в то же время вам не нравится, что работа других людей может принести вам неприятности. Главное — общение, не так ли? Следовательно, вы должны найти лучший способ сказать людям, что их код не соответствует критериям качества.

В целом мы полагаем, что лучше сосредоточить внимание на программе, тонко спрашивая, почему это было написано так, а не иначе. Таким образом, вы сможете больше узнать о причинах низкого качества, среди которых могут быть дезинформация, безответственное отношение к работе, ограниченные навыки или, возможно, требования, о которых вы просто не знали.

Никогда не судите программистов, не имея явных доказательств. Таким образом, вы просто будете выглядеть любознательным человеком, интересующимся проблемами кодирования и готовым обсудить альтернативы.

Повышайте уровень программистов

Сформулируем золотое правило для формирования команды, создающей качественный код.

> *Обсуждайте программу, а не программиста. Но при этом старайтесь улучшить код, повышая уровень программиста.*

В системе всегда может обнаружиться фрагмент, который необходимо исправить. Однако, когда это происходит, вы не должны обвинять программиста; вместо этого следует помочь ему повысить свой уровень. В результате вы повысите уровень и мотивацию разработчика. Вы сможете вызвать у разработчика ощущения спасателя и стремление делать свою работу как можно лучше.

Для того чтобы усовершенствоваться в некоторых областях, всем нужны обучение и практика. Самый эффективный способ сделать это — объединить обучение и практику в рамках гибкого подхода. Вместо этого мы слишком часто видим компании, которые покупают учебные пакеты, обучают людей несколько дней, а затем ожидают, что в следующий понедельник люди уже будут работать в полную силу. Это по меньшей мере неэффективно.

Напомним выражение, которое было довольно популярно несколько лет назад: производственное обучение. Оно означает выполнение некоторой работы одновременно с обучением, которое происходит благодаря сотрудничеству людей, работающих в одной команде и обладающих разными навыками.

Проверяйте код перед его регистрацией

У вас в компании могут быть самые хорошие стандарты кодирования, но как вы проводите их в жизнь? Как говорится, доверяй, но проверяй. Программирование на одном и том же уровне и регулярная проверка проекта — вот конкретные способы проверить состояние кодовой базы. В ходе типичной проверки проекта вы можете взять некоторую типичную программу и обсудить ее открыто и коллективно. Код может быть реальным отрывком из проекта, который написали некоторые из ваших сотрудников, или, чтобы избежать эмоциональной реакции, точно написанной частью кода, иллюстрирующей тезисы, которые вы хотите огласить.

Для того чтобы провести в жизнь стандарты кодирования, вы можете также рассмотреть применение политики регистрации кода в вашей системе управления исходным кодом — Microsoft Team Foundation Server (TFS), TeamCity или другой системе. Можно ли автоматизировать этот процесс?

Сегодня почти любой инструмент управления исходным кодом предлагает способы контроля над зарегистрированными файлами. Например, в системе TFS есть управляемая регистрация — по существу, регистрация по правилам. Иначе говоря, файлы будут приняты в систему, только если они удовлетворяют установленным правилам. Когда вы принимаете решение создать управляемую регистрацию, система TFS предлагает вам указать, какой сценарий сборки использовать. Файлы будут зарегистрированы, только если выполнены определенные правила сборки.

В системе TFS сборка — это просто сценарий MSBuild, который можно настроить для решения множества задач. Система TFS поставляется со многими предопределенными задачами, которые могут быть объединены. Например, вы можете выбрать задачу анализа кода (FxCop) и задачу для управления выбранными списками тестов. Поскольку задача MSBuild — не что иное, как зарегистрированный компонент, который осуществляет законтрактованный интерфейс, вы можете создать новые задачи сами, чтобы добавить свои правила проверки.

Обратите внимание на то, что ReSharper от компании JetBrains — один из вышеупомянутых инструментов кодового помощника — предлагает в последнем выпуске ряд свободных инструментов командной строки, которые могут использоваться в задаче MSBuild для обнаружения дублированного кода и проведения типичных проверок, включая специальные проверки с пользовательскими шаблонами. Вам даже не нужна лицензия на использование ReSharper, чтобы использовать инструменты командной строки. Для получения дополнительной информации об инструментах командной строки ReSharper зайтите на сайт `http://www.jetbrains.com/resharper/features/command-line.html`.

Хороший проект в героях не нуждается

Разработчики, как правило, очень самолюбивы и, по крайней мере, в своих секретных мечтах, могут работать по 80 часов в неделю, чтобы спасти проект, доставить удовольствие заказчику и выглядеть героем в глазах менеджеров и коллег.

Перефразируя известную цитату поэта и драматурга Бертольда Брехта (Berthold Brecht), можно сказать: “Мы предпочитаем ситуации, в которых не надо проявлять

героизма". Потребность в героях и сопутствующее давление являются результатом неправильного планирования.

Иногда менеджеры с самого начала устанавливают нереальные сроки сдачи проекта. В других ситуация нарушение сроков может быть результатом возникших проблем.

Причинами давления, возникающего при разработке программного обеспечения, обычно являются неправильное планирование или отсутствие опыта и знаний. Обе ситуации можно предотвратить, если их вовремя распознать.

Мы не хотим быть героями, и хотя мы много раз сами были героями (как и большинство из вас, как мы предполагаем), героизм — это исключительная ситуация. Как известно, исключительных ситуаций в программном обеспечении следует избегать.

Поощряйте практику

Зачем профессиональные спортсмены почти каждый день тренируются в течение многих часов? Существует ли аналогия между разработчиками и профессиональными игроками? Это как посмотреть.

Одна точка зрения состоит в том, что разработчик и так каждый день практикуется на работе и не конкурирует с другими разработчиками. Если исходить из этого, то между программистами и спортсменами нет никакой аналогии.

Другая точка зрения состоит в том, что игроки очень часто выполняют основные движения, доводя их до автоматизма. Регулярное использование объектно-ориентированной технологии, шаблонов проектирования, стратегии кодирования и определенные компоненты API помогают поддерживать знания на высоком уровне и сделать их более доступными.

Замечание

Написав несколько книг по ASP.NET и приобретя многолетний опыт в области аутентификации и регистрации членства, Дино стал испытывать серьезное беспокойство при работе с ролевыми системами ASP.NET. "Приятель, — сказал он мне недавно, — а когда я в последний раз работал с ролями?" В результате он потратил много времени на создание инфраструктуры пользовательских интерфейсов и связанных с ними регистрационных систем.

Непрерывное изменение — часть сделки

Непрерывное изменение — эффективный способ описать динамику современных проектов программного обеспечения. Проект программного обеспечения начинается с идеи или даже туманного соображения в области бизнеса. Архитекторы и эксперты в предметной области сталкиваются с проблемой смягчения некоторых формальных требований, чтобы сделать оригинальную идею или деловые потребности немного более реальными.

Как показывает наш опыт, большинство проектов программного обеспечения напоминает движущиеся цели, а требования — механизм, перемещающий цель. Каждый раз после добавления нового требования изменяются контекст и динамика системы

(которая без этого требования работает прекрасно). Требования изменяются вследствие более точного понимания быстро изменяющейся проблемной области или из-за давления сроков.

Пересмотр требований (requirements churn) — общепринятый термин, обозначающий скорость изменения требований (функциональных и нефункциональных) в контексте проекта программного обеспечения. Высокая скорость пересмотра с высокой вероятностью приводит к появлению BBM.

Пересмотр всей системной архитектуры каждый раз, когда вводится новое требование, является единственным способом предотвращения BBM. Пересмотр всей системной архитектуры действительно требует рефакторинга и связан со значительными затратами. Главное — найти способ поддержать затраты на низком уровне. Рефакторинг вряд ли можно отнести к разряду факторов, повышающих стоимость проекта. Кроме того, невозможность рефакторинга вообще аннулирует стоимость проекта.

Замечание

Социальная сеть Twitter появилась в веб в 2010 году и предложила своим клиентам множество функций. Многие функции обеспечивались генерированием HTML-кода на лету при динамической загрузке данных в формате JSON. В 2012 году компания Twitter выполнила полный рефакторинг системы и выбрала рендеринг на серверной стороне. Этот рефакторинг имел архитектурный характер и, конечно, был затратным. Однако компания поняла его необходимость и сохранила несколько сотен пользователей. Таким образом, каким бы ни было это решение — правильным или неправильным, — сеть сохранила работоспособность.

Для архитектора ключевым инструментом является как архитектурный, так и проектный рефакторинг. Архитекторы не могут повлиять на историю, эволюцию бизнес-сценария и реальный мир. Архитектор должен постоянно исправлять ситуацию, но никогда не перестраивать проект. Именно в этом случае следует применять рефакторинг.

Вылезайте из болота

Даже если команда руководствуется лучшими намерениями и предпринимает все необходимые усилия, проект системы в некоторый момент может стать на сколькую дорожку. Формирование BBM обычно является медленным процессом, который ухудшает проект за относительно длительный период. Он возникает, заполняя ваши классы “грязным” кодом, пока значительная часть кода не выйдет из-под контроля.

В этот момент возникает серьезная неприятность.

Менеджеры оказываются перед выбором: заключить сделку с дьяволом и смириться с накоплением “грязного” кода или выполнить глубокую модернизацию, основанную на пересмотренных требованиях и новых архитектурных решениях.

Каково различие между полной модернизацией системы и переделкой проекта с нуля? С точки зрения призыва к действию различие минимально. Психологическая

сторона выбора, тем не менее, отличается. Призывая к модернизации, менеджмент требует, чтобы команда нашла и устранила недостатки. Призывая к переписыванию, менеджмент признает неудачу. Очень редко в программном обеспечении неудача считается приемлемым выбором.

Если менеджмент призывает к значительной реструктуризации существующей кодовой базы, значит, команда потерпела бедствие. В этом случае существующая кодовая база становится отвратительной разновидностью *унаследованного кода*.

Этот странный унаследованный код

В индустрии программного обеспечения мы часто наследуем существующий код, который приходится эксплуатировать, поддерживать или просто хранить и интегрировать с новыми программами. Этот код называют *унаследованным*. Однако основной проблемой, стоящей перед архитекторами и разработчиками, является не спасение существующего унаследованного кода, а предотвращение создания его новой разновидности.

Унаследованный код — это, во-первых, код, а во-вторых, унаследованный. В соответствии с Oxford Dictionary, слово *наследство* (legacy) означает нечто оставленное или переданное предшественником. Более того, в словаре есть специальное определение этого слова в контексте программного обеспечения: то, что устарело, но которое трудно заменить из-за его широкого использования.

По нашему мнению, унаследованный код — это любой созданный ранее, но нежелательный код. Унаследованный код продолжает существование только до тех пор, пока он сохраняет работоспособность. Нет никакой разницы между хорошо спроектированным и написанным кодом и плохо спроектированным и написанным кодом, если они оба являются работоспособными. Проблемы начинаются, когда вам приходится поддерживать и совершенствовать унаследованный код.

Идея унаследованного кода так или иначе связана с идеей катастрофы программного обеспечения. Однако наличие унаследованного кода в проекте само по себе не является катастрофой по сути. Однако его наличие является предвестником неприятностей. Как превратить унаследованный код — свой или чужой — в более управляемый код, который не мешает развитию и расширению проекта?

Замечание

Десять лет назад Майкл Физерс написал статью, в которой описал проблемы, связанные с унаследованным кодом, и стратегии их решения. Эту статью можно найти по адресу `http://www.objectmentor.com/resources/articles/WorkingEffectivelyWithLegacyCode.pdf`. Особую ценность этой статье придает отношение, которое автор выявил между унаследованным кодом и тестами. Иначе говоря, унаследованный код — это код без тестов.

Замечание

Слишком многие разработчики вынуждены работать с унаследованным кодом, в котором совершенно не учитываются фундаментальные принципы проектирования программного обеспечения и правила тестирования, из-за чего этот код превращается в катастрофу. Мы хотим указать на ресурс, содержащий описание общей стратегии превращения плохого кода в хороший с помощью сочетания методов рефакторинга и инструментов анализа кода: `http://blog.jetbrains.com/blog/2013/05/14/recording-refactoring-legacy-code-bases-with-resharper`.

Мат в три хода

В принципе восстановление программного обеспечения после катастрофы напоминает восстановление после травмы. Предположите на мгновение, что, находясь в отпуске, вы решили сделать долгую пробежку и получили сильное растяжение ахиллова сухожилия.

Да, для очень активного человека это катастрофа.

Вы вызываете врача, и он предлагает простую, но эффективную процедуру. Во-первых, он приказывает вам прекратить любую физическую активность, включая ходьбу. Во-вторых, накладывает повязку на травмированную лодыжку или фиксирует всю ногу. В-третьих, когда вы чувствуете себя лучше, врач предлагает вам сделать несколько шагов и остановиться, если почувствуете себя некомфортно. Этот способ хорошо работает, и много людей успешно излечиваются благодаря этой процедуре.

Такая же процедура применяется к проблемному программному обеспечению, и в большинстве случаев она оказывается успешной. Точнее, эта стратегия является разумной, но ее реальная эффективность зависит от серьезности проблем.

Остановка новой разработки

Предпосылкой любой стратегии, нацеленной на превращение плохо написанного кода в нечто намного более управляемое, является остановка разработки системы. Фактически добавление нового кода может только усугубить проблемы. Однако остановка разработки не означает прекращение работы над системой. Просто вы не добавляете новые функции, пока не выполните реструктуризацию существующей системы и не создадите исправленную кодовую базу, которая может содержать новые функции, не ставя под угрозу существующие.

Модернизация разрабатываемой системы напоминает ловлю сбежавшего цыпленка. Вы должны быть в очень хорошей форме, чтобы поймать его. Но находится ли в форме команда, которая потерпела неудачу?

Изолируйте проблемный код

Так же как врач накладывает повязку на поврежденную лодыжку, архитектор должен изолировать плохо написанные блоки кода. Но что означает *блок кода*? Вероятно, проще сказать, чем не является блок кода.

Блок кода — это не класс, хотя иногда он охватывает несколько классов и даже модулей. Блок кода идентифицирует поведение системы — выполняемую функцию — и содержит все компоненты программного обеспечения, связанные с этой функцией. Ключевой момент заключается в идентификации макрофункций и определении инвариантного интерфейса для каждого блока. Вы должны определить уровень, который реализует интерфейс и представляет собой фасад поведения, а затем изменить код в каждом из блоков и на каждом пути, ведущем к выполнению поведения за фасадом.

Соответствующий пример приведен на рис. 2.1. На этом рисунке представлен небольшой раздел запутанной кодовой базы, в которой два важных компонента — C1 и C2 — имеют слишком много связей с другими компонентами.

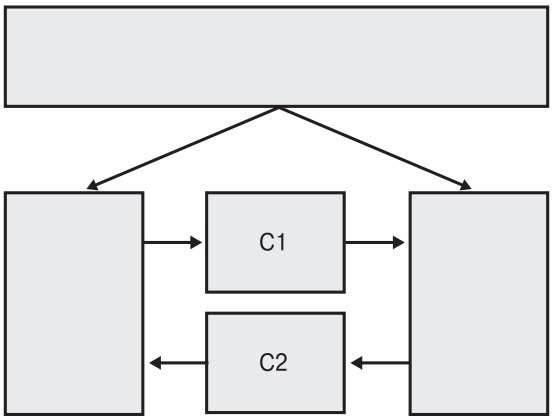

Рис. 2.1. *Раздел запутанной кодовой базы*

Наша цель — превратить рис. 2.1 во что-то более ясное, с более четко разделенными блоками. Следует понимать, что создать правильный проект с первой попытки невозможно. Сначала просто изолируйте проблемные блоки и не беспокойтесь о размере уровней, в которые вы вмешиваетесь. Возвращаясь к аналогии с врачом, вспомним, что он фиксирует всю ногу, даже если боль сосредоточена в области лодыжки. Тем не менее после нескольких дней отдыха врач может наложить повязку только в области лодыжки.

Аналогично изолирование проблемных блоков кода представляет собой итеративный процесс. На рис. 2.2 показан возможный способ изоляции блоков, изображенных на рис. 2.1.

Рассматривая рис. 2.2, вы могли бы предположить, что блоки C1 и C2 были продублированы; однако это лишь промежуточный этап, необходимый для того, чтобы получить две аккуратно разделенные подсистемы, вызываемые клиентом. Теперь разделенные подсистемы должны быть похожи на черные ящики.

Следует отметить, что блок кода может охватывать даже уровни, а иногда и ярусы. Наша окончательная цель, как и у врача, — уменьшить область, покрытую изолированными блоками.

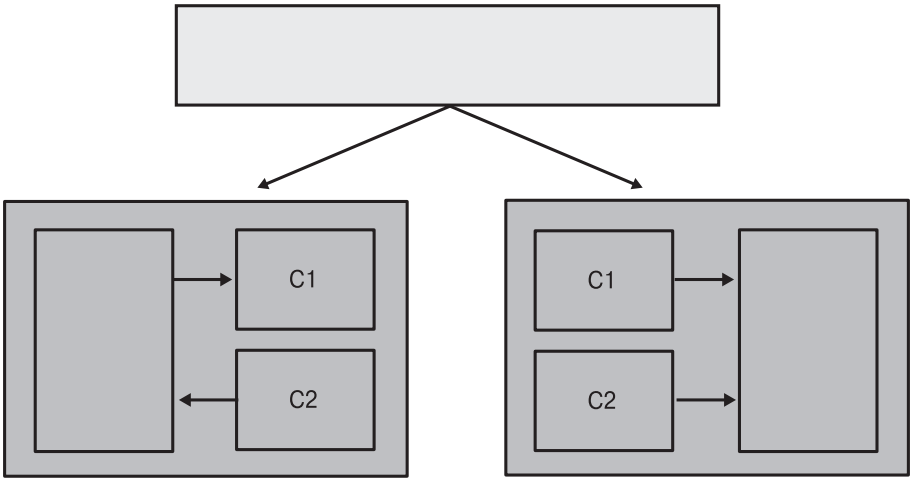

Рис. 2.2. *Промежуточный этап проекта, на котором клиент обращается к изолированному блоку с помощью нового законтрактованного интерфейса*

Терминология

В настоящей книге, начиная с главы 5, широко используется предметно-ориентированное проектирование (Domain-Driven Design — DDD). С понятием изолированного блока, описанного выше, тесно связана концепция DDD под названием *ограниченный контекст*. В проекте DDD *ограниченный контекст* также может быть черным ящиком, содержащим унаследованный код. В рамках подхода DDD ограниченный контекст имеет одну связанную с ним модель и может состоять из нескольких модулей. Следовательно, к одной и той же модели может относиться несколько модулей.

Тестовое покрытие

Если после того как вы завершите рефакторинг системы и превратите ее в набор аккуратно разделенных блоков, система просто сохранит работоспособность — это незначительное улучшение с точки зрения проектирования. На этом этапе останавливаться нельзя. Мы настоятельно рекомендуем провести тестирование на этом этапе и выяснить, работает ли система после рефакторинга.

В этом контексте тесты носят главным образом интеграционный характер, т.е. являются тестами, которые могут охватывать многочисленные модули, уровни и ярусы. Эти тесты трудно настраиваются — например, базы данных необходимо наполнять оперативными данными для моделирования, а службу надо подготовить к соединению — и долго выполняются. И все же они абсолютно необходимы. В своей вышеупомянутой статье Физерс (Feathers) использует термин *тестовое покрытие* (test covering), чтобы выделить тесты, нацеленные на определение инвариантов поведения, для дальнейшего изменения.

Замечание

Тесты играют ключевую роль в любом рефакторинге. Этот процесс необходимо завершать тестированием, чтобы измерить степень регрессии. Однако в некоторых ситуациях тесты целесообразно выполнять до изоляции проблемных блоков.

Непрерывный рефакторинг

После первой итерации тестовых покрытий вы должны стабилизировать систему и обеспечить определенную степень управления ее поведением. Вы можете даже ввести в тесты определенные входные значения и проверить поведение системы. Затем вы должны начать цикл рефакторинга и попытаться упростить структуру черных ящиков, которые создали. При этом вы можете извлечь определенный код из черных ящиков и снова использовать его посредством новых интерфейсов (рис. 2.3).

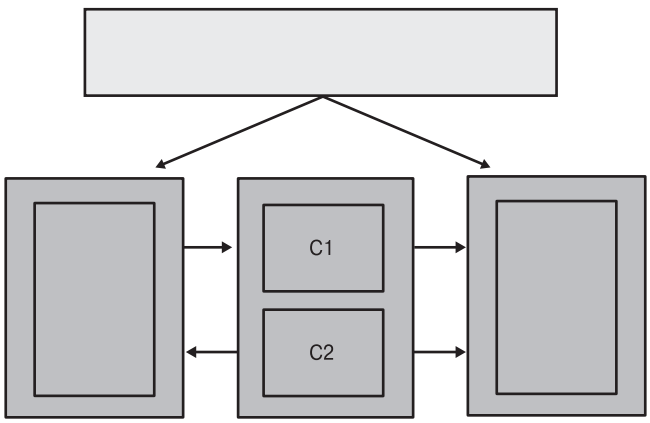

Рис. 2.3. *Дополнительный черный ящик, добавленный для разделения системы на более управляемые компоненты*

Как видите, теперь блоки C1 и C2 удалены из подсистем и инкапсулированы в новом черном ящике, который является предметом тестирования. Повторяя эту процедуру, вы можете постепенно уменьшить размер черных ящиков. Благодаря этому существующие интеграционные тесты можно переписать так, чтобы они стали намного ближе к модульным тестам.

Следует ли добавлять рабочую силу

В широко известной цитате Фредерика П. Брукса (Frederick P. Brooks) из его популярной книги *The Mythical Man Month* (Addison-Wesley, 1995)[4] утверждается, что добавление рабочей силы к опаздывающему проекту просто еще больше задерживает

[4] Русский перевод: Брукс Ф. *Мифический человеко-месяц, или Как создаются программные системы* — М.: Наука, 1988. — *Примеч. ред.*

его. Дополним цитату: к тому же это не оказывает значительного влияния на расписание. И все же, если проект опаздывает, первое, что приходит на ум, — увеличить количество сотрудников. Однако вследствие последовательных ограничений на проектные операции (например, отладка выполняется после разработки) увеличение количества сотрудников не приносит выигрыша. Согласно Бруксу, вынашивание ребенка занимает девять месяцев, и девять женщин никогда не родят ребенка за один месяц.

Итак, вопрос заключается в следующем: что делать, если проект опаздывает? Следует ли рассмотреть возможность увеличить количество сотрудников? Ответ зависит от результатов честного анализа причин, по которым проект опаздывает.

Требуется больше времени

Проект может опаздывать просто потому, что для завершения разработки каждой функции требуется больше времени, чем предполагалось. Если задачи являются последовательными по своей природе, то большее количество людей в команде всего лишь увеличит объем работы для менеджеров и, вероятно, снизит оптимальное использование ресурсов.

Действительность перемещает фокус внимания на оценки. Разработчики программного обеспечения, как известно, не в состоянии правильно оценить объем работы. Они уверенны, что до самого конца все будет идти хорошо и что они успеют все исправить, работая сверхурочно. Это отношение также усложняет мониторинг прогресса. В эпиграфе к этой главе именно об этом и говорится: проект постепенно опаздывает на один день. Если прогресс отслеживается своевременно, то отставание от расписания может быть наверстано без долгой сверхурочной работы. В худшем случае это время можно растянуть на более длительный промежуток времени.

Но даже если оценки объема работы корректны, то другим аспектом часто пренебрегают. У любого проекта есть и прямые, и косвенные затраты. Прямые включают такие статьи, как зарплаты и расходы на командировки. Косвенные затраты включают, например, приобретение оборудования и администрирование. Кроме того, существуют непредвиденные затраты: встречи, ревизии и все те небольшие задачи, которые не были переадресованы или не полностью поняты. Ошибки в оценках — распространенное явление. Они исчезают только благодаря опыту — вашему или экспертов. Очевидно, вы всегда должны прилагать все усилия, чтобы четко выписать каждую небольшую задачу, которая делает работу завершенной, и оценить ее, по крайней мере, с точки зрения временны*х* затрат.

В рамках прагматического подхода после завершения проекта реальные затраты всегда сравнивают с предварительными оценками. Разница между реальными затратами и предварительными оценками является коэффициентом, положительным или отрицательным, на который в следующий раз будут умножаться все оценки.

В зависимости от рабочей модели в проектах программного обеспечения есть два основных фактора: фиксированная цена или время/материалы. В первом случае, если вы понимаете, что вам требуется больше времени, то как руководитель проекта пытаетесь найти способ наверстать расписание. Вы пытаетесь снизить время разработки и тестирование в ущерб качеству, ограничивая документирование и любое действие, которое не является строго необходимым для сдачи окончательных тестов на проверку.

Вы также пытаетесь пересмотреть список требований, чтобы увидеть, можно ли что-либо уже согласованное повторно сформулировать как запрос на изменение. Во втором случае можно вычислить разницу между оценками и фактически затраченным временем на прошлых итерациях и внести поправки для каждой новой итерации.

Необходимость в более опытных сотрудниках

Другая причина, по которой проект может опоздать, состоит в том, что реализация определенных функций занимает больше времени, потому что некоторые люди не справляются с задачами. Если вам нужно больше знаний и опыта, не бойтесь нанять самого талантливого специалиста, который только сможете найти. Однако, разыскивая талантливых людей, вы должны четко понимать, зачем они вам нужны. У вас могут быть эксперты, чтобы обучить ваших людей или самостоятельно решить проблему. Ни одну из этих возможностей нельзя назвать предпочтительнее другой.

Обучение увеличивает добавленную стоимость, потому что результаты обучения остаются в компании и увеличивают ее активы. В то же время обучение — даже при поставке нестандартного программного обеспечения для автоматизированного обучения — обычно решает проблемы на довольно общем уровне и может занять дополнительное время, которое будет адаптировано к текущему проекту. С другой стороны, обращение к консультантам теоретически более эффективно, но требует, чтобы вы предоставили полный доступ экспертов к коду, людям и документации. Чем сложнее кодовая база, тем больше времени потребуется затратить и менее надежные результаты можно получить.

Эксперты — не волшебники; в программной инженерии нет волшебников. Если эксперт выглядит волшебником, он, скорее всего, просто фокусник, как и в обычной жизни.

Резюме

Несмотря на название, программная инженерия — это не инженерная деятельность (по крайней мере, в обычном смысле этого термина). Программное обеспечение носит слишком динамический и эволюционный характер, чтобы описать его разработку в виде совокупности фиксированных правил.

Главный враг проектов программного обеспечения — `BBM`. Он жестко связан с постепенным увеличением и продолжительностью жизни проекта. Постепенное увеличение проектов — это реальность; важно иметь эффективную стратегию борьбы с этим явлением. Это может потребовать человеческих и материальных ресурсов для проведения переговоров с руководителями проекта, клиентами и заинтересованными сторонами. Опыт позволяет вам разумно судить о функциях, которые скорее всего потребуются, и лучше понимать клиента, чтобы выявить его действительные нужды.

Не все проекты равнозначны, и понимание, какова продолжительность жизни проекта, — еще один чрезвычайно важный фактор. Нет смысла затрачивать одинаковые усилия и нести одинаковые затраты при разработке недолговременного проекта и критически важной системы. Сложностью нужно управлять там, где она действительно существует, и не создавать ее там, где ее нет и не предполагается.

Механика проектов программного обеспечения иногда выглядит извращенной, но успешные проекты возможны. Что бы плохое ни случилось с вашим проектом программного обеспечения, это не происходит мгновенно. Если вы можете своевременно обнаружить и устранить проблему, то потратите на ее предотвращение намного меньше времени, чем на восстановление системы.

Шутливые афоризмы

На сайте `http://www.murphys-laws.com` читатели найдут большой список шутливых законов, связанных с компьютерами (и не только).

- Все модели неправильные, но некоторые из них приносят пользу. — Джордж Е.П. Бокс (George E. P. Box).
- Ходить по воде и разрабатывать программное обеспечение легко, если и то и другое заморожено.
- Утечка масла только увеличивается.