Министерство образования и науки Российской Федерации
Ярославский государственный университет им. П. Г. Демидова
Кафедра компьютерных сетей

И. В. Парамонов, А. М. Васильев

Инженерия программных систем и комплексов на основе гибкой методологии разработки

Учебно-методическое пособие

Рекомендовано
Научно-методическим советом университета
для студентов, обучающихся по направлению
Прикладная математика и информатика

Ярославль
ЯрГУ
2015

УДК 004.41 (072)
ББК 3973.2-018.2я73
П 18

Рекомендовано
Редакционно-издательским советом университета
в качестве учебного издания. План 2015 года.

Рецензент
кафедра компьютерных сетей Ярославского
государственного университета им. П. Г. Демидова.

Парамонов, Илья Вячеславович

П 18 Инженерия программных систем и комплексов на основе гибкой методологии разработки : учебно-методическое пособие / И. В. Парамонов, А. М. Васильев ; Яросл. гос. ун-т им. П. Г. Демидова. — Ярославль : ЯрГУ, 2015. — 47 с.

Учебно-методическое пособие описывает базовые принципы организации деятельности команд разработчиков программных систем согласно гибкой методологии. Разумное следование данным принципам позволит любой организации повысить эффективность как внутренних процессов, так и взаимодействия с заказчиком. Для каждой группы описанных принципов пособие предлагает набор вопросов и упражнений для проведения семинаров.

Предназначено для студентов, обучающихся по направлению 01.03.02 (010400.62) Прикладная математика и информатика (дисциплина «Программная инженерия», цикл Б3), очной формы обучения.

Библиогр.: 14 назв.

УДК 004.41 (072)
ББК 3973.2-018.2я73

© ЯрГУ, 2015

Введение

Гибкая методология разработки программного обеспечения — это современная альтернатива традиционной «тяжеловесной» методологии разработки, описываемой во многих учебниках по программной инженерии и используемой во многих крупных компаниях. И хотя данная методология ещё сравнительно молода (отдельные попытки применения свойственных для неё подходов датируются серединой 1990-х гг.) и не является столь системно сформированной по сравнению с традиционной методологией, история которой насчитывает уже полвека, её результативность подтверждена множеством реализованных проектов.

Вероятно, главным фактором успеха гибкой методологии является эффективное управление изменениями. Традиционная методология весьма тяжеловесна, поэтому внесение изменений в требования во время реализации проекта часто становится болезненным при её применении. В то же время скорость изменения требований и бизнес-условий в современном мире только нарастает, что делает гибкую методологию желательным инструментом для многих групп разработчиков.

Следует понимать, что вопреки мнениям многих разработчиков и даже некоторым литературным источникам гибкая методология не перечёркивает традиционной инженерии. Она лишь предлагает другие решения тех же самых проблем, часто менее формальные и более компромиссные. Эти решения обычно сложнее тиражировать в больших компаниях, но они хорошо подходят для небольших, хорошо сфокусированных групп квалифицированных разработчиков, помогая этим группам достигать высокой продуктивности и конкурентоспособности на рынке.

Недостаточное или излишне схематичное рассмотрение применения гибкой методологии в традиционных учебниках по программной инженерии побудило авторов к созданию настоящего учебно-методического пособия. Изложение в пособии отталкивается от основных принципов гибкой методологии, которые даются в соответствии с книгой [1]. Описание каждого принципа включает краткую формулировку его основной идеи, назначения, возможных следствий применения в команде разработчиков, а также список литературы для более глубокого изучения. В конце каждой главы приведены вопросы для обсуждения на семинарах.

1. Гибкая методология разработки

1.1. Характерные черты гибкой методологии разработки. Основой гибкой методологии является Agile Manifesto [2], в котором сформулированы базовые принципы, формирующие процессы внутри команды разработчиков. Данный манифест явился результатом осмысления техник, успешно применявшихся различными командами для реализации качественных продуктов. Целью этого документа было представление широкой общественности фундаментальных идей, лежащих в основе этих техник, чтобы инициировать обсуждение и улучшение методик, применяемых при разработке программного обеспечения.

В манифесте выделены следующие идеи:

— люди и взаимодействие важнее процессов и инструментов;
— работающий продукт важнее исчерпывающей документации;
— сотрудничество с заказчиком важнее согласования контракта;
— готовность к изменениям важнее следования изначальному плану.

При этом следует отметить, что данные утверждения постулируют приоритет определённых ценностей процесса разработки над другими, но не отрицают их.

1.2. Принципы гибкой разработки. Принципами[1] гибкой разработки называются фундаментальные положения, которые, реализуясь в тех или иных правилах и процедурах процесса разработки, позволяют придать этому процессу черты гибкости и на практике воплотить идеи Agile Manifesto, перечисленные в предыдущем пункте. Принципы заключают в себе квинтэссенцию опыта множества команд разработчиков.

Не существует единственного «правильного» набора принципов, по которым можно было бы заключить, что некоторый процесс использует гибкую методологию. Об этом можно судить лишь по результатам этого процесса, в первую очередь по его способности эффективно использовать обратную связь и безболезненно реагировать на изменения в требованиях. Однако данные принципы являются полезными, т. к. позволяют поэтапно и эволюционно достигать внедрения гибкой методологии в рамках команды разработчиков или компании по производству про-

[1] В англоязычной литературе используется термин practice. Однако перевод этого термина как «практика» не вполне отражает фундаментальную природу данных положений в рамках методологии. В связи с этим, в данном пособии авторы используют термин «принцип».

граммного обеспечения. Рассмотрению данных принципов посвящены главы 2-8 настоящего пособия.

1.3. Существующие методики в рамках гибкой методологии. Конкретные методики гибкой разработки могут реализовывать основополагающие принципы различным образом. Одни из них фокусируются на применении специальных процессов для организации всей команды, другие — на повышении квалификации каждого разработчика в отдельности. Рассмотрим особенности нескольких популярных методик гибкой разработки.

Экстремальное программирование (extreme programming, XP) — методика, нацеленная на взаимодействие в группе высококвалифицированных разработчиков, способных за счёт применения ряда подходов создавать качественный продукт [3]. Эти подходы нацелены в первую очередь на обеспечение эффективного взаимодействия внутри команды, поддержание простоты и понятности кодовой базы проекта, доставку в первую очередь необходимого функционала. Название методика получила из-за установления жёстких требований к процедурам проверки качества кода, таких как парное программирование, повсеместное модульное тестирование, постоянное выполнение рецензирования кода.

Методика Scrum [4] описывает организационную структуру команды, где каждый разработчик играет определённую роль, определяет политики распределения задач во времени, отталкиваясь от итеративной модели разработки, выделяет необходимые для управления процессом артефакты. Данные аспекты наиболее понятны менеджерам проектов, желающим эффективно обрабатывать постоянно изменяющиеся требования заказчика в рамках процесса разработки, но также полезны всей команде разработчиков, так как их действия детерминированы в каждой фазе проекта.

Методика Lean [5] описывает модели поведения членов команды, нацеленной на постоянное улучшение своей продуктивности за счёт удаления ненужных шагов в процессе работы, повышение квалификации сотрудников, предоставление возможности принимать решения са́мому квалифицированному члену команды, на предельно быструю доставку результата заказчику и т. д. Данная методика базируется на подходах из автомобильной индустрии, накладывающей схожие требования по обеспечению качества и скорости доставки решений потребителю.

1.4. Создание собственных методик. Команды, внедряющие в свой процесс гибкую методологию, часто делают серьёзную ошибку, пытаясь сломать существующие процессы и выстроить новые согласно методикам, изложенным в литературе. Такие действия скорее всего приведут к неудаче текущих проектов и даже могут привести компанию к разрушению.

Для результативного внедрения принципов всегда следует чётко осознавать их сильные и слабые стороны, реально оценивать возможности своей команды, а также идентифицировать ключевые риски проекта, который необходимо реализовать. При этом внедрение следует осуществлять поэтапно, адаптируя предлагаемые подходы к существующему в рамках компании процессу. Вот почему для большинства команд разработчиков более целесообразным представляется внедрение гибкой методологии на основе принципов, а не готовых методик. В результате менеджеры и разработчики получают возможность создать свою собственную методику, обеспечивающую высокую результативность в условиях конкретного процесса разработки с учётом возможностей членов команды.

1.5. Вопросы и упражнения для семинара

1. Оцените возможности применения гибкой методологии в рамках каскадной и итеративной моделей процесса разработки?
2. Объясните, каким образом идеи манифеста гибкой разработки нашли своё выражение в конкретных методиках, упомянутых в п. 1.3.
3. Выделите основные риски процесса разработки, на управление которыми направлена гибкая методология и конкретные методики в рамках гибкой методологии.
4. Почему попытки одномоментного внедрения гибкой методологии в процесс разработки практически всегда заканчиваются неудачей?

2. Фундаментальные принципы гибкой разработки

2.1. Работайте на результат (Work for outcome). Нацеленность на достижение результата всегда должна быть основным приоритетом команды разработчиков. Следуя данному принципу, разработчик ориентируется на результат своей работы в виде готового продукта, удовлетворяющего требованиям заказчика. При этом разработчик понимает, каким образом каждое его действие (проектирование интерфейса, реализация функции, тестирование модуля и т. д.) вносит вклад в достижение этой цели.

Назначение. Принцип обеспечивает более эффективное использование человеческих ресурсов команды за счёт фокусирования деятельности её членов. В результате общая продуктивность команды повышается за счёт синергетического эффекта от совместной работы.

Следствия:

1. Разработчики начинают более осмысленно решать свои задачи, т. к. имеют чёткий ориентир (работоспособный продукт). При этом пропадает весьма распространённый стереотип поведения, когда разработчик лишь выполняет действия, предписанные его начальником, не задумываясь о том, для чего эти действия нужны.

2. При возникновении проблем (например, при обнаружении ошибок в разрабатываемом программном обеспечении) члены команды стараются в первую очередь предложить решение по устранению этих проблем, а не занимаются выяснением, по чьей вине эта проблема возникла.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 12–14]; [6, § 1].

2.2. Непродуманные исправления заводят в тупик (Quick fixes become quicksand). Иногда, сталкиваясь с ошибками в программах, разработчики допускают быстрые, непродуманные исправления без полного понимания источника проблемы. Такие ситуации легко распознаются по таким фразам разработчиков: «Сейчас я прибавлю к этой переменной единицу, и всё будет работать правильно», «Непонятно почему, но вроде бы ошибка исчезла» и т. д. Особенно много непродуманных исправлений возникает в ситуациях, когда разработчики стеснены во времени из-за приближения сроков сдачи проекта или его этапа.

При кажущейся безобидности, исправления, внесённые без детального понимания, могут иметь очень серьёзные последствия и даже могут завести весь проект в тупик: если разработчик сам не понимает причину проблемы, он не может гарантировать отсутствие побочных эффектов своих исправлений, причём эти побочные эффекты могут затрагивать как непосредственно изменённый код, так и другой код, явно или неявно от него зависящий. Если же внесение непродуманных исправлений становится обычной практикой разработчиков, исходные тексты проекта быстро деградируют и превращаются в так называемый «спагетти-код», который практически невозможно сопровождать.

Назначение. Принцип препятствует накоплению низкокачественного кода, который не до конца понятен, не проанализирован на предмет побочных эффектов, не обладает чётко обоснованным поведением и поэтому сам по себе может провоцировать появление новых ошибок. В результате недопущения появления такого кода в проекте снижаются риски, связанные с сопровождением программного продукта.

Следствие. Если разработчики всегда разбираются в причинах возникающих проблем и затем осмысленно вносят исправления, программный код становится понятным для каждого разработчика, а не только для автора этого кода. При нарушении данного принципа смысл внесённых исправлений непонятен даже для автора и тем более для других членов команды разработчиков.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 15–17]; [6, § 31].

2.3. Критикуйте идеи, а не людей (Criticize ideas, not people). Оценка и обсуждение идей и подходов к решению задач разработки являются неотъемлемой частью процесса, причём так или иначе все члены команды разработчиков вовлекаются в эти виды деятельности. Из-за различия точек зрения и квалификации разработчиков такого рода обсуждения могут перерастать в продолжительные споры по поводу архитектуры, пользовательского интерфейса, путей реализации того или иного функционала и т. д.

Принцип «критикуйте идеи, а не людей» призывает разработчиков обоснованно критиковать идеи, отстаивать свои точки зрения на основе знаний, логики и опыта и никогда не переходить на личности.

Назначение. Принцип нацелен на создание здоровой среды для выработки правильных решений задач в ходе дискуссий внутри команды разработчиков.

Следствия:

1. Разработчики не боятся высказывать свои идеи, даже если они не согласуются с идеями других участников, т. к. ситуация, когда разработчик предложил неправильную или неудачную идею, воспринимается командой как обычный рабочий момент, а не свидетельство его глупости, некомпетентности и т. д.

2. Поскольку решение, подлежащее реализации, в каждом конкретном случае формируется коллегиально и объективно, снижается риск принятия неправильного и неоптимального решения.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 18–22].

2.4. Будьте смелы в трудных ситуациях (Damn the torpedoes, go ahead). Иногда, решая свои текущие задачи, разработчики находят дефекты и/или различные проблемы в существующем программном коде. Часто эти дефекты бывают внесены другими разработчиками.

В том случае, когда обнаруженный дефект мешает решению текущей задачи, разработчик должен попытаться сначала устранить найденную проблему и только потом переходить к решению исходной задачи. Неправильной является попытка внесения «быстрых», необдуманных исправлений (см. п. 2.2), которые позволяют решить текущую задачу без устранения найденного дефекта. В таком случае дефект оставляется неисправленным, а внесённые изменения могут только усугубить ситуацию, усложнив исправление дефекта впоследствии.

Если же обнаруженный дефект не мешает решению исходной задачи, то следует обязательно сообщить о нём другим членам команды разработчиков и ни в коем случае не «заметать под ковёр» найденную проблему. Озвученная проблема может быть занесена в список решаемых задач и исправлена в плановом порядке, тогда как неозвученная проблема становится «миной замедленного действия», которая может проявить себя в самый неподходящий момент, и тогда её уже придётся решать экстренно.

Следует отметить, что применять данный принцип на практике часто бывает сложно, в том числе по причинам психологического характера: исправление дефектов требует времени, а его, как правило, в процессе разработки программного обеспечения всегда не хватает.

Назначение. Принцип побуждает активно решать существующие в коде проблемы, а также предупреждать появление новых проблем. Он борется с инертностью разработчиков и заставляет их проявлять сме-

лость в решении профессиональных задач, которая вознаграждается снижением рисков в поддержке программного продукта.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 23–25]; [6, § 2].

2.5. Вопросы и упражнения для семинара

1. Почему принцип «работайте на результат» столь критичен для команд разработчиков, придерживающихся методологии гибкой разработки, но может уходить на второй план в командах, придерживающихся традиционной методологии? К каким последствиям может приводить нарушение этого принципа в тех и других командах?
2. Объясните, почему механическое выполнение требований начальника может быть пагубным для процесса разработки. Какие требования предъявляет принцип «работайте на результат» менеджеру проекта, реализуемого в рамках гибкой методологии?
3. Вспомните из собственной практики случай, когда Вы пренебрегли принципом «непродуманные исправления заводят в тупик» и это повлекло негативные последствия. Предложите шаги, которые надо было предпринять для правильного решения проблемы.
4. Предложите способы борьбы с внесением разработчиками непродуманных изменений.
5. Объясните, в чём заключаются риски сопровождения продукта, обусловленные непродуманными исправлениями.
6. Найдите взаимосвязь между принципами «работайте на результат» и «критикуйте идеи, а не людей». Как эта взаимосвязь может оказаться полезной при внедрении гибкой методологии в процесс разработки?
7. Представьте, что при разработке некоторого программного продукта обнаружен серьёзный архитектурный дефект, который существенно мешает решению текущих задач разработчиков; при этом исправление данного дефекта потребует столь больших временных затрат, что сроки сдачи продукта в эксплуатацию будут сорваны. Предложите возможные варианты решения данной проблемы.

3. Принципы гибкого профессионального развития

3.1. Будьте готовы к изменениям (Keep up with change). История развития вычислительной техники связана с решением всё более сложных задач в новых прикладных областях. Это стимулирует разработчиков и исследователей создавать новые технологии. В результате каждые 5–7 лет требования рынка к навыкам разработчиков значительно изменяются. Чтобы всегда оставаться востребованным на рынке труда, разработчику следует следить за новыми технологиями и на регулярной основе повышать свою квалификацию.

Источниками информации о новых технологиях могут служить семинары, конференции, списки рассылок, рабочие группы и статьи. Для лучшего восприятия информации разработчику следует принимать активное участие в очных встречах, выступать на них с докладами и задавать вопросы, а также самостоятельно публиковать статьи о новых и перспективных технологиях. В основе таких статей могут лежать проекты, создаваемые разработчиками в свободное время специально для экспериментирования с конкретной технологией.

Назначение. Принцип побуждает программиста изучать новые технологии, которые позволяют ему эффективнее решать поставленные задачи. При этом разработчик учится объективно оценивать положительный и отрицательный эффекты от применения каждой технологии.

Следствие. Когда на рынке появляется новая технология, разработчик легко воспринимает её концепции, т. к. знаком с её аналогами, и потому может с лёгкостью экстраполировать свой опыт.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 28–30]; [6, § 5]; [7, § 44, 47].

3.2. Инвестируйте в свою команду (Invest in your team). Команды разработчиков состоят из людей, имеющих различную квалификацию и различные области специализации. Для решения задач в проекте необходимо их эффективное взаимодействие друг с другом. Оно становится возможным, если все члены команды знакомы с терминологией и основополагающими идеями используемых в проекте технологий.

Удобным подходом к решению задачи повышения профессиональной эффективности команды являются еженедельные семинары, во время которых разработчики по очереди делятся знакомыми им концепци-

ями и технологиями. Семинар обычно начинается с краткого выступления докладчика, затем происходит обсуждение затронутой темы внутри команды. В ходе дискуссии важно объективно рассмотреть преимущества и недостатки разбираемой технологии, а также рассмотреть возможности её применения в текущих и будущих проектах команды.

Назначение. Принцип призывает повышать квалификацию каждого разработчика внутри команды, а также эффективность взаимодействия членов команды.

Следствия:

1. Разработчики начинают лучше ориентироваться в современных технологиях, обсуждения подходов к решению задач становятся более содержательными.

2. Общий уровень квалификации команды повышается, становится возможной реализация всё более сложных, высокооплачиваемых проектов.

3. При подготовке к выступлению разработчик инвестирует дополнительное время в саморазвитие, повышая свой уровень понимания конкретной области.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 31–33]; [7, § 17].

3.3. Умейте разучиваться (Know when to unlearn). Информационные технологии всегда связаны с решением задач в рамках ограничений, определяемых техническим обеспечением и используемыми программными средами. Со временем эти ограничения изменяются: некоторые из них становятся нерелевантными, другие изменяют свой масштаб. Если при применении новой технологии разработчик ориентируется на устаревшие ограничения, тогда созданные им решения будут менее эффективными и более сложными в поддержке.

Во время изучения новых технологий следует оценивать степень влияния своих знаний и подходов к решению задач на восприятие изучаемой технологии, так как есть опасность потери её преимуществ на практике. Чтобы избежать этого, следует изучать не только конкретные решения новой технологии, но и её фундаментальные идеи.

Назначение. Принцип призывает программиста не отвергать новые подходы из-за их несоответствия собственным профессиональным установкам, а также расширять набор используемых инструментов и обоснованно выбирать подходящий инструмент для каждой задачи.

Следствие. Изучение новых технологий и новых парадигм позволяет разработчику эффективнее решать стоящие перед ним задачи в текущих проектах без изменения технологического стека.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 34–36].

3.4. Спрашивайте, пока не поймёте (Question until understand). Часто при постановке задачи на разработку той или иной функциональности продукта заказчик явно описывает поведение системы, которого необходимо добиться в результате решения. Так происходит, когда заказчику проще описать сходное решение, которое он видел в других системах, чем сформулировать стоящую перед ним проблему. В этом случае разработчик должен самостоятельно выяснить все аспекты решаемой проблемы, провести анализ соответствия предложенного поведения решаемой задаче и, возможно, предложить более удачное альтернативное решение, концептуально согласующееся с целями заказчика.

Серия вопросов «почему?» позволяет эффективно пробиться через наслоение неявно принятых решений до сути проблемы, стоящей перед заказчиком. Задавая эти вопросы, разработчик должен направлять собеседника, поощрять обсуждение и обоснование решений, в том числе тех, которые кажутся заказчику очевидными. Кроме того, усилия, потраченные разработчиком на выяснение сути проблемы, окупятся во время реализации функциональности, так как позволят не упустить небольшие детали и уберегут от неверных решений.

Назначение. Принцип направлен на минимизацию рисков, связанных с непониманием или неправильным пониманием разработчиками истинных целей и желаний заказчика.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 37–39].

3.5. Чувствуйте ритм (Feel the rhythm). Процесс разработки программного обеспечения включает в себя множество разнообразных видов деятельности (взаимодействие с заказчиком, разработка нового функционала, тестирование, рефакторинг и т. д.), которые необходимо постоянно выполнять, чтобы проект двигался вперёд. При плохо организованном процессе разработчики зачастую не представляют, что необходимо делать сейчас, а что завтра или на следующей неделе. Непредсказуемость давит на людей, не даёт сконцентрироваться на текущей задаче и может привести к утрате мотивации, так как у разработчи-

ков появляется ощущение разочарования ввиду отсутствия результатов приложенных усилий.

Гибкая методология призывает выстраивать работу каждого сотрудника и команды в целом на основе разбиения рабочего времени на небольшие промежутки, в рамках которых определяется чёткая и конкретная цель, которую необходимо достичь. Например, для разработчика таким промежутком может являться рабочий день: утром он выбирает несколько задач, а целью ставит их полное решение, абстрагируясь от всех прочих задач проекта.

Назначение. Принцип призывает каждого разработчика и команду в целом вырабатывать процедуры эффективного управления рабочим временем для достижения конкретных результатов, видимых как самой командой, так и заказчиком проекта.

Следствия:

1. Достижение небольших, понятных целей мотивирует разработчиков на продолжение реализации проекта.

2. Разработчик, эффективно управляющий своим временем, учится оценивать свою продуктивность, что помогает команде при планировании будущих работ.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 40–42]; [8].

3.6. Вопросы и упражнения для семинара

1. Проанализируйте причины устаревания знаний и навыков. Приведите примеры быстро и медленно устаревающих знаний. Какие категории знаний делают программиста конкурентоспособным?
2. Следует ли разработчикам быть компетентными в технологиях, применяемых дизайнерами, менеджерами, тестировщиками?
3. Проанализируйте, как принцип «инвестируйте в свою команду» может повлиять на выбор технологий для будущего проекта.
4. Достаточно ли разработчику участия в семинарах, обсуждаемых в разделе 3.2, для того, чтобы оставаться конкурентоспособным на рынке труда? Приведите примеры.
5. Применим ли шаблон программирования «модель-вид-контроллер» для создания консольных приложений?
6. Изучите техники управления рабочим временем «Pomodoro technique» [8] и «Getting things done» [9]. Оцените их применимость в рамках гибкой методологии разработки.

4. Принципы гибкого проектирования и гибкой доставки разрабатываемого продукта пользователю

4.1. Позвольте заказчику принимать решения (Let customers make decisions). В ходе проекта принимается множество решений. Те решения, которые касаются архитектуры приложения, взаимодействия его компонентов и прочих технических аспектов, безусловно должны приниматься разработчиками. Однако в каждом проекте необходимо принимать также бизнес-решения, которые непосредственно касаются пользовательских качеств программного обеспечения и сценариев его использования. Такие решения разработчик не может принять самостоятельно в силу недостаточного знания предметной области, бизнес-реалий и т. д. Вот почему при необходимости принять решение такого типа разработчики должны обязательно обращаться к заказчику. При этом разработчик может предложить несколько возможных вариантов решений, а уже заказчик выберет из них подходящий и примет окончательное решение.

Назначение. Принцип стимулирует разработчиков отделять свою сферу ответственности, включающую технические вопросы, от сферы ответственности заказчика, включающей бизнес-решения.

Следствие. Поскольку в результате применения принципа каждое решение принимается людьми в сфере их компетенции, то снижается вероятность допустить серьёзные ошибки в ходе реализации проекта. Особенно это актуально для бизнес-решений, где такого рода ошибки могут оказаться чрезвычайно дорогими и привести либо к необходимости серьёзной переделки продукта, либо вообще сделают продукт непригодным для эксплуатации конечным пользователем.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 45–47].

4.2. Архитектура должна направлять, а не диктовать (Let design guide, not dictate). В традиционной инженерии проектирование (выработка архитектуры) является центральным этапом разработки, а документы, описывающие архитектуру, содержат детальные указания относительно реализации проекта. В рамках гибкой методологии значимость архитектурных решений несколько снижается, т. к. жизненный цикл продукта не содержит выделенного этапа проектирования (он заменён

серией коротких сессий проектирования на каждой итерации), совокупный объём вырабатываемой документации мал, а изменения в архитектуре могут быть осуществлены на любой стадии реализации проекта. Вследствие этого и формулируется принцип, утверждающий, что архитектура должна использоваться как совокупность фундаментальных идей об устройстве продукта и рекомендаций по его реализации, а не как руководство к действию.

Однако не следует считать, что проекты, реализуемые в рамках гибкой методологии, могут обойтись вообще без проектирования. Как и в случае традиционной методологии разработки, отсутствие архитектуры влечёт за собой серьёзные риски, связанные с невозможностью при определённых условиях добавить новую функциональность в проект, а также с появлением расходящихся или неконсистентных реализаций сходного функционала. Именно поэтому в каждом проекте важно найти баланс, определив минимально необходимый набор архитектурных артефактов.

Назначение. Принцип нацелен на оптимизацию работ по выработке архитектуры продукта в условиях непрерывных изменений.

Следствия:

1. Разработчики начинают осмысленно, неформально, критически относиться к архитектуре, что придаёт дополнительную устойчивость проекту, защищая его от бездумного претворения в жизнь ошибочных или недостаточно проработанных архитектурных решений.

2. Поскольку изменение архитектуры в рамках проекта поощряется, то архитектура эволюционирует вместе с проектом и помогает ему двигаться вперёд, а не устаревает и не становится источником накопления дефектов и неоптимальных решений.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 48–51].

4.3. Обоснуйте использование выбранных технологий (Justify technology use). Правильный выбор используемых в проекте языков, технологий и инструментальных средств может существенно снизить трудозатраты на разработку продукта, тогда как неправильный — привести к выходу проекта за рамки заявленных сроков и бюджета, а иногда даже к его краху. Вот почему выбор средств реализации проекта является ответственным мероприятием и должен сопровождаться чётким обоснованием, почему те или иные технологии представляются более предпочтительными, чем другие. Чтобы осуществить такой выбор,

разработчики, как минимум, должны ответить на следующие вопросы: «Действительно ли выбранная технология позволяет решить поставленную задачу? Окажутся ли возможности разработчиков в дальнейшем ограничены выбранной технологией или стеснены ею? Какова будет стоимость поддержки продукта?»

Недопустимо основывать выбор технологий на рассуждениях такого рода: «Мы всегда применяли эту технологию, поэтому в данном проекте будем продолжать её использовать», «Я никогда не использовал этот фреймворк, поэтому самое время попробовать его в реальном проекте, чтобы потом написать в резюме, что я умею его использовать». Подобные соображения не принимают в рассмотрение интересы проекта и потому ставят его под угрозу, в случае если выбор окажется неудачным.

Назначение. Принцип заставляет разработчиков чётко аргументировать, почему технологии и инструменты, которые планируется применить в процессе разработки, являются подходящими и какие конкретные преимущества они несут для проекта.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 52–54].

4.4. Продукт должен быть готов к выпуску в любой момент (Keep it releasable). Добавляя новую функциональность или исправляя ошибки, разработчики вносят изменения в программный код. В течение промежутков времени, когда изменения внесены лишь частично, продукт часто оказывается временно неработоспособным и не может быть показан заказчику или использован конечным пользователем. Рассматриваемый принцип утверждает, что разработчики должны стремиться минимизировать длительность таких периодов неработоспособности путём выбора подходящей стратегии внесения изменений. Последняя включает в себя разбиение крупных задач на подзадачи, тестирование работоспособности продукта после решения каждой из подзадач с помощью автоматизированных тестов, предварительное проведение рефакторинга для упрощения внесения изменений, а также другие мероприятия.

Назначение. Принцип нацелен на повышение устойчивости и ремонтопригодности продукта, так как препятствует накоплению кода, делающего продукт неработоспособным. Кроме того, данный принцип требует регулярного выполнения процедур проверки работоспособности продукта (quality assurance).

Следствия:

1. Если изменились бизнес-обстоятельства и заказчик неожиданно потребовал продемонстрировать ему продукт прямо сейчас, разработчики, придерживающиеся данного принципа, с лёгкостью смогут это сделать.

2. Если разработчикам потребовалась дополнительная информация от заказчика, касающаяся разрабатываемой в данный момент функциональности, они могут продемонстрировать продукт в его текущем состоянии и получить релевантные комментарии.

3. Если в какой-то момент продукт окажется «сломан», провести восстановление будет несложно, т. к. последняя «точка работоспособности» отделена от текущего состояния минимальным количеством изменений.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 55–57].

4.5. Выполняйте раннюю и регулярную интеграцию (Integrate early, integrate often). Системная интеграция — это процесс, в ходе которого отдельные, разработанные независимо друг от друга компоненты соединяются в единое приложение. В силу различных причин системная интеграция часто проходит не настолько просто и безболезненно, как того ожидают разработчики, вскрывая множество неожиданных проблем в разрабатываемом программном обеспечении. Для предупреждения этих проблем рассматриваемый принцип призывает, во-первых, начинать интеграцию как можно раньше, не дожидаясь последней фазы проекта, а во-вторых, повторять интеграцию возможно чаще, чтобы убедиться, что внесённые изменения не привели к проблемам в функционировании продукта в целом. Инструментом, гарантирующим успешность интеграции, являются интеграционные тесты, которые проверяют корректность функционирования компонентов в комплексе.

Назначение. Принцип направлен на минимизацию рисков, связанных с ситуациями, когда разработанные компоненты по отдельности функционируют корректно, но при этом работают неправильно или вообще не могут работать вместе в составе продукта.

Следствия:

1. Частые интеграции препятствуют изоляции разработчиков рамками их собственного кода, что повышает эффективность совместной работы.

2. Общее время, затраченное на интеграцию, сокращается, так как мелкие трудности интеграции выявляются и преодолеваются на ранних этапах, а большие и серьёзные проблемы просто не возникают.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 58–60]; [10, глава 15].

4.6. Автоматизируйте развёртывание возможно раньше (Automate deployment early). Современное программное обеспечение устроено достаточно сложно, поэтому, как правило, требуется выполнение процедуры развёртывания, чтобы оно могло правильно функционировать на целевой системе (сервере, настольном компьютере, мобильном устройстве и т. д.). Развёртывание может включать в себя целый ряд действий: копирование исполняемых файлов и файлов ресурсов, установку виртуальных машин, библиотек, регистрацию компонентов в системе и многое другое. Выполнение всех этих действий вручную раз за разом чревато дополнительными временны́ми издержками и ошибками. Последние могут быть как чисто механическими (забыли скопировать нужный файл), так и более сложными, обусловленными недостаточным контролем за средой развёртывания (не учли, что в целевой системе установлена другая версия необходимой библиотеки, не обновлена операционная система и т. п.).

Автоматизация развёртывания посредством написания необходимых скриптов, инсталляторов, использования инструментов управления конфигурацией позволяет решить все перечисленные проблемы. Если развёртывание автоматизировано, то можно быть уверенным, что все необходимые действия, обеспечивающие корректное функционирование продукта на целевой системе, будут выполнены, а сама среда выполнения будет воспроизводимой.

Назначение. Принцип позволяет минимизировать затраты на развёртывание приложения и устранить ошибки, связанные с человеческим фактором и возможной вариативностью сред, в которых происходит развёртывание.

Следствие. Поскольку автоматическое развёртывание продукта требует минимума затрат, появляется возможность тестировать продукт в среде, максимально приближённой к среде эксплуатации. Это позволяет выявить и исправить ошибки, которые в обычных условиях не проявляются на компьютерах разработчиков, так как окружение разработки всегда отличается от среды эксплуатации.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 61–63]; [12, глава 3].

4.7. Получайте обратную связь на основе демонстраций (Get frequent feedback using demos). В ходе выполнения проекта требования заказчика всегда изменяются. Эта изменчивость обусловлена физической невозможностью проработать все детали сложного проекта до момента начала реализации, непостоянством бизнес-условий и среды эксплуатации будущего продукта, а также эволюцией восприятия заказчиком продукта за время его разработки.

Для того чтобы эффективно реагировать на все эти изменения, гибкая методология разработки предлагает использовать демонстрации, во время которых разработчики показывают заказчику частично разработанный продукт и получают от него комментарии, включающие оценку соответствия выполненной работы ожиданиям заказчика, а также информацию об изменениях, которые необходимо внести в разрабатываемый продукт относительно его текущего состояния.

Назначение. Принцип позволяет быстро обрабатывать изменения в требованиях заказчика посредством частого получения оценок уже выполненных частей работы. Тем самым минимизируется риск создания нерелевантного продукта.

Следствие. Минимизируется вероятность ошибок, вызванных неправильным или неточным пониманием требований заказчика: если ошибка всё-таки допущена, она будет замечена заказчиком во время ближайшей демонстрации и исправлена на очередной итерации.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 64–68]; [6, § 10].

4.8. Используйте короткие итерации, выпускайте приращениями (Use short iterations, release in increments). Процесс разработки с применением гибкой методологии является итеративным, и на каждой итерации выполняются проектирование, кодирование, тестирование и системная интеграция. Тем самым достигается поэтапная разработка функций разрабатываемого продукта. Для того чтобы обеспечить эффективное управление изменениями, длительность итерации должна быть небольшой — обычно порядка 1–3 недель. По окончании итерации разработчики обычно могут провести демонстрацию продукта заказчику (см. п. 4.7), однако далеко не всегда готовы выполнить его поставку для эксплуатации конечным пользователям.

Рассматриваемый принцип рекомендует планировать выпуск версий продукта по окончании «больших итераций» или приращений (increment) длительностью от 1 до 6 месяцев (в зависимости от особенностей продукта). Такой период обычно оказывается достаточным для того, чтобы разработать некоторый законченный блок функциональности, представляющий самостоятельную ценность для пользователей. Разумеется, чтобы обеспечить выполнение данного принципа, необходимо заранее спланировать, какая ключевая функциональность будет доставлена в течение приращения.

Назначение. Принцип задаёт ритм разработки продукта как на уровне отдельных функций (итерации), так и на уровне больших функциональных блоков, поэтапно доставляемых заказчику (приращения).

Следствия:

1. Применение данного принципа позволяет получить обратную связь непосредственно от пользователей, эксплуатирующих продукт в реальных условиях до окончания проекта.

2. Поставка промежуточной версии заказчику часто является эффективным средством, позволяющим добиться продолжения финансирования проекта.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 69–72]; [10, глава 9].

4.9. Контракты с фиксированной стоимостью — источник разочарований пользователей (Fixed prices are broken promises). Контракты, стоимость которых определена заранее, трудно реализовывать в рамках гибкой методологии, поскольку такие контракты требуют точных оценок ресурсов и бюджета проекта до начала процесса разработки. Гибкая методология не располагает средствами для таких оценок, поскольку она больше нацелена на организацию разработки в условиях постоянных изменений с присущими им многократными коррекциями первоначальных планов. В результате оказывается, что гибкая методология способна доставить либо продукт в полном объёме, но без фиксации сроков и бюджета, либо часть функционала (причём заранее неизвестного объёма) в пределах фиксированных сроков и бюджета.

Однако следует иметь в виду, что другие подходы к разработке (например, каскадная модель), вынуждая выполнять планирование проекта сразу и целиком, не предоставляют средств управления изменениями требований, что может привести к разочарованию пользователей, кото-

рым не принесёт никакой пользы соблюдение сроков и бюджета, так как в конечном счёте продукт окажется нерелевантным.

Назначение. Принцип предупреждает об опасностях реализации контрактов с фиксированной стоимостью средствами гибкой методологии разработки. Тем самым он очерчивает границы применимости данной методологии.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 73–75]; [10, глава 5].

4.10. Вопросы и упражнения для семинара

1. По каким признакам разработчик должен определить, что для принятия решения по некоторому вопросу он должен обратиться к заказчику? Какими навыками должен обладать разработчик для того, чтобы эффективно применять принцип «позвольте заказчику принимать решения»?
2. Почему в большинстве случаев желательно, чтобы разработчик не только делегировал заказчику принятие бизнес-решения, но также предлагал различные варианты таких решений?
3. Какие виды архитектурных артефактов и какие инструменты могут быть полезны для разработки в рамках гибкой методологии?
4. Определите критерии, которыми следует руководствоваться при выборе технологий и инструментов для проектов разработки ПО разных типов (например, мобильные приложения, веб-приложения, информационные системы уровня предприятия, системы реального времени и т. д.).
5. Рассмотрите несколько современных языков программирования, фреймворков, технологий и сформулируйте условия их применимости в разработке программного обеспечения.
6. Раскройте типичные причины возникновения неожиданных проблем при выполнении системной интеграции.
7. Ознакомьтесь с возможностями, предоставляемыми серверами непрерывной интеграции. Каким образом такие серверы могут быть использованы для воплощения в жизнь принципа «выполняйте раннюю и регулярную интеграцию»?
8. Подготовьте детальный обзор инструментов, которые могут применяться для автоматизации развёртывания, а также их возможностей.
9. Какие требования предъявляются к промежуточной версии продукта, показываемой во время демонстраций для получения обратной связи?

10. Представляют ли изменения, выявляемые в результате каждой демонстрации, угрозу для первоначальных планов доставки функциональности в результате приращений?

5. Принципы гибкой обратной связи

5.1. Пишите тесты (Put angels on your shoulders). В ходе процесса разработки программный код проекта постоянно изменяется. При этом всегда существует опасность регрессии — ошибочного поведения кода, который до изменений работал правильно. Данная опасность является весьма серьёзной, т. к. по мере увеличения размера кодовой базы регрессии могут накапливаться, а замечать и исправлять их становится всё сложнее. Автоматические тесты являются тем самым инструментом, который позволяет не только убедиться в том, что код работает правильно сейчас, но и гарантировать обнаружение регрессий спустя минимальное время после их появления.

Назначение. Принцип направлен на предотвращение накопления регрессий в кодовой базе разрабатываемого проекта и, как следствие, на минимизацию трудозатрат на корректирующее сопровождение.

Следствия:

1. Тесты предоставляют разработчику важную обратную связь, убеждающую его в правильности или предупреждающую о неправильности решения каждой задачи.

2. Благодаря раннему обнаружению возникающие регрессии немедленно исправляются, поскольку разработчик точно знает, какие изменения повлекли эти регрессии, и ещё помнит контекст этих изменений.

3. Если ошибка была обнаружена в готовом продукте, добавление теста, демонстрирующего проявление этой ошибки, в общую базу тестов гарантирует, что после исправления та же самая ошибка на стороне пользователя больше не возникнет.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 78–81]; [6, § 43]; [11, главы 1–3].

5.2. Практикуйте разработку, управляемую тестированием (Use it before you build it). Разработка, управляемая тестированием (test-driven development, TDD), — это техника, предписывающая разработчику сначала написать тесты, проверяющие правильность реализации кода, который он собирается написать, и только после этого уже разработать сам этот код. При этом разработчик должен представить себе, что целевой код уже написан. Интерфейс, предоставляемый этим «воображаемым» кодом, разрабатывается в процессе написания теста, а реализация целевого кода — после него.

Разумеется, до реализации целевого кода тест не будет выполняться успешно, а факт успешного выполнения теста будет признаком правильной реализации целевого кода. На практике после выполнения цикла TDD дополнительно проводится рефакторинг, повышающий качество реализации без внесения функциональных изменений.

Назначение. Принцип направлен на улучшение качества интерфейсов между компонентами приложения (на всех уровнях абстракции) за счёт чёткого отделения этапа проектирования интерфейса компонента от этапа его реализации, что побуждает заранее продумывать правильный способ использования этого интерфейса.

Следствия:

1. Разработка, управляемая тестированием, предоставляет способ обработки требований пользователя: на каждом шаге разработчик должен выделить одно требование, написать тест, формализующий требование, и код, который будет это требование удовлетворять.

2. Рассматриваемый принцип стимулирует разработчиков искать максимально простые решения, а не вносить ненужные усложнения в код и архитектуру проекта.

3. Если в требованиях присутствуют внутренние ошибки или неоднозначности, их становится легче найти, т. к. трудно разработать тест на основе противоречивых или туманных требований.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 82–86]; [11, глава 4].

5.3. Принимайте во внимание отличия в среде выполнения (Different makes a difference). Окружение, в котором выполняется приложение, оказывает существенное влияние на его работу. Во многих случаях невозможно принять во внимание все детали (отличия в версиях операционной системы, типе и версии веб-браузера, системных и пользовательских библиотеках, установленном стороннем программном обеспечении и т. д.), которые могут привести к тому, что нормально функционирующее на одной машине приложение не будет работать на другой.

Для предупреждения проблем такого рода следует тестировать приложение в возможно большем количестве сред выполнения. Чтобы сделать этот процесс надёжным и эффективным, лучшим решением является запуск автоматических тестов в различных средах с помощью сервера непрерывной интеграции. Для имитации различных сред выполнения полезным средством являются виртуальные машины: с их помо-

щью можно создать множество различных конфигураций системы, а затем выполнить тесты во всех созданных окружениях. При этом, помимо снижения затрат на необходимое оборудование, можно легко обеспечить воспроизводимость среды выполнения в каждом из тестов за счёт сохранения слепков (snapshot) состояния виртуальной машины.

Назначение. Принцип направлен на снижение рисков, связанных с влиянием различий среды выполнения на работу приложений.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 87–89].

5.4. Автоматизируйте приёмочное тестирование (Automate acceptance testing). Процедура приёмочного тестирования позволяет заказчику убедиться, что разработанный продукт соответствует его ожиданиям. Разумеется, в общем случае выполнение приёмочного тестирования без заказчика неосуществимо, однако в некоторых случаях можно эффективно автоматизировать приёмочное тестирование критичной бизнес-логики, сделав участие заказчика опосредованным.

Возможный вариант такой автоматизации может выглядеть следующим образом: заказчик подготавливает файлы с исходными данными и результатами обработки этих данных в удобном для него формате (текстовый файл без форматирования, электронная таблица MS Excel и т. п.), а разработчик организует тесты таким образом, чтобы они осуществляли чтение предоставленных файлов и сравнение результатов работы модулей приложения с предоставленными «ответами».

Можно пойти ещё дальше и использовать специальные инструменты (например, Cucumber), которые позволяют заказчику описывать желаемое поведение системы сценариями использования на естественном языке. При этом, однако, следует чётко сопоставлять преимущества с возможными затратами, поскольку накладные расходы при применении подобных инструментов могут быть весьма высокими.

Назначение. Принцип направлен на повышение надёжности реализации критичной бизнес-логики за счёт непосредственного вовлечения заказчика в процесс её разработки и тестирования.

Следствия:

1. Когда заказчик самостоятельно подготавливает примеры исходных данных и результатов их обработки, он тем самым гораздо более точно и детально формулирует задачу для разработчиков, одновременно упрощая и, возможно, ускоряя их работу.

2. Поскольку тесты могут быть предоставлены заказчиком до начала разработки соответствующих модулей, сама разработка может осуществляться в стиле TDD (см. п. 5.2).

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 90–92]; [12, глава 8].

5.5. Используйте правильные метрики (Measure real progress). Отслеживание прогресса проекта играет важнейшую роль в управлении процессом разработки. Однако поскольку при использовании гибкой методологии план проекта постоянно изменяется, а оценки временных затрат всё время уточняются, выбор правильных метрик может быть не самой простой задачей.

Одним из наиболее распространённых инструментов является бэклог (backlog или master story list) — приоритезированный список крупных, ещё не выполненных задач проекта. Перед началом каждой итерации из верхней части списка выбираются задачи, которые предположительно возможно решить за данную итерацию. После выполнения задачи, она вычёркивается из списка. Если требования изменяются или дополняются, соответствующие изменения вносятся в бэклог.

Диаграммой сгорания задач (burndown chart) называется диаграмма, показывающая изменение количества оставшихся задач во времени. Такая диаграмма позволяет оценивать скорость разработки и прогнозировать объём работ, который можно выполнить на очередной итерации, а также (хотя и весьма приблизительно) сроки окончания разработки.

Следует также отметить, что, хотя затраченное на решение задач время и является важной метрикой для планирования, оно не подходит для оценки прогресса. Так происходит из-за возникающих рисков процесса разработки, в результате которых время решения конкретно взятой задачи может измениться значительно. Бэклог менее подвержен влиянию подобных рисков за счёт некоторого усреднения отличий между планируемым и затраченным временем по разным задачам.

Назначение. Принцип предлагает подход к отслеживанию прогресса проекта и указывает на необходимость в вырабатывании собственных метрик, которые будут эффективны при управлении проектом в условиях постоянных изменений требований.

Следствие. Бэклог можно показывать заказчику, чтобы совместно выработать план очередной итерации или предоставить возможность быстро оценить прогресс проекта.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 93–95]; [10, глава 8].

5.6. Прислушивайтесь к пользователям (Listen to users). Гибкая методика разработки подразумевает достаточно тесное взаимодействие с пользователями. К их замечаниям, комментариям и жалобам необходимо прислушиваться, поскольку за счёт этого гибкая методология приобретает конкурентное преимущество над более традиционными подходами. Иногда разработчики склонны считать пользователей назойливыми или глупыми, однако это обычно не так, и чаще всего проблема состоит либо в неумении или нежелании приходить к взаимопониманию при общении, либо в скрытых ошибках, допущенных разработчиками.

Нужно понимать, что пользователь видит продукт не так, как его видят разработчики, и потому в принципе не может использовать их терминологию во время взаимодействия. В то же самое время именно пользователь знает, какие возможности он хочет получить от продукта или что мешает ему использовать продукт для решения его собственных задач. Следовательно, разработчик должен научиться эффективно взаимодействовать с пользователем.

Иногда пользователи систематически делают ошибки при работе с продуктом. Это может свидетельствовать о том, что продукт слишком сложен для использования или что пользовательский интерфейс провоцирует пользователей делать эти ошибки. В каждом из указанных случаев именно разработчик должен понять и устранить проблему.

Назначение. Принцип направлен на установление правильных взаимоотношений с пользователями для получения самой ценной разновидности обратной связи, позволяющей понять, как продукт выглядит с точки зрения тех, для кого он разрабатывается.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 96–97]; [6, § 45].

5.7. Вопросы и упражнения для семинара

1. Почему неавтоматические тесты не позволяют следовать принципу «пишите тесты» в полном объёме?
2. Идентифицируйте все возможные роли, которые могут играть автоматические тесты в проекте.
3. Исследуйте, в каких случаях рационально применять разработку, управляемую тестированием, а в каких — писать тесты после написания кода.

4. Каким образом применение техники TDD может оказать влияние на архитектуру разрабатываемой системы? Упрощает ли разработка, управляемая тестированием, выработку архитектуры системы или усложняет её?
5. Для разных типов приложений (настольные, мобильные, веб-приложения и т. д.) определите возможные отличия сред выполнения, которые надо принимать во внимание во время разработки.
6. Ознакомьтесь со специализированными инструментами автоматизации приёмочного тестирования. В каких случаях их рационально применять?
7. Сравните бэклог с планом реализации проекта, разрабатываемого в рамках каскадной модели. В чём сходства и различия между этими двумя инструментами?
8. Как, используя диаграмму сгорания задач, можно строить планы и прогнозы относительно проекта?
9. Какие риски могут оказывать существенное влияние на время решения задач разработки? Предложите меры по управлению этими рисками.
10. Вспомните «типичную» ошибку, которую Вы регулярно совершаете при работе с каким-либо приложением. Обусловлена ли она ошибкой разработчика? Можете ли Вы предложить лучшее решение?
11. Тесное взаимодействие с заказчиком может провоцировать его изменять требования к разрабатываемому продукту слишком часто. Предложите способы решения данной проблемы.

6. Принципы гибкого кодирования

6.1. Программируйте осознанно и выразительно (Program intently and expressively). Сложный код вызывает затруднения при понимании и проблемы в сопровождении, а также часто содержит скрытые ошибки. Вот почему разработчики всегда должны стремиться писать простой и выразительный код. В ходе разработки программный код читается во много раз чаще, чем пишется и исправляется, поэтому его удобочитаемость всегда должна быть приоритетом для программиста. Как правило, простота и выразительность не достигаются без дополнительных затрат: для получения легкочитаемого и легкосопровождаемого кода необходимо предпринимать дополнительные усилия во время его написания, но результат обычно оправдывает себя за счёт существенного сокращения времени на изучение и поддержку уже написанного кода.

Приёмам работы с кодом, направленным на его упрощение и повышение выразительности, посвящено множество книг. Перечислим здесь некоторые из рассматриваемых в них приёмов: использование осмысленных имён переменных, классов, методов, точно выражающих намерение программиста; правильное форматирование и комментирование исходных текстов; использование коротких функций, каждая из которых решает свою отдельную задачу; использование языков программирования, технологий и библиотек, обладающих внутренней выразительностью встроенных в них конструкций и т. д.

Назначение. Принцип направлен на снижение временны́х затрат во время каждодневной работы программистов с программным кодом, а также на снижение количества возможных ошибок в разрабатываемом коде.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 100–104]; [13].

6.2. Взаимодействуйте с коллегами посредством кода (Communicate in code). Документация играет важную роль при разработке программного обеспечения в рамках традиционной методологии, но при использовании гибкой методологии её ценность снижается. Основные причины — более высокая интенсивность разработки и частое изменение требований. В результате документация быстро устаревает, становится нерелевантной, а поддержание её в актуальном состоянии требует слишком больших затрат. В связи с этим гибкая методология предлагает другой подход к документированию: программный код должен быть на-

столько прост и выразителен, чтобы по возможности не требовать комментариев и дополнительной документации (самодокументированный код).

Стандартная практика предполагает, что непосредственно в коде есть минимальные комментарии на уровне классов, интерфейсов, полей и методов, позволяющие легко понять программисту, как следует использовать те или иные элементы программы. Главный акцент подобных комментариев направлен не на то, *что* делает данный код, а на то, *как* и *зачем* он это делает, а также на возможные ограничения. Для комментариев подобного рода обычно используются такие инструменты, как Doxygen и Javadoc. В свою очередь, реализации методов не содержат комментариев, поскольку понятны и без них. Последнее достигается в основном за счёт удачного выбора имён элементов приложения (классов, полей, методов и т. д.), а также удачного распределения кода по методам (методы короткие и хорошо сфокусированные на решении отдельных мелких задач).

Назначение. Принцип направлен на оптимизацию затрат на создание и поддержание документации, касающейся программного кода. Самодокументированный код позволяет избежать лишних затрат в процессе разработки, в то же самое время повышая сопровождаемость продукта.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 105–109]; [6, § 44].

6.3. Осознанно ищите компромиссы (Actively evaluate trade-offs). Все возможные качества продукта невозможно максимизировать в рамках одного приложения, поскольку многие из них несовместимы друг с другом. Например, повышение производительности обычно приводит к снижению понятности кода, улучшение безопасности — к снижению удобства, а постоянная работа над повышением качества может поставить под угрозу своевременность доставки продукта заказчику. В ходе создания продукта разработчик постоянно сталкивается с необходимостью нахождения компромисса в реализации тех или иных качеств и потому должен уметь оценивать последствия собственных решений конкретных задач разработки для продукта в целом.

Умение находить компромисс составляет достаточно сложный навык разработчика, т. к. требует от него видеть проект одновременно на нескольких уровнях абстракции и с разных точек зрения. Неопытные разработчики склонны пропускать шаг оценки различных возможностей, зачастую неосознанно выбирая первое попавшееся решение, что

может быть весьма губительным для проекта. В некоторых случаях разработчик не может правильно найти компромиссный вариант, поскольку просто не знает, какие качества продукта важны для заказчика. В этом случае он должен на шаге оценки возможностей предложить имеющиеся варианты заказчику и позволить ему принять решение (см. также п. 4.1).

Назначение. Принцип нацеливает разработчиков на комплексное рассмотрение качеств продукта, позволяющее делать осознанный выбор в пользу того или иного решения задач разработки в зависимости от важности достигаемых в каждом случае результатов с точки зрения продукта в целом и его заказчика.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 110–112]; [6, § 4].

6.4. Пишите код короткими сеансами (Code in increments). Написание кода требует сосредоточенности, аккуратности и серьёзных умственных усилий. Для того чтобы поддерживать высокую продуктивность и избежать снижения качества во время работы в течение длительных промежутков времени, необходимо правильно организовать процесс написания кода. Обычно оптимальным оказывается кодирование короткими сеансами, в течение каждого из которых разработчик решает одну маленькую задачу. По окончании её решения необходимо сделать короткий перерыв перед тем, как переходить к следующей задаче, или вообще сменить вид деятельности (например, перейти от кодирования к проектированию, тестированию или рефакторингу).

Описанный стиль разработки предполагает, что решаемые задачи по реализации функциональности разбиваются на мелкие подзадачи, т. к., если этого не сделать, их невозможно будет решать короткими сеансами. Использование разработки, управляемой тестированием (п. 5.2), также помогает следовать данному принципу, т. к. в этом случае весь процесс естественным образом разбивается на короткие сеансы написания теста, реализации функциональности и рефакторинга.

Назначение. Принцип направлен на организацию процесса кодирования, обеспечивая экономное расходование сил разработчиков и помогая избегать ошибок, вызванных утомлением и длительной фиксацией на одних и тех же проблемах программного кода.

Следствие. Благодаря коротким сеансам и постоянной смене видов деятельности, разработчикам удаётся избежать утраты перспективы, т. к. в промежутках между сеансами кодирования появляется возможность рассмотреть проект на более высоком уровне абстракции.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 113–114].

6.5. Предпочитайте простые решения (Keep it simple). Современное программное обеспечение всегда является сложным. Вероятно, бóльшая часть принципов, методологий, приёмов и инструментальных средств индустрии ПО направлены именно на управление этой сложностью, чтобы сделать реализацию проектов возможной. В то же самое время программисты иногда предлагают переусложнённые решения задач разработки, тем самым снижая эффективность всего процесса. Причины данного явления могут быть самыми разными. В их число входят, например, желание применить недавно изученный шаблон проектирования, попытка учесть будущие изменения и недостаточное владение используемым языком программирования или фреймворком.

Рассматриваемый принцип рекомендует программистам искать простые и элегантные решения возникающих перед ними задач разработки. Поскольку гибкая методология содержит в своём арсенале средства управления изменениями, то даже в случае непредусмотренных изменений требований простой код можно будет переработать, чтобы их учесть. Сложный код несёт в себе риски: изменения могут не потребоваться (и тогда затраты на создание переусложнённой архитектуры не окупятся) или потребоваться, но совсем не те, которые были учтены (и тогда вносить изменения в переусложнённый код будет очень сложно и долго).

Следует также отметить, что к простым решениям часто бывает сложнее прийти, чем к сложным, т. к. они не «лежат на поверхности». В связи с этим полезно выполнять ревизию уже написанного кода на предмет возможности его улучшения и упрощения. Бывает, что требуется несколько итераций, чтобы найти элегантное решение.

Назначение. Принцип направлен на улучшение сопровождаемости программного кода и его продуктивности, т. к. призывает разработчика фокусироваться на решении текущих, а не вымышленных проблем.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 115–116]; [14, глава 7].

6.6. Пишите связный код (Write cohesive code). Под связностью (cohesion) в программировании понимается мера того, насколько элементы одного и того же класса, модуля, пакета направлены на предоставление некоторого целостного функционала или на решение одной и

той же задачи. Высокая связность упрощает использование программного кода и его сопровождение. Низкая связность затрудняет понимание кода, поскольку взаимодействующие элементы программы оказываются разбросаны по разным частям проекта, а также усложняет внесение изменений и повышает вероятность появления ошибок, поскольку приходится вносить множество разнородных исправлений для изменения всего лишь одной части функционала.

Для поддержания связного кода в программировании существует множество инструментов. Примерами могут служить принципы абстракции и инкапсуляции, направленные на объединение кода, соответствующего поведению однородных объектов в один класс, и выделение необходимого и достаточного интерфейса взаимодействия этого класса с внешним миром, а также многие архитектурные шаблоны (например, «модель—вид—контроллер»), предлагающие способы разделения логики приложения по функциональным слоям.

Назначение. Принцип направлен на облегчение использования программных компонентов за счёт правильной группировки и распределения ответственности, а также на улучшение сопровождаемости кода.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 117–120]; [6, § 29][2].

6.7. Предпочитайте методы-команды методам-запросам (Tell, don't ask). Объектно-ориентированное программирование направлено на передачу объектам полномочий по управлению своими данными. При этом для внешнего мира объекты предоставляют методы, с помощью которых программист «приказывает» объектам выполнить то или иное действие. Однако иногда в программном коде встречаются ситуации, когда некоторый объект получает данные, хранимые внутри некоторого другого объекта, и на основании этих данных производит вычисления или принимает решение. Несмотря на то что формально такой код остаётся объектно-ориентированным, он нарушает принцип инкапсуляции, т. к. внутренняя логика классов, хранящих данные, по существу размещается не внутри этих классов, а разбрасывается по классам, выступающим по отношению к данному классу в роли клиентов.

[2]Обратите внимание, что авторы перевода книги [6] используют термин «связанность» в смысле «coupling», а не «cohesion»! Это может вызывать непонимание, т. к. применительно к коду смысл этих двух понятий практически противоположен.

Чтобы следовать данному принципу, необходимо отделять методы-запросы (queries), позволяющие получать данные из классов и не меняющие состояние объектов, от методов-команд (commands), позволяющих изменять состояние объектов. И если в какой-то части кода наблюдается преобладание вызовов методов-запросов, то это служит признаком, что рассматриваемый принцип нарушается и что соответствующую логику скорее всего необходимо перенести в тот класс, на объектах которого вызываются эти методы-запросы. Для облегчения идентификации таких ситуаций рекомендуется использовать специальные соглашения об именах (например, добавлять к именам методов-запросов префикс *get*).

Назначение. Принцип направлен на улучшение сопровождаемости кода за счёт правильного распределения ответственности между компонентами и сокрытия реализации функционала до того момента, когда программисту потребуется изменять именно эту реализацию функционала.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 121–123].

6.8. Не нарушайте семантику интерфейса при наследовании (Substitute by contract). В объектно-ориентированном программировании наследование позволяет повторно использовать код классов путём создания на их основе более частных и специализированных версий. При этом принцип полиморфизма постулирует, что взаимодействие классов определяется только их интерфейсом, следовательно экземпляры классов-предков могут быть прозрачно заменены экземплярами их наследников. Данный механизм является мощным и выразительным инструментом разработки. Однако следует понимать, что его работоспособность целиком и полностью основывается на допущении, что классы-наследники всегда соблюдают семантику интерфейса, определённого классами-предками.

Нарушение данного допущения приводит к чрезвычайно тяжёлым патологиям в коде и архитектуре проекта, которые выражаются прежде всего в невозможности гарантировать, что вызов того или иного метода действительно выполнит действие, соответствующее его названию (т. к. семантика этого метода в наследнике могла измениться несогласованным образом), а также в нелогичности организации самих иерархий наследования (поскольку классы-наследники могут уже не являться частными, специализированными версиями классов-предков).

Назначение. Принцип ориентирован на предупреждение неправильного применения механизма наследования при разработке объектно-ориентированного кода.

Следствие. Если разработчик не сможет гарантировать выполнение данного принципа к разрабатываемой иерархии классов, он откажется от наследования и применит вместо него делегирование.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 124–127].

6.9. Вопросы и упражнения для семинара

1. Выберите и сравните несколько языков программирования на предмет их внутренней выразительности. Оцените, насколько существенным может быть влияние выбора языка программирования на удобочитаемость и сложность программного кода.
2. Рассмотрите 10-15 приёмов повышения удобочитаемости и выразительности кода, описанных в литературе. Определите, как конкретно влияет применение данных приёмов на удобочитаемость и выразительность.
3. Найдите фрагмент собственного кода, непонятного или с трудом понятного без дополнительной документации. С помощью серии рефакторингов сделайте этот код самодокументированным.
4. Какие качества наиболее важны для продуктов следующих типов: настольное приложение, мобильное приложение, веб-приложение, система реального времени, корпоративная информационная система?
5. Известно, что для решения многих задач разработки необходимо «погружение» в предметную область и детали реализации системы. Не вступает ли необходимость такого «погружения» в противоречие с принципом «пишите код короткими сеансами»?
6. Помимо переусложнённых решений задач разработки встречаются также переупрощённые. Исследуйте, где проходит грань между простыми решениями и переупрощёнными. Приведите примеры.
7. Каким образом метод карт CRC помогает следовать принципу «пишите связный код»?
8. Рассмотрите преимущества и недостатки методов-команд, имеющих возвращаемые значения. Приведите примеры.
9. Приведите примеры нарушения семантики интерфейсов при наследовании, используя примеры из собственного кода и стандартных библиотек. Покажите, к каким конкретным патологиям в коде приводят данные нарушения.

7. Гибкая отладка приложений

7.1. Записывайте принимаемые решения (Keep a solutions log). При создании проекта любой сложности встречаются однотипные задачи. Когда разработчик впервые решает нетривиальную задачу, он находит материалы, описывающие подходы к её решению, и на основании их делает выбор в пользу того или иного варианта. Для сокращения времени поиска решения аналогичных задач следует зафиксировать результаты этого выбора в общей базе знаний. В описание решения могут входить следующие элементы: дата; краткая характеристика решённой задачи; расширенное описание решения с обоснованием; ссылки на статьи или интернет-ресурсы, использованные при поиске решения; любые дополнительные материалы, способствующие разъяснению решения.

Назначение. Принцип способствует накоплению знаний команды об используемых в проекте инструментах и предметной области в удобной для поиска форме. В результате разработчики быстрее реализуют новую функциональность в проекте.

Следствие. Если разработчики делятся решениями проблем друг с другом, то общая эффективность команды возрастает. При поиске решений база знаний становится эффективным средством поиска качественной информации для решения проблем.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 129–131].

7.2. Предупреждения компилятора указывают на ошибки (Warning are really errors). При написании исходного кода приложения разработчики допускают синтаксические и логические ошибки. Первый тип ошибок не позволяет компилятору создать исполняемый файл, поэтому эти ошибки быстро исправляются. Некоторые логические ошибки также могут быть обнаружены компилятором — для них выводятся предупреждения. К сожалению, многие разработчики считают предупреждения несерьёзными и часто игнорируют их или даже просто отключают. Однако это заблуждение: большинство предупреждений свидетельствуют о серьёзных проблемах в программах или являются признаком возникновения таких проблем в будущем. В связи с этим разработчики должны исправлять дефекты, о которых сообщают исключения, наряду с выявленными синтаксическими ошибками.

Дополнительным источником информации о скрытых проблемах являются статические анализаторы кода. Их основная задача состоит в вы-

явлении устаревших API, распространённых антипаттернов реализации (bad practices) и других фрагментов неудачного кода. Команда разработчиков может определить необходимый набор проверок, которым должен подвергаться исходный код приложения, и исправлять те проблемы, на которые укажут анализаторы.

Назначение. Принцип нацелен на обнаружение и исправление как существующих, так и потенциальных проблем в исходном коде приложения на как можно раннем этапе, что достигается путём делегирования задачи выявления патологий специализированному инструментарию.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 132–135].

7.3. Изолируйте проблему перед решением (Attack problems in isolation). Для обнаружения источника проблемы разработчику следует рассматривать приложение не как единое целое, а как набор небольших модулей, взаимодействующих друг с другом. В этом случае поиск источника проблемы можно достаточно быстро сузить до одного модуля, содержащего ошибку.

Идеальным инструментом для исследования работы модулей являются модульные тесты (см. п. 5.1). При их отсутствии можно прибегнуть к практике создания прототипа, использующего проблемный модуль. Основная задача прототипа — выполнение сценария, демонстрирующего появление ошибки. Прототип позволяет быстрее добиться решения проблемы, так как объём исходного кода значительно меньше, и разработчик не отвлекается на детали работы других подсистем.

Назначение. Принцип направлен на уменьшение трудозатрат при исправлении ошибок в коде. Приложение легче поддерживать, если оно было грамотно разделено на функциональные модули.

Следствие. При появлении проблемы разработчик может оценить объём работ, необходимых для решения проблемы. Разработчик быстрее получает помощь от коллег, так как проблемная ситуация описывается небольшим сценарием, реализация которого опирается на минимально возможное число зависимостей.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 136–138]; [6, § 11].

7.4. Оповещайте об исключительных ситуациях (Report all exceptions). Во время работы приложения могут возникнуть различные исключительные ситуации: нужный файл был удалён, сетевое соединение

разорвано и т. д. Исключение должно быть обработано в методе, в котором оно было зарегистрировано, если достаточно данных для восстановления состояния системы после ошибки. В противном случае о данной ситуации должен быть оповещён вызывающий метод. Эта цепочка оповещений должна прекратиться либо автоматическим разрешением ситуации, либо информированием пользователя.

Назначение. Принцип направлен на реализацию грамотной политики по обработке исключительных ситуаций, что способствует созданию более надёжного и устойчивого приложения.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 139–140]; [6, § 22].

7.5. Предоставляйте содержательные сообщения об ошибках (Provide useful error messages). Готовое приложение поставляется конечным пользователям, после чего разработчику приходится выполнять сопровождение продукта, исправляя ошибки в исходном коде, предоставляя консультации по настройке окружения и работе приложения. При обращении в службу поддержки пользователь описывает своё взаимодействие с приложением, которое привело к ошибочной ситуации. Если текст сообщения об ошибке не конкретен, тогда для решения проблемы потребуется потратить много времени на выяснение множества деталей, касающихся работы приложения.

Содержание сообщения об ошибке должно быть в первую очередь ориентировано на лицо, способное исправить возникшую ситуацию. Если ошибка связана с действиями пользователей, тогда сообщение должно описывать причину ошибки и указывать правильный путь решения задачи. Если ошибка связана с окружением, тогда сообщение должно предоставлять необходимые данные о проблеме для системного администратора, чтобы он смог решить проблему. Если ошибка вызвана дефектами в приложении, тогда сообщение должно как можно более детально указывать место в приложении, где этот дефект проявился.

Назначение. Принцип обеспечивает проработку сценариев работы приложения, учитывающих не только успешное выполнение заданных операций, но также и работу приложения при возникновении ошибочных ситуаций.

Следствие. Компания тратит меньше ресурсов на оказание услуг поддержки, а пользователи в большинстве случаев могут самостоятельно решить проблему исходя из содержания сообщений.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 141–145].

7.6. Вопросы и упражнения для семинара

1. Рассмотрите различные способы организации базы знаний для хранения информации о принимаемых решениях, а также инструменты, упрощающие такую организацию. Для чего может потребоваться выполнение сопровождения базы знаний и как его осуществлять?
2. Отберите 5-10 распространённых предупреждений компиляторов различных языков программирования и объясните, о каких возможных проблемах в коде эти предупреждения сигнализируют.
3. Вспомните из собственной практики случай, когда Вы пренебрегли принципом «изолируйте проблему перед решением», и это повлекло негативные последствия. Предложите шаги, которые надо было предпринять для правильного решения проблемы.
4. Предложите варианты архитектуры, поддерживающие оповещение пользователя об исключительных ситуациях, для различных типов приложений (консольное, настольное приложение с графическим интерфейсом, мобильное, веб-приложение).
5. Какую роль могут играть системные журналы (log) на различных этапах процесса разработки (кодирование, тестирование, отладка, сопровождение)? Для каких целей следует использовать журналы, а для каких нет? Какую информацию имеет смысл выводить в журнал?

8. Принципы гибкого взаимодействия внутри команды

8.1. Выделите время для регулярных совместных обсуждений проекта (Schedule regular face time). Совместные обсуждения проекта являются важнейшим механизмом, позволяющим координировать работу членов команды, а также вырабатывать решения сложных задач разработки. Однако во время таких обсуждений ни один из разработчиков не занимается выполнением своих основных обязанностей, что может приводить к неэффективному расходованию рабочего времени. В связи с этим необходимо выбрать оптимальную длительность и режим обсуждений, чтобы их преимущества не оказались снижены или даже перечёркнуты за счёт накладных расходов.

Гибкая методология рекомендует проводить ежедневные обсуждения, где каждый член команды в течение короткого времени (обычно 2 минуты) отвечает на следующие вопросы: «Что я сделал вчера? Что я собираюсь сделать сегодня? Что мне мешает?» Как правило, ответов оказывается достаточно для информирования всех членов команды о текущей деятельности друг друга, оценки состояния проекта и идентификации возможных проблем. В последнем случае для обсуждения обнаруженных проблем назначаются дополнительные совещания с участием только тех разработчиков, которых они затрагивают.

Совместные обсуждения должны быть регулярными, а не проводиться от случая к случаю, т. к. в противном случае их ценность будет ниже. Для предотвращения чрезмерного затягивания совместных обсуждений можно использовать следующий полезный приём: разрешать разработчикам участвовать в совещании только стоя («stand-up meetings»). В этом случае фактор некоторого физического неудобства заставляет участников быть более краткими в своих выступлениях.

Назначение. Принцип направлен на улучшение взаимодействия членов команды разработчиков при эффективном расходовании их рабочего времени за счёт коротких и хорошо сфокусированных обсуждений.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 148–151].

8.2. Архитекторы должны писать код (Architects must write code). В индустрии программного обеспечения распространено представление об архитекторах как о специалистах, которые лишь проектируют про-

граммный продукт и направляют процесс разработки, осуществляемый другими людьми. К сожалению, при возникновении проблем такие архитекторы ничем не могут помочь разработчикам, т. к. не знают необходимых деталей или к этому моменту они уже в принципе не задействованы в проекте.

Гибкая методология предлагает бороться с проблемами такого рода, отказываясь от роли архитектора, работающего в выделенном режиме, предлагая ему и активно участвовать в разработке. Такой подход оказывается оправданным, потому что архитекторы, являющиеся одновременно разработчиками, в гораздо большей степени способны реагировать на изменения в проекте и потому могут эффективно справляться с недостатками архитектуры, которые не были замечены в начале проекта или появились в ходе его реализации.

Назначение. Принцип направлен на эффективное управление изменениями архитектуры программного продукта в ходе разработки за счёт вовлечения разработчиков, владеющих всеми деталями реализации, в процесс проектирования.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 152–154]; [6, § 9].

8.3. Практикуйте коллективное владение кодом (Practice collective ownership). Одной из самых характерных черт гибкой методологии разработки является коллективное владение кодом — отсутствие границ в возможностях изучения и изменения всей кодовой базы проекта. Коллективное владение означает, что любой член команды может (и даже должен) изменять любой код, если это необходимо для решения его текущих задач, независимо от того, кто этот код написал. Преимущества коллективного владения обусловлены тем, что код постоянно изучается, поддерживается и тестируется различными членами команды, что позволяет оперативнее находить логические и технические ошибки, неудачные решения и архитектурные проблемы.

Менеджеры могут стимулировать коллективное владение, используя ротацию в назначении разработчиков на задачи, локализованные в различных частях кода.

Назначение. Принцип направлен на повышение эффективности работы членов команды разработчиков с кодом, а также на повышение качества самого кода за счёт его многократного изучения и тестирования разными людьми.

Следствия:

1. Коллективное владение кодом способствует распространению знаний между разработчиками и, следовательно, профессиональному совершенствованию всей команды.

2. При коллективном владении проект становится более устойчивым к возможному выбыванию разработчика из команды, т. к. другие разработчики при необходимости смогут его заменить.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 155–156].

8.4. Будьте наставником (Be a mentor). Наставник — это член команды разработчиков, который оказывает помощь (как правило, методического плана) другим разработчикам в решении нетривиальных проблем при создании или поддержке программного обеспечения. Во многих крупных компаниях наставничеством занимаются преимущественно старшие разработчики, но в гибких командах практически любой разработчик может быть наставником, поскольку для осуществления деятельности такого рода требуется лишь минимальное наличие знаний, которыми не обладают другие члены команды, и желание этими знаниями поделиться. Как правило, данный тип взаимодействия разработчиков имеет краткосрочный (иногда циклический) характер и нацелен на анализ проблемы и выработку идеи решения, для реализации которой помощь наставника уже не требуется.

Для того чтобы стимулировать наставничество, может быть установлено следующее правило: если разработчик решает какую-то проблему и в течение некоторого временного лимита (обычно порядка 1.5-2 часов) не находит решение, он должен поискать коллегу, который обладает достаточными знаниями и опытом, чтобы оказать помощь в её решении.

Назначение. Принцип направлен на ускорение решения задач разработки за счёт эффективного использования знаний и опыта нескольких членов команды, а также является инструментом профессионального развития членов такой команды.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 157–159]; [7, § 14].

8.5. Давайте ключ к решению, а не само решение (Allow people to figure it out). Гибкая методология поощряет разработчиков задавать друг другу вопросы, касающиеся как различных аспектов разработки, так и деталей решаемых задач. Более того, она рекомендует не ограни-

чиваться краткими ответами, а обосновывать эти ответы или вообще давать лишь ключ к решению вместо самого решения. Такой подход позволяет спрашивающему получить более полезную информацию, которую можно будет использовать в дальнейшей профессиональной деятельности в аналогичных ситуациях.

Назначение. Принцип нацелен на мотивацию разработчиков искать решения, а не ограничиваться готовыми ответами, что способствует профессиональному совершенствованию всей команды.

Следствия:

1. Получив лишь указания к решению задачи вместо прямого ответа, спрашивающий может найти даже лучшее решение, чем то, которое имел в виду отвечающий.

2. Необходимость постоянно обосновывать предлагаемые решения способствует улучшению коммуникационных навыков разработчиков.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 160–161].

8.6. Публикуйте только готовый код (Share code only when ready). Код, содержащий законченное решение задачи, должен немедленно публиковаться в системе управления версиями. Это позволяет проводить его раннюю интеграцию и получать обратную связь как со стороны автоматических тестов, так и со стороны других разработчиков.

В тех же случаях, когда решение незаконченно, вопрос о целесообразности публикации изменений, внесённых в кодовую базу, становится более сложным. Незаконченное решение может нарушать функциональность других частей разрабатываемого приложения, мешать другим разработчикам, влиять на успешность выполнения тестов и т. д. Если подобные ситуации имеют место, то программист не должен публиковать незавершённые результаты своей работы либо предварительно принять меры, которые предотвратят негативное влияние его изменений на остальную часть проекта (например, вставить заглушки нереализованных интерфейсов).

Однако здесь следует отметить, что разработчик не должен задерживать код в своём локальном репозитории, аргументируя это неготовностью кода к публикации. Такого рода задержка ведёт к изоляции работы программиста и к последующим сложностям при системной интеграции. В действительности ситуация, когда код не готов и потому не может быть опубликован, практически всегда связана с неправильным разбиением задачи на подзадачи, а вовсе не с практической необходимостью.

Назначение. Принцип нацелен на предотвращение неправильного использования системы управления версиями, в результате которого продукт становится неработоспособным из-за того, что незаконченная реализация некоторой функциональности нарушает работу других частей приложения.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 162–164].

8.7. Рецензируйте код (Review code). Рецензирование кода (code review) — это процесс, подразумевающий изучение кода, написанного одним или несколькими разработчиками, другими членами команды. Большинство специалистов в области разработки программного обеспечения считают, что рецензирование кода является одним из самых эффективных способов выявления внутренних проблем в программном коде.

Существует несколько стилей рецензирования кода. Наиболее распространённые подходы подразумевают либо проверку каждого набора изменений, отправляемого в систему управления версиями, разработчиком, отличным от автора этих изменений, либо парное программирование. В последнем случае два разработчика работают за одним компьютером и в то время, когда один из них пишет код, второй немедленно его проверяет, после чего разработчики меняются ролями.

Для эффективного рецензирования кода разработчикам полезно иметь контрольные списки типичных проблем, которые следует искать в коде. Такие списки могут содержать как общие вопросы («Понятно ли, что делает код? Есть ли очевидные ошибки? Есть ли побочные эффекты? Есть ли дублирование кода?»), так и более специфические («Вся ли выделенная память освобождена? Все ли деструкторы объявлены виртуальными?»).

Назначение. Принцип направлен на организацию управления качеством продукта за счёт многократной проверки программного кода разными людьми.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 165–167].

8.8. Держите коллег и заказчика в курсе дел (Keep others informed). Разработке программного обеспечения сопутствует множество рисков. Из-за этих рисков часто случается так, что текущие задачи, которыми занимается разработчик, не могут быть решены в срок. В этом случае

разработчик должен при первой возможности уведомить своих коллег, менеджера и, возможно, заказчика о сложившейся ситуации. Благодаря этому проблема может быть устранена или смягчена на раннем этапе и не приведёт к неприятным последствиям для всего проекта в целом (таким как срыв сроков поставки, неудовлетворённость заказчика, снижение качества, потеря продуктом конкурентоспособности и т. д.).

Для того чтобы эффективно следовать данному принципу, важна своевременность: разработчик должен среагировать при первых признаках проблемы, а не в самом конце срока, отведённого на решение задачи, когда практически никакие меры уже не смогут принести позитивного эффекта. Для обеспечения этой своевременности разработчик должен выработать навык экстраполяции своего прогресса на весь объём работы, регулярно оценивая объём времени, который нужен для завершения решения задачи, и информируя других членов команды немедленно, если экстраполированная величина превышает установленные сроки.

Назначение. Принцип направлен на организацию управления всевозможными рисками разработки программного обеспечения, что позволяет принимать адекватные меры при первых признаках возникновения проблем.

Материал для самостоятельного изучения: подробное изложение принципа [1, с. 168–169]; [6, § 13].

8.9. Вопросы и упражнения для семинара

1. В чём состоит специфика организации взаимодействия в распределённых командах разработчиков по сравнению с командами, сосредоточенными в одном пространстве? Предложите эффективные способы взаимодействия для распределённых команд.
2. Каким требованиям должна удовлетворять первоначальная, выработанная до реализации проекта архитектура, чтобы обеспечить выполнение принципа «Архитекторы должны писать код»? Какие меры должны предпринимать разработчики, изменяя архитектуру проекта во время его разработки, чтобы обеспечить его сопровождаемость?
3. Если команда следует принципу коллективного владения кодом, то для её проектов могут представлять некоторую опасность начинающие и неопытные разработчики, а также разработчики, склонные вносить быстрые непродуманные изменения. В чём могут заключаться эти опасности и как с ними бороться?

4. Выполняя функции наставника, разработчик вынужден отвлекаться от решения собственных задач, что приводит к дополнительным незапланированным временным затратам. Как это может сказываться на общем времени реализации проекта? Рассмотрите различные ситуации.
5. Занимаясь наставничеством, разработчик делится своими знаниями и опытом с другими членами команды, до некоторой степени теряя при этом свою исключительность в команде. Не несёт ли это угрозы для его карьеры?
6. Как принцип «Давайте ключ к решению, а не само решение» позволяет совершенствовать управление внутри команды разработчиков?
7. Раскройте связь между принципом «Публикуйте только готовый код» и другими принципами гибкой разработки.
8. Как механизм ветвей (branches) систем управления версиями помогает следовать принципу «Публикуйте только готовый код»?
9. Сравните автоматическое тестирование, статический анализ и рецензирование кода по возможностям обнаружения ошибок различных типов.
10. Многие разработчики боятся держать своего начальника в курсе возникающих проблем, опасаясь обвинений в некомпетентности. Проанализируйте данную проблему и предложите решения.

Литература

1. Subramaniam, V. Practices of an Agile Developer / Venkat Subramaniam, Andy Hunt. — Pragmatic Bookshelf, 2006.
2. Fowler, M. The agile manifesto / Martin Fowler, Jim Highsmith // Software Development. — 2001. — Vol. 9, no. 8. — P. 28—35.
3. Бек, К. Экстремальное программирование / Кент Бек. — СПб. : Питер, 2002. — 224 с.
4. Кон, М. Scrum. Гибкая разработка ПО / Майк Кон. — М. : Вильямс, 2015. — 576 с.
5. Попендик, М. Бережливое производство программного обеспечения. От идеи до прибыли / Мэри Попендик, Том Попендик. — М. : Вильямс, 2010. — 256 с.
6. Хант, Э. Программист-прагматик. Путь от подмастерья к мастеру / Эндрю Хант, Дэвид Томас. — СПб. : Питер Пресс, 2007. — 289 с.
7. Фаулер, Ч. Программист-фанатик / Чед Фаулер. — СПб. : Питер, 2015. — 208 с.
8. Cirillo, F. The pomodoro technique / Francesco Cirillo. — 2009. — 46 p.
9. Allen, D. Getting things done: The art of stress-free productivity / David Allen. — Penguin, 2002.
10. Rasmusson, J. The Agile Samurai: How Agile Masters Deliver Great Software / Jonathan Rasmusson. — Pragmatic Bookshelf, 2010.
11. Kaczanowski, T. Practical Unit Testing with JUnit and Mockito / Tomek Kaczanowski. — Tomasz Kaczanowski, 2013. — 402 p.
12. Хамбл, Д. Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий программ / Джез Хамбл, Дэвид Фарли. — М. : Вильямс, 2011. — 432 с.
13. Босуэлл, Д. Читаемый код, или Программирование как искусство / Дастин Босуэлл, Тревор Фаучер. — СПб. : Питер, 2012. — 208 с.
14. Kanat-Alexander, M. Code Simplicity / Max Kanat-Alexander. — O'Reilly Media, 2012. — 84 p.

Учебное издание

Парамонов Илья Вячеславович
Васильев Андрей Михайлович

Инженерия программных систем и комплексов на основе гибкой методологии разработки

Учебно-методическое пособие

Редактор, корректор М. Э. Левакова
Компьютерный набор, вёрстка И. В. Парамонова

Подписано в печать 24.04.2015г. Формат 60×84/16.
Усл. печ. л. 2,8. Уч.-изд. л. 2,5.
Тираж 30 экз. Заказ

Оригинал-макет подготовлен в редакционно-издательском отделе
Ярославского государственного университета.

Ярославский государственный университет
150000, Ярославль, ул. Советская, 14.