Методы коллективной работы в проектах на базе свободного программного обеспечения

Грудинин В.А., Сомс Н.Л.

2012

Оглавление

Технологические процессы разработки комплексных программных продуктов. .5
Основные процессы 5
История и эволюция технологии программирования..........8
Основные процессы жизненного цикла программного обеспечения..........10
Приобретение..........10
Поставка..........10
Разработка..........11
Эксплуатация..........12
Сопровождение..........12
Вспомогательные процессы..........13
Документирование..........13
Управление конфигурацией..........13
Обеспечение качества..........14
Верификация..........14
Аудит..........14
Решение проблем..........15
Организационные процессы..........16
Управление..........16
Создание инфраструктуры..........16
Усовершенствование..........16
Обучение..........17
Программные продукты для управления проектами разработки программного обеспечения..........18
Системы управления проектами..........18
Организационные средства..........19
Средства оценки качества..........20
Техника совместной разработки комплексных программных продуктов 21
Стандартные фазы разработки проектов..........21
Метод «кодирование и исправление»..........22
Каскадный метод разработки..........23
Классический каскадный подход..........23
Каскадно-возвратный подход..........26
Каскадно-итерационный подход..........27
Каскадный подход с перекрывающимися процессами..........28
Каскадный подход с подпроцессами..........29
V-model..........31
Спиральная модель разработки..........39
Модель Хаоса..........41
Каркасные технологические подходы..........43
Рациональный унифицированный процесс 43
OpenUP..........43
Итерационный метод разработки..........46

Итерационный подход....46
Эволюционное прототипирование....48
Гибкие методологии разработки....49
Экстремальное программирование....51
SCRUM....52
LEAN....54
Test-Driven Development (TDD)....56
Добавление теста....60
Запуск всех тестов: новые тесты не проходят....60
Написание кода....60
Запуск всех тестов: все тесты проходят....60
Рефакторинг....61
Повтор цикла....61
Принципы....61
Fake-, mock-объекты и интеграционные тесты....64
Feature driven development....66
Разработка общей модели....66
Составление списка возможностей (функций)....66
План по свойствам (функциям)....67
Проектирование функций....67
Реализация функции....67
Технологии коллективной разработки....70
Авторская разработка....70
Коллективная разработка....70
Равноправные исполнители....70
Бригада главного программиста....72
Программирование в парах....72
Ядерная модель....73
Общинная модель разработки....74
Офшорное программирование....75
Обеспечение качества программного продукта....78
Понятие об обеспечении качества и тестировании. Историческое развитие представлений....78
Качество программного обеспечения....78
Характеристики качества программного обеспечения....81
Качество процесса разработки....84
Определение возможностей и улучшение процесса создания программного обеспечения....85
Качество баз данных....86
Тестирование программного обеспечения....97
Цели тестирования....102
Классификация методов тестирования....106
Классификация по форме тестирования....106
Классификация по объекту тестирования....106
Классификация по знанию системы....106
Классификация по степени автоматизации....107

Классификация по степени изолированности компонентов........................107
Классификация по времени проведения тестирования..............................107
Классификация по признаку позитивности сценариев...............................107
Классификация по степени подготовленности к тестированию.................107
Уровни тестирования (юнит-, интеграционное, системное тестирование)…108
Модульное тестирование (Unit testing)..108
Интеграционное тестирование (Integration testing)....................................110
Системное тестирование (System testing)...113
Альфа- и бета-тестирование...113
Другие виды тестирования..115
Статическое тестирование...115
Программное обеспечение для статического анализа............................116
Юзабилити-тестирование..117
Автоматизированное тестирование..121
Результат тестирования — баг..123
Программные продукты для Unit-тестирования...127
Системы регистрирования ошибок..129
Разновидности автоматических тестов..131
Регрессионное тестирование...131
Тестирование безопасности...133
Тестирование на отказ и восстановление...136
Тестирование производительности..138
Нагрузочное тестирование...142
Стресс-тестирование...146
Тестирование стабильности...149
Конфигурационное тестирование...150
Непрерывная интеграция..152
Сборка по расписанию...152
Методология жизненного цикла автоматизированного тестирования...........154
Системы автоматизированного тестирования...163
Литература...166

Технологические процессы разработки комплексных программных продуктов

Основные процессы

Известный эксперт по управлению высокотехнологичными проетами Арчибальд так определяет жизненный цикл проекта [Арчибальд Р., 2003, с.58-59]:

> “Жизненный цикл проекта имеет определенные начальную и конечную точки, привязанные к временной шкале. Проект в своем естественном развитии проходит ряд отдельных фаз.
>
> Жизненный цикл проекта включает все фазы от момента инициации до момента завершения. Переходы от одного этапа к другому редко четко определены, за исключением тех случаев, когда они формально разделяются принятием предложения или получением разрешения на продолжение работы. Однако, в начале концептуальной фазы часто возникают сложности с точным определением момента, когда работу можно уже идентифицировать как проект (в терминах управления проектами), особенно если речь идет о разработке нового продукта или новой услуги.
>
> Существует общее соглашение о выделении четырех обобщенных фаз жизненного цикла (в скобках приведены используемые в различных источниках альтернативные термины):
> - концепция (инициация, идентификация, отбор)
> - определение (анализ)
> - выполнение (практическая реализация или внедрение, производство и развертывание, проектирование или конструирование, сдача в эксплуатацию, инсталляция, тестирование и т.п.)
> - закрытие (завершение, включая оценивание после завершения)
>
> Однако, эти фазы столь широки, что ... необходимы конкретные определения, быть может пяти-десяти основных фаз для каждой категории и подкатегории проекта, обычно с несколькими подфазами, выделяемыми внутри каждой из этих фаз.
>
> Нередко можно наблюдать частичное совмещение или одновременное выполнение фаз проекта, называемое “быстрым проходом” в строительных и инжиниринговых проектах и “параллелизмом” – в военных и аэрокосмических. Это усложняет планирование проекта и координацию усилий его участников, а также делает более важной роль менеджера проектов.”

В общем случае, жизненный цикл определяется моделью и описывается в форме методологии (метода). Модель или парадигма жизненного цикла определяет концептуальный взгляд на организацию жизненного цикла и, часто, основные фазы жизненного цикла и принципы перехода между ними. Методология (метод) задает комплекс работ, их детальное содержание и ролевую ответственность специалистов на всех этапах выбранной модели жизненного цикла, обычно определяет и саму модель, а также рекомендует практики (best practices), позволяющие максимально эффективно воспользоваться соответствующей методологией и ее моделью.

В индустрии программного обеспечения можно (так как это уже конкретная область приложения концепций и практик проектного управления) и необходимо (для обеспечения возможности управления) более четкое разграничение фаз проекта (что не подразумевает их линейного и последовательного выполнения).

На сегодняшний день принято следующее определение — жизненным циклом программного обеспечения (ПО) называют период времени от момента возникновения идеи о создании программного продукта до полного изъятия его из эксплуатации.

Приняты следующие стандарты жизненного цикла программного обеспечения:

1. ГОСТ 34.601-90
2. ISO/IEC 12207:1995 (в РФ аналогом является ГОСТ Р ИСО/МЭК 12207-99)

ГОСТ Р ИСО/МЭК 12207 является переводом международного стандарта ISO/IEC 12207, на основе которого, в свою очередь, создан соответствующий стандарт IEEE 12207. Второй – в рамках семейства ГОСТ 34 – разрабатывался в СССР самостоятельно, как стандарт на содержание и оформление документов на программные системы в рамках Единой системы программной документации (ЕСПД) и Единой системы конструкторской документации (ЕСКД). В последние годы, акцент делается на стандарты ГОСТ, соответствующие международным стандартам. В то же время, 34-я серия является важным дополнительным источником информации для разработки и стандартизации внутрикорпоративных документов и формирования целостного понимания и видения концепций жизненного цикла в области программного обеспечения.

Скотт Амблер (Scott W. Ambler) [Ambler, 2005], автор концепций и практик гибкого моделирования (Agile Modeling) и Enterprise Unified Process (расширение Rational Unified Process), предлагает следующие уровни жизненного цикла, определяемые соответствующим содержанием работ (см. рис.1):

- Жизненный цикл разработки программного обеспечения – проектная деятельность по разработке и развертыванию программных систем
- Жизненный цикл программной системы – включает разработку, развертывание, поддержку и сопровождение
- Жизненный цикл информационных технологий – включает всю деятельность ИТ-департамента
- Жизненный цикл организации/бизнеса – охватывает всю деятельность организации в целом

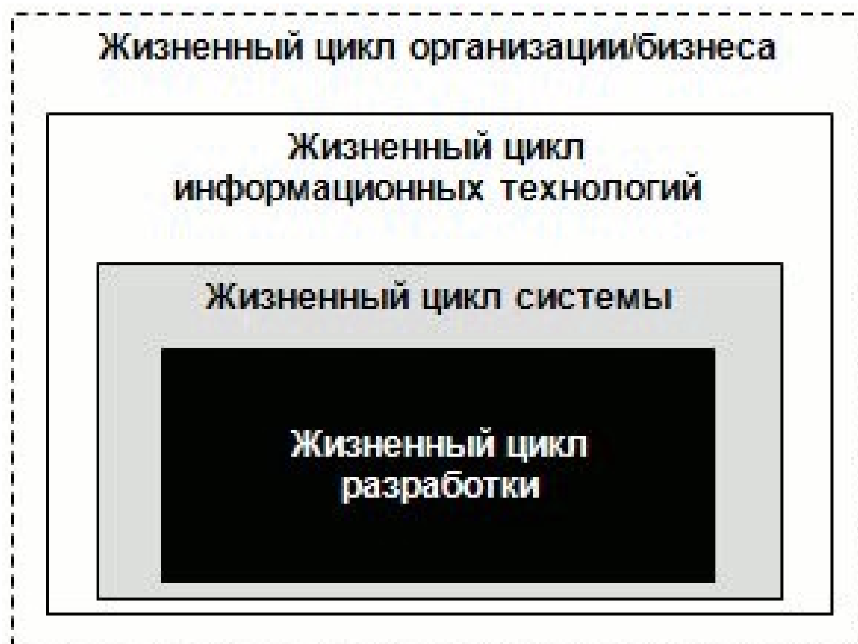

Рисунок 1: Содержание четырех категорий жизненного цикла по Амблеру (используется с разрешения автора) [Ambler, 2005]

Каждый стандарт группирует различные виды деятельности (как связанные непосредственно с разработкой, так и сопутствующие разработке — управленческие и организационные процессы) в группы процессов, среди которых выделяют следующие:

1. Основные процессы
2. Вспомогательные процессы
3. Организационные процессы

История и эволюция технологии программирования

В истории технологии программирования можно выделить три этапа.

1. Осмысление опыта разработки больших систем. Понимание того, что важно не только на каком языке программирования разрабатывается программа, но и как это делается. Проведение первых международных и национальных конференций (конец 60-х — 70-е годы XX века).
    1. 1968 г. — НАТО проводит первую конференцию по инженерии программирования (Software Engineering).
    2. 1975 г. — 1-я международная конференция IEEE.
    3. 1979 г. — 1-я Всесоюзная конференция по технологии программирования.
2. Разработка новых технологических подходов (начало 70-х годов XX века — настоящее время).
    1. 1973 г. — Дагласом Россом (Douglas Ross) разработана технология проектирования сложных систем SADT (Structured Analysis and Design Technique). Стандартизована под названием IDEF (Integrated DEFinition).
    2. 1985 г. — Харланом Миллзом (Harlan Mills) сформулированы основные идеи технологии стерильного цеха.
    3. 1995 г. — в октябре появилась первая пробная версия Унифицированного метода. Этот проект с 1996 года известен как UML (Unified Modeling Language)..
3. Принятие стандартов на состав процессов жизненного цикла программного обеспечения (середина 80-х годов XX века — настоящее время). Попытки решить проблему качества программных продуктов.
    1. 1985 г. — впервые утвержден стандарт жизненного цикла для проектирования программных систем (для систем военного назначения по заказам Министерства обороны США).
    2. 1994 г. — в Великобритании создан международный консорциум, разрабатывающий на постоянной основе проекты стандартов и технологии быстрого создания приложений DSDM (Dnamic Systems Development Method).
    3. 1995 г. — принят международный стандарт ISO 12207:1995

"Information Technology — Software Life Cycle Processes", регламентирующий состав процессов жизненного цикла программного обеспечения.

Основные процессы жизненного цикла программного обеспечения

К основным процессам жизненного цикла программного обеспечения относят следующие процессы:

1. приобретение;
2. поставка;
3. разработка;
4. эксплуатация;
5. сопровождение;

Приобретение

Процесс приобретения определяет задачи и работы заказчика в ходе приобретения программного обеспечения или услуг, связанных с программным обеспечением. В ГОСТ 12207 этот процесс называется «Заказ».

Процесс состоит из следующих работ (названия ГОСТ 12207 даны в скобках, если предлагают другой перевод названий работ оригинального стандарта):

1. Inititation – инициирование (подготовка)
2. Request-for-proposal preparation – подготовка запроса на предложение (подготовка заявки на подряд)
3. Contract preparation and update –подготовка и корректировка договора
4. Supplier monitoring – мониторинг поставщика (надзор за поставщиком)
5. Acceptance and completion – приемка и завершение (приемка и закрытие договора)

Поставка

Процесс поставки определяет работы и задачи поставщика программного обеспечения или услуги, связанной с поставляемым продуктом. Процесс включает следующие работы:

1. Inititation – инициирование (подготовка)
2. Preparation of response – подготовка предложения (подготовка ответа)
3. Contract – разработка контракта (подготовка договора)
4. Planning - планирование

5. Execution and control – выполнение и контроль
6. Review and evaluation –проверка и оценка
7. Delivery and completion – поставка и завершение (поставка и закрытие договора)

Разработка

Процесс разработки включает в себя работы и задачи разработчика по созданию программного обеспечения и его компонент в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов, необходимых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала и т.д. Процесс включает в себя следующие работы:

1. Process implementation – определение процесса (подготовка процесса)
2. System requirements analysis – анализ системных требований (анализ требований к системе)
3. System design – проектирование системы (проектирование системной архитектуры)
4. Software requirements analysis – анализ программных требований (анализ требований к программным средствам)
5. Software architectural design – проектирование программной архитектуры
6. Software detailed design – детальное проектирование программной системы (техническое проектирование программных средств)
7. Software coding and testing – кодирование и тестирование (программирование и тестирование программных средств)
8. Software integration – интеграция программной системы (сборка программных средств)
9. Software qualification testing – квалификационные испытания программных средств
10. System integration – интеграция системы в целом (сборка системы)
11. System qualification testing – квалификационные испытания системы
12. Software installation – установка (ввод в действие)
13. Software acceptance support – обеспечение приемки программных средств

Эксплуатация

Процесс эксплуатации определяет работы и задачи оператора службы поддержки. Процесс эксплуатации включает в себя работы по внедрению компонентов программного обеспечения в эксплуатацию (в частности — конфигурирование базы данных и рабочих мест пользователей, обеспечение эксплуатационной документацией, проведение обучения персонала и т.д.), и непосредственно эксплуатацию, в том числе локализацию проблем и устранение причин их возникновения, модификацию программного обеспечения в рамках установленного регламента, подготовку предложений по совершенствованию, развитию и модернизации системы.

Процесс включает следующие работы:

1. Process implementation – определение процесса (подготовка процесса)
2. Operational testing – операционное тестирование (эксплуатационные испытания)
3. System operation – эксплуатация системы
4. User support – поддержка пользователя

Сопровождение

Процесс сопровождения определяет работы и задачи, проводимые специалистами службы сопровождения. Процесс включает следующие работы:

1. Process implementation – определение процесса (подготовка процесса)
2. Problem and modification analysis – анализ проблем и изменений
3. Modification implementation – внесение изменений
4. Maintenance review/acceptance – проверка и приемка при сопровождении
5. Migration – миграция (перенос)
6. Software retirement – вывод программной системы из эксплуатации (снятие с эксплуатации)

Следует отметить, что стандарт 12207 не определяет последовательность и разбиение выполнения процессов во времени, адресуя этот вопрос также работам по адаптации стандарта к конкретным условиям и окружению и применению выбранных моделей, практик, техник и т.п.

Вспомогательные процессы

К вспомогательным процессам жизненного цикла программного обеспечения относят следующие процессы:

1. документирование;
2. управление конфигурацией;
3. обеспечение качества;
4. верификация;
5. аудит;
6. разрешение проблем

Документирование

Процесс направлен на формализованное описание информации, созданное в течении жизненного цикла программного обеспечения, состоит из набора действий, с помощью которых планируют, проектируют, разрабатывают, выпускают, редактируют, распространяют и сопровождают документы, необходимые для всех заинтересованных лиц — руководство, технических специалистов и пользователей системы.

Процесс составляют следующие действия:

1. подготовительную работу;
2. проектирование и разработку;
3. выпуск документации;
4. сопровождение документации.

Управление конфигурацией

При создании сложных информационных систем, состоящих из многих компонентов, каждый из которых может иметь различные версии, встает вопрос учета их связей и функций, создания унифицированной структуры, а также проблема обеспечения развития системы в целом. Процесс управления конфигурацией направлен на организацию, систематизацию учета и контроль внесения изменений в программное обеспечение на всех стадиях жизненного цикла.

Процесс включает следующие работы:

1. определение состояния компонентов ПО в системе;

2. управление модификациями ПО;
3. описание и подготовка отчетов о состоянии компонентов программного обеспечения и запросов на модификацию, обеспечение полноты, совместимости и корректности компонентов;
4. управление хранением и поставкой программного обеспечения.

Обеспечение качества

Процесс направлен на осуществление контроля и оценки программного обеспечения с целью обеспечения установленных стандартов качества — в частности, определение различий между действительными и ожидаемыми результатами и соответствия программного обеспечения исходным требованиям.

Процесс включает следующие действия:

1. подготовительную работу;
2. обеспечение качества продукта%
3. обеспечение качества процесса;
4. обеспечение прочих показателей качества системы.

Верификация

Это процесс определения того, отвечает ли текущее состояние разработки, достигнутое на данном этапе, требованиям этого этапа. Проверка позволяет оценить соответствие параметров разработки с исходными требованиями. Проверка частично совпадает с тестированием, которое связано с идентификацией различий между действительными и ожидаемыми результатами и оценкой соответствия характеристик программного обеспечения исходным требованиям. Процесс состоит из двух действий — подготовительной работы и непосредственно верификации.

Аудит

Процесс направлен на определение следующих аспектов использования разрабатываемого или уже разработанного программного обеспечения:

1. определение соответствия возможностей программного обеспечения и потребностей бизнеса, требованиям заказчика;
2. формализация бизнес-операций и описание основных деловых процессов автоматизируемых программным обеспечением;

3. описание технических характеристик и архитектурных особенностей программного обеспечения;
4. определение целесообразности дальнейшего использования программного обеспечения, его развития или же замены на новый продукт;
5. список выявленных возможных рисков, связанных с использованием программного обеспечения;
6. перечень конкретных рекомендаций, направленных на повышение качества программного обеспечения и оптимизацию его использования.

Процесс состоит из подготовительной работы и самого аудита.

Решение проблем

Процесс определяет работы и задачи, направленные на изменение программного обеспечения (независимо от источника их возникновения) для увеличения соответствия исходным или измененным требованиям. Состоит из двух этапов — подготовительной работы и самого процесса разрешения проблем.

Организационные процессы

К организационным процессам относят следующие процессы:

1. управление;
2. создание инфраструктуры;
3. усовершенствование;
4. обучение.

Управление

Процесс управления состоит из действий и задач, которые могут выполняться любой стороной, управляющей своими ресурсами. Данная сторона (менеджер) отвечает за управление выпуском продукта, управление проектом и управление задачами соответствующих процессов, таких, как приобретение, поставка, разработка, эксплуатация, сопровождение и т.д.

Процесс включает следующие действия:

1. определение области управления;
2. планирование;
3. выполнение и контроль;
4. процевку и оценку;
5. завершение.

Создание инфраструктуры

Процесс создания инфраструктуры охватывает выбор и поддержку (сопровождение технологии), стандартов и инструментальных средств, выбор и установку аппаратных и программных средств, используемых для разработки, эксплуатации или сопровождения программного обеспечения. Инфраструктура должна модифицироваться и сопровождаться в соответствии с изменениями требований к соответствующим процессам. Инфраструктура, в свою очередь, является одним из объектов управления конфигурацией.

Процесс состоит из трех действий — подготовительной работы, создания и сопровождения инфраструктуры.

Усовершенствование

Процесс усовершенствования предусматривает оценку, измерение, контроль и усовершенствование процессов жизненного цикла программного

обеспечения. Процесс включает 3 действия — создание, оценку и усовершенствование процесса.

Обучение

Процесс обучения охватывает первоначальное обучение и последующее постоянное повышение квалификации персонала.

Процесс составляют три действия — подготовительную работу, разработку учебных материалов и реализацию плана обучения.

Программные продукты для управления проектами разработки программного обеспечения

Когда говорят об инструментарии для управления проектами разработки программного обеспечения, обычно выделяют 3 группы систем управления:

- системы управления проектами;
- организационные средства;
- средства оценки качества.

Системы управления проектами

При управлении проектом необходимы следующие средства:

- средства описания комплекса работ, связей между отдельными работами и их временных характеристик;
- средства поддержки информации о ресурсах и затратах по проекту и его отдельными работами;
- средства контроля над ходом выполнения проекта;
- графические средства предоставления структуры проекта и средства создания отчетов по проекту.

Кроме того, система должна позволять изменять планы и сроки работ в режиме реального времени.

Наиболее распространены следующие системы управления проектами:

- Microsoft Project (компания Microsoft) — на сегодняшний одна из наиболее распространенных систем. Обычно используется для планирования несложных проектов. Пакет достаточно прост, имеет неплощую систему справок и удобные инструменты создания отчетов.
- Planner (http://live.gnome.org/Planner) — бесплатный продукт с открытым исходным кодом, практически полный аналог MS Project.
- Redmine (http://www.redmine.org) — система управления проектами, построенная на языке Ruby, совмещена с систему управления ошибками (баг-трекером).
- JIRA (http://www.atlassian.com/JIRA) — платная система управления проектами, построенная на языке Java, совмещена с системой управления ошибками. В ней вызывает определенные сложности создавать связи

между большим количеством вложенных задач.

- Spider Project — имеет следующие особенности: расписание выполнения работ и использование ресурсов; встроенная система анализа рисков и управления резервами по срокам и стоимости работ; возможность создания, хранения и включения в проекты типовых фрагментов проектов; оптимальная организация групповой работы и мультипроектного управления и многие другие.
- Open Plan — позволяет обеспечивать полномасштабное мультипроектное управление, планирование по методу критического пути и оптимизацию использования ресурсов в масштабах предприятия. Программный продукт может быть эффективно использован на всех уровнях контроля и управления проектами – от высшего руководства и менеджеров проектов, до начальников функциональных подразделений и рядовых исполнителей.

Организационные средства

К организационным средствам относят:

- электронную почту;
- электронный календарь;
- интарнет.

Электронная почта играет может играть очень важную организационную роль в процессе управления проектами. Нередко для больших распределенных проектов, электронная почта является чуть ли не основным инструментом для коммуникации и организации процессов. Преимуществом коммуникации посредством электронной почты является то, что почта сохраняется в истории. В связи с этим очень часто решения, принятые на совещаниях или встречах с заказчиками, дублируются по электронной почте.

Электронный календарь наряду с электронной почтой является очень эффективным организационным средством — в него заносятся пометки о будущих событиях — совещаниях, семинарах, крайние сроки важных событий и пр. Электронный календарь входит практически во все современные операционные системы, существуют веб-сервисы для электронных календарей, например http://calendar.google.com.

Под понятием интранет обычно понимают информационные системы, построенные на прицнипах сети Интернет. Внутренний сайт с системой разделения ролей часто выступает в роли хранилища проектной информации и т.п.

Средства оценки качества

Для сравнения качества программных продуктов применяются количественные методы оценки. Среди программ оценки качества часто отмечают Metricate компании Software Productivity Centre, которая специализируется на оценке деятельности компании по разработке программного обеспечения. Здесь рассматриваются эффективность технологических процессов, качество программного кода, уровень управления проектами, стоимость выполнения различных этапов, производительность получаемой системы, продуктивность труда разработчиков и качество производимых изделий.

Техника совместной разработки комплексных программных продуктов

Стандартные фазы разработки проектов

Выделим основные группы технологических подходов и укажем подходы для каждой из них.

1. Подходы со слабой формализацией. Эти подходы не используют явных технологий и их можно применять только для очень маленьких проектов, как правило, завершающихся созданием демонстрационного прототипа. К подходам со слабой формализацией относятся так называемые ранние технологические подходы, например подход "кодирование и исправление".
2. Строгие (классические, жесткие, предсказуемые) подходы. Данную группу подходов рекомендуется применять для средних, крупномасштабных и гигантских проектов с фиксированным объемом работ. Одно из основных требований к таким проектам — предсказуемость. В данную группу входят, в частности, каскадные подходы.
3. Гибкие (адаптивные, легкие) подходы. Подходы этой группы рекомендуется применять для небольших или средних проектов в случае неясных или изменяющихся требований к системе. Команда разработчиков должна быть ответственной и квалифицированной, а заказчики должны быть согласны принимать участие в разработке. В данную группу, в частности, входит экстремальное программирование.

Метод «кодирование и исправление»

Подход «кодирование и исправление» (code and fix) относится к группе неформализованных технологических подходов или как еще говорят к группе подходов со слабой формализацией.

Чаще всего разработчик начинает кодирование системы с самого первого дня, не занимаясь сколь-нибудь серьезным проектирование. Это обычно приводит к тому, что ошибки обнаруживаются к концу кодирования и требуют исправления через повторное кодирование.

Этот подход рекомендуется использовать для очень небольших проектов, завершающихся разработкой демонстрационного прототипа, либо для целей обучения программированию, либо для доказательства некоторой концепции.

Каскадный метод разработки

Обычно под каскадным методом разработки понимают группу подходов, где последовательность выполнения процессов обычно изображают в виде каскада. Эти подходы также иногда называют подходами на основе модели водопада.

Классический каскадный подход

Каскадный подход (pure waterfall) считается прародителем всех технологических подходов к ведению жизненного цикла. Фактически, его можно рассматривать гак отправную точку для огромного количества других подходов. Сформировался каскадный подход в период с 1970 по 1985 годы. Специфика "чистого" каскадного подхода предполагает строго последовательное (во времени) и однократное выполнение всех фаз проекта с жестким (детальным) предварительным планированием в контексте предопределенных или однажды и целиком определенных требований к программной системе. Возвраты к уже пройденным процессам не предусмотрены.

Будучи активно используема (де факто и, например, в свое время, как часть соответствующего отраслевого стандарта в США), эта модель продемонстрировала свою “проблемность” в подавляющем большинстве ИТ-проектов, за исключением, может быть, отдельных проектов обновления программных систем для критически-важных программно-аппаратных комплексов (например, авионики или медицинского оборудования). Практика показывает, что в реальном мире, особенно в мире бизнес-систем, каскадная модель не должна применяться. Специфика таких систем (если можно говорить о “специфике” для подавляющего большинства создаваемых систем) - требования характеризуются высокой динамикой корректировки и уточнения, невозможностью четкого и однозначного определения требований до начала работ по реализации (особенно, для новых систем) и быстрой изменчивостью в процессе эксплуатации системы.

Фредерик Брукс во втором издании своего классического труда “Мифический человеко-месяц” так описывает главную беду каскадной модели [Брукс, 1995]:

> “Основное заблуждение каскадной модели состоит в предположениях, что проект проходит через весь процесс *один раз*, архитектура хороша и проста в использовании, проект осуществления разумен, а ошибки в

реализации устраняются по мере тестирования. Иными словами, каскадная модель исходит из того, что все ошибки будут сосредоточены в реализации, а потому их устранение происходит равномерно во время тестирования компонентов и системы.”

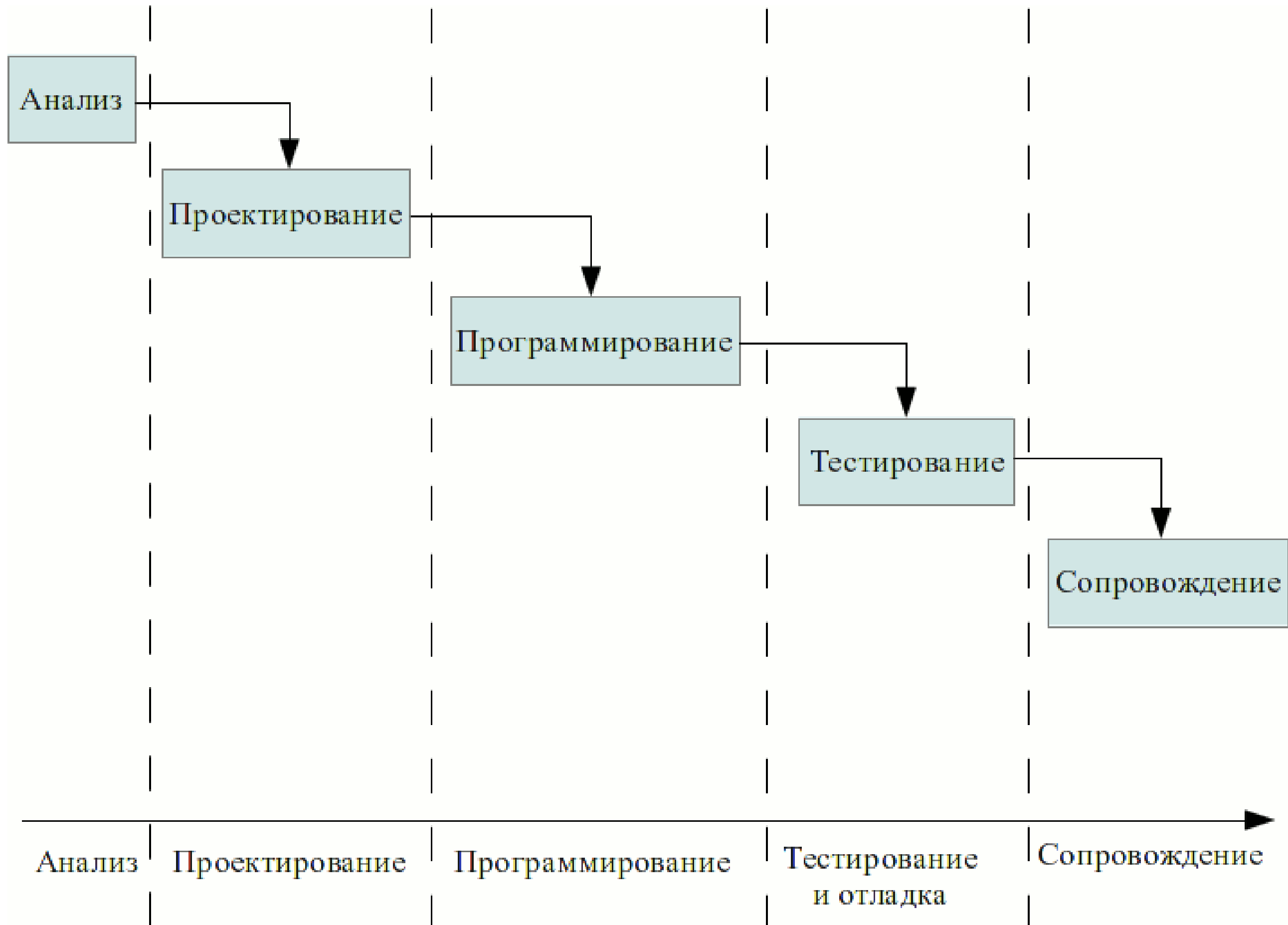

Рисунок 2: "Чистый" каскадный технологический подход

В каскадной модели переход от одной фазы проекта к другой предполагает полную корректность результата (выхода) предыдущей фазы. Однако, например, неточность какого-либо требования или некорректная его интерпретация, в результате, приводит к тому, что приходится “откатываться” к ранней фазе проекта и требуемая переработка не просто выбивает проектную команду из графика, но приводит часто к качественному росту затрат и, не исключено, к прекращению проекта в той форме, в которой он изначально задумывался. Кроме того, эта модель не способна гарантировать необходимую скорость отклика и внесение соответствующих изменений в ответ на быстро меняющиеся потребности пользователей, для которых программная система является одним из инструментов исполнения бизнес-функций.

Данный подход может быть рекомендован к применению в тех проектах, где в самом начале все требования могут быть сформулированы точно и полно. Например, в задачах вычислительного характера. Достаточно легко при таком

технологическом подходе вести планирование работ и формирование бюджета.

Каскадно-возвратный подход

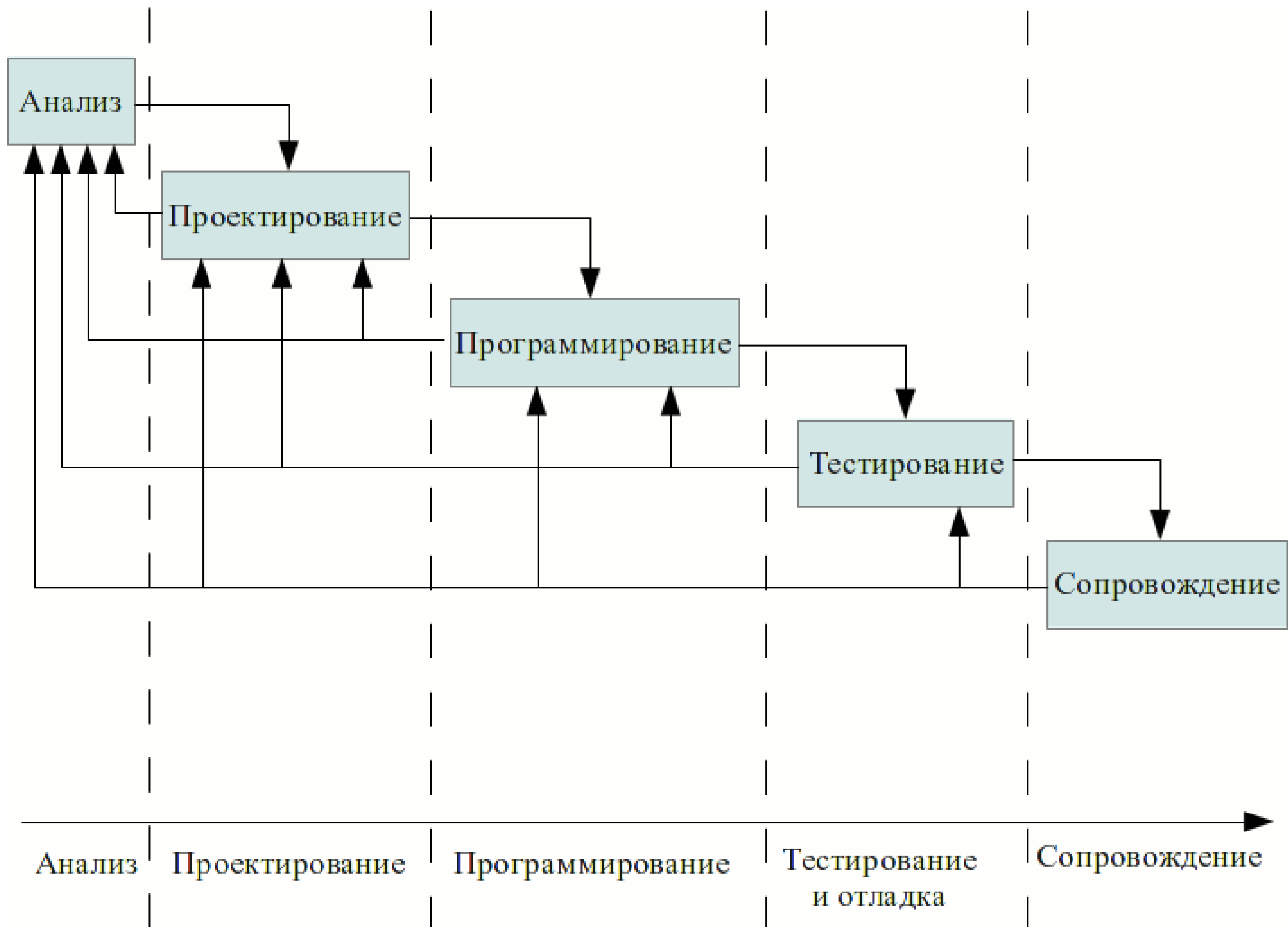

Рисунок 3: Каскадно-возвратный подход

Как было отмечено, основной недостаток каскадного подхода — отсутствие гибкости. Для преодоления этой трудности был предложен каскадно-возвратный подход, в котором разрешены возвраты к предыдущим стадиям и пересмотр или уточнение ранее принятых решений. Каскадно-возвратный подход отражает итерационный характер разработки программного обеспечения. Этот подход в значительной степени отражает реальный процесс создания программного обеспечения, в том числе и существенное запаздывание с достижением результата. На задержку оказывают существенное влияние корректировки при возвратах.

Каскадно-итерационный подход

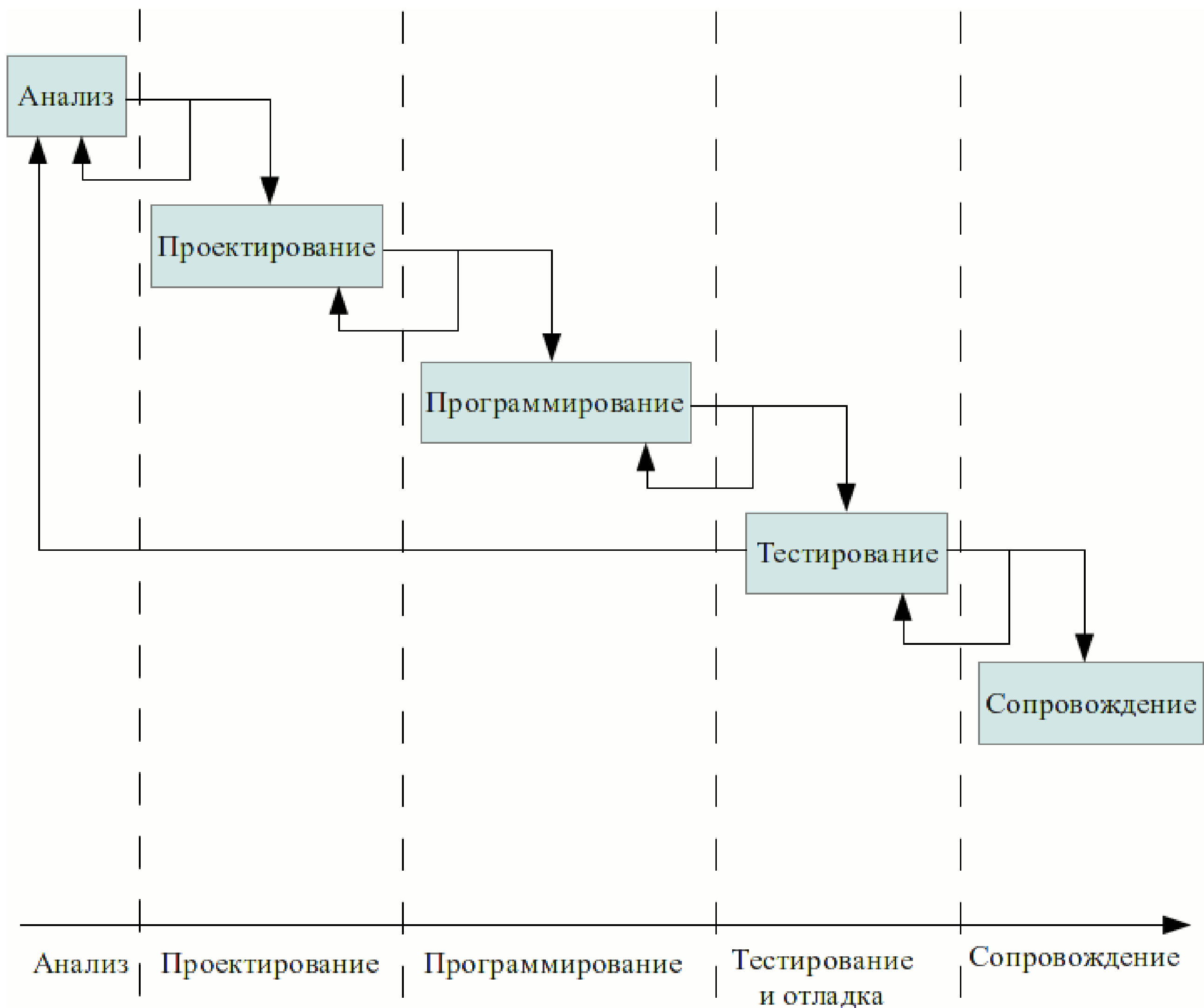

Рисунок 4: Каскадно-итерационный подход

Этот подход предусматривает последовательные итерации каждого процесса до тех пор, пока не будет достигнут желаемый результат. Каждая итерация является завершенным этапом, и ее итогом будет некоторый конкретный результат. Возможно, данный результат будет промежуточным, не реализующим всю ожидаемую функциональность.

Каскадный подход с перекрывающимися процессами

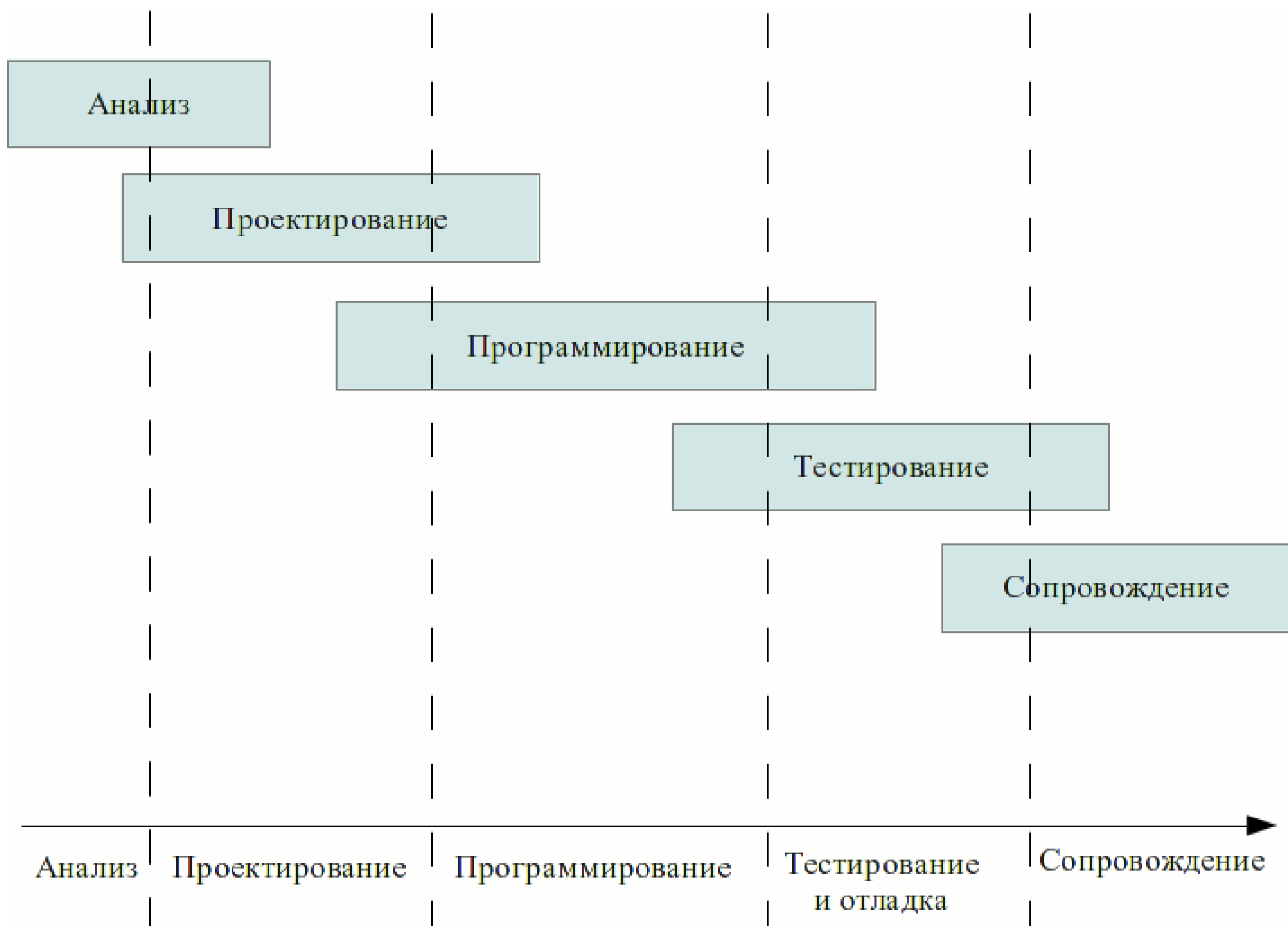

Рисунок 5: Каскадный подход с перекрывающимися процессами

Классический каскадный подход позволяет выполнять каждый процесс отдельной команде. Достаточно разумным является использование команды на том же самом процессе в следующей разработке. Каскадный подход с перекрывающимися процессами (waterfall with overlapping) предполагает наличие таких специализированных команд, позволяющих до определенной степени сократить передаваемую документацию. Следующий процесс начинается до завершения текущего. Более того, несколько процессов могут выполняться параллельно.

Каскадный подход с подпроцессами

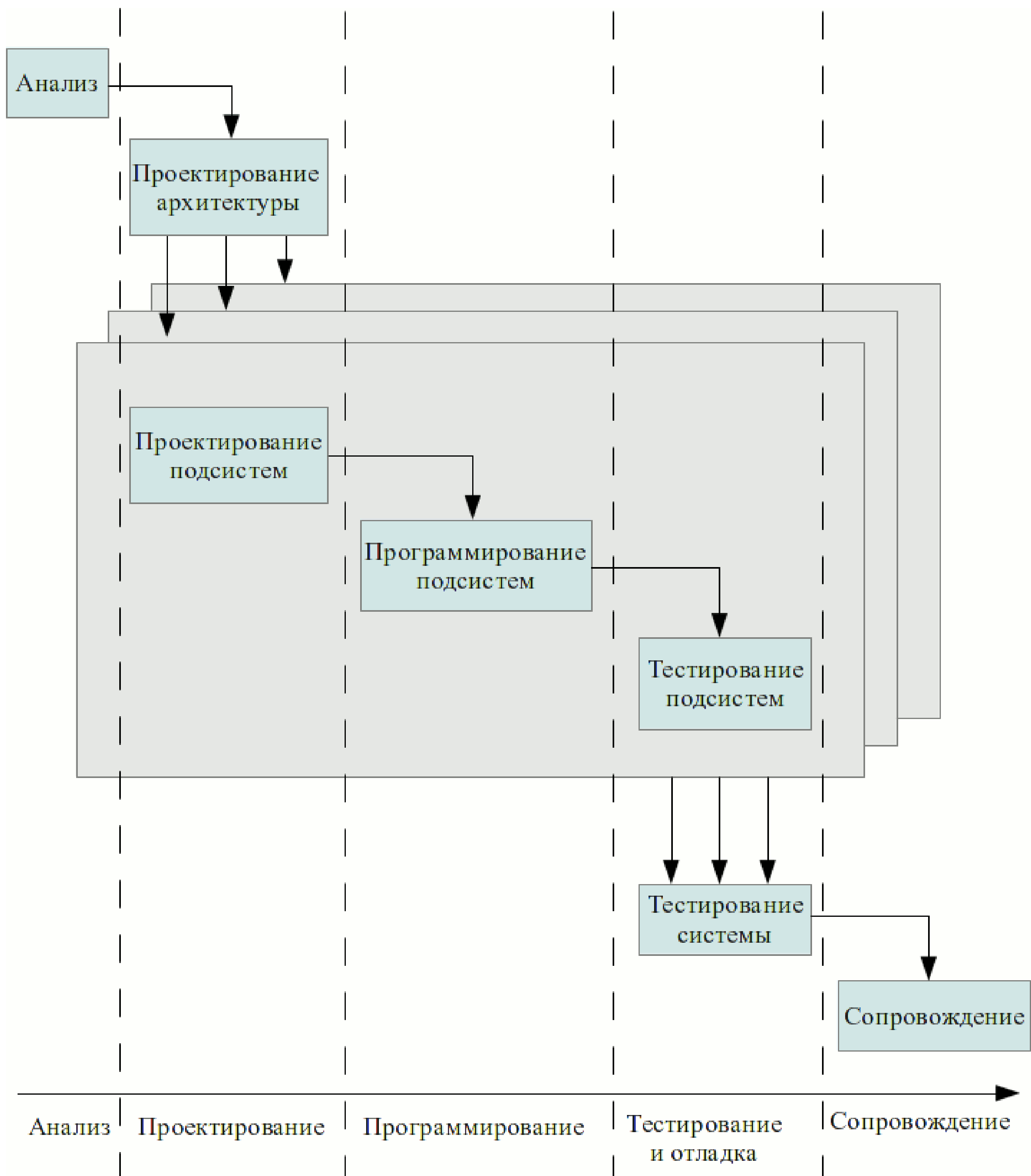

Рисунок 6: Каскадный подход с подпроцессами

Каскадный подход с подпроцессами (waterfall with subprocesses) очень близок подходу с перекрывающимися процессами. Особенность его в том, что с архитектурной точки зрения проект достаточно часто может быть разделен на подпроекты, которые могут разрабатываться индивидуально. В данном подходе

требуется дополнительная фаза тестирования подсистем до объединения их в единую систему. Следует особое внимание обращать на грамотное деление проекта на подпроекты, которое должно учесть все возможные зависимости между подсистемами.

V-model

V-Model (или VEE модель, или Vee-модель) является моделью разработки информационных систем (ИС), направленной на упрощение понимания сложностей, связанных с разработкой систем. Она используется для определения единой процедуры разработки программных продуктов, аппаратного обеспечения и человеко-машинных интерфейсов.

Концепция V-образной модели была разработана Германией и США в конце 1980-х годов независимо друг от друга:

- Немецкая V-модель.

  Была разработана аэрокосмической компанией IABG в Оттобрунне рядом с Мюнхеном в содействии с Федеральным департаментом по закупке вооружений в Кобленце, для Министерства обороны Германии. Модель была принята немецкой федеральной администрацией для гражданских нужд летом 1992.

- Американская V-Model (VEE).

  Была разработана национальным советом по системной инженерии (международным — с 1995 года) для спутниковых систем, включая оборудование, программное обеспечение и взаимодействие с пользователями.

Современной версией V-Model является V-Model XT, которая была утверждена в феврале 2005 года. V-модель используется для управления процессом разработки программного обеспечения для немецкой федеральной администрации. Сейчас она является стандартом для немецких правительственных и оборонных проектов, а также для производителей ПО в Германии. V-Model представляет собой скорее набор стандартов в области проектов, касающихся разработки новых продуктов. Эта модель во многом схожа с PRINCE2 и описывает методы как для проектного управления, так и для системного развития.

Основной принцип V-образной модели заключается в том, что детализация проекта возрастает при движении слева направо, одновременно с течением времени, и ни то, ни другое не может повернуть вспять. Итерации в проекте производятся по горизонтали, между левой и правой сторонами буквы.

Применительно к разработке информационных систем V-Model — вариация каскадной модели, в которой задачи разработки идут сверху вниз по

левой стороне буквы V, а задачи тестирования — вверх по правой стороне буквы V. Внутри V проводятся горизонтальные линии, показывающие, как результаты каждой из фаз разработки влияют на развитие системы тестирования на каждой из фаз тестирования. Модель базируется на том, что приемо-сдаточные испытания основываются, прежде всего, на требованиях, системное тестирование — на требованиях и архитектуре, комплексное тестирование — на требованиях, архитектуре и интерфейсах, а компонентное тестирование — на требованиях, архитектуре, интерфейсах и алгоритмах.

V-модель организует фазы разработки исходя из уровня сложности, где наиболее сложный пункт будет вверху, а самый простой — внизу (также известный как Самый нижний пункт конфигурации).

Это ставит требования в начало рядом с функционированием продукта в конце и проектирование рядом с проверкой. Это имеет смысл, так как когда разработчик поставляет продукт клиенту, клиент может спросить: «Почему я должен принять этот продукт?» и разработчик должен ответить: «Потому что он отвечает вашим (клиентским) требованиям». Требования связаны с функционированием. При выполнении тестирования продукта, инженер-тестировшик может спросить: «Какие тесты я должен провести?» и проектировщик должен ответит: «Вы должны провести тесты, чтобы показать продукт, который был построен в соответствии с проектом.» Проверка связана с проектом. V-модель может быть расширена в нескольких направлениях для удовлетворения системных требований. Это может включать в себя семь INCOSE слоев сложности системы (например система, элемент, подсистема, сборка, подсборка, компонент и часть). Это может включать в себя разбиение, определение, интеграцию и проверку. Также это может включать участие пользователей/заинтересованных сторон, управление рисками и решения проблем.

Рисунок 7: Структура V-Model

При разработке сложных систем, системный инженер должен управлять итоговой конфигурацией системы от начала и до конца. Итоговая конфигурация может включать проектную документацию, руководства по эксплуатации, сам продукт и должна отвечать на вопросы Что, Почему? и Кто? по архитектуре системы. На каждой фазе разработки будут изменения в системе, которые приведут к изменению итоговой конфигурации. Ядро Vee является развивающаяся модель от начальных требований к готовой системе. Vee Архитектура отвечает на вопросы что, почему и кто (какой уровень системы) отвечает за архитектуру системы. Нисходящие из V ядра исследования усиливают процесс получения знаний для подтверждения итоговых архитектурных решений сделанных в ядре V. Восходящие из ядра взаимодействия с клиентами и пользователями усиливают поэтапное подтверждение поддерживая вовлеченность заинтересованных лиц в конечном продукте. Необходимо обратить внимание на то, что во всех V моделях представление времени и завершенности продукта движется слева направо. Так же как мы не можем перемещаться назад во времени, также мы не можем двигаться справа налево в V моделях. Последовательное приближение — это основное в разработке системы и все шаги делаются вертикально от ядра, вверх к пользователям и клиентам (что есть поэтапное подтверждение) и вниз к управлению рисками и новыми возможностями.

Левая ветвь Vee ядра сосредоточена вокруг того, какая концепция лучше и какая архитектура лучше для этой концепции. Например, коммерческие продукты обычно сталкиваются с дилеммой: батареи должны быть стандартными, уникальными, заменяемые или нет нисходящие из V ядра исследования и анализы могут ускорить определение наиболее желательного метода, который мог бы быть позже зафиксирован на V ядре если все заинтересованные лица согласны. Подобные исследования могут доказать жизнеспособность и технические возможности варианта концепции. Правая ветвь V ядра нисходящих исследований управляется на уровне исследования интеграционных аномалий для определения причин и устранения их. Восходящие взаимодействия с заинтересованными лицами определяют, согласны ли они с такой интеграцией и выполненной проверкой.

На каждом представленном уровне существует прямая связь между действиями с левой и правой стороны V модели. Это сделано преднамеренно. Например, метод интеграции, проверки и утверждения которые будут использоваться в правой части должны быть определены в левой части также как концепции, определенные на каждом из уровней. Это минимизирует вероятность того, что концепции задуманы таким образом, что их невозможность осуществить.

Структура V модели показывает ее разработку и реализацию, процесс который описывает как каждый объект будет получен (разработан, приобретен, повторно использован и т. д.) Модель Vee существует для каждого объекта архитектуры начиная с уровня системы вниз до уровня мельчайших конфигурационных элементов (LCI), такие как элементы компьютерного обеспечения или микросхемы. Все действия внутри объекта Vee находятся на том же самом уровне архитектура (Система, Подсистема, LCI). Левая ветвь Vee представляет определение модели, ее развитие от черновых требований клиента, через определение концепции и до описания решения и полностью построенного образца. Правая ветвь Vee представляет последовательность сборки объекта и ее гарантированная производительность, полученная посредством проверок и утверждений объекта.

В каждой разработке, существует зависимость между действиями на левой и правой ветви Объекта Vee. Это сделано специально. Метод проверки, которые будут использоваться в правой ветви Vee должны быть определены одновременно с разработкой требований на левой ветви, в противном случае могут быть созданы требования, которые не могут быть проверены. Например, «быть дружественным к пользователю» является подходящим требованием, но

его проверить невозможно. Вместо этого, требование, которое утверждает что на экране компьютера может быть «не более пяти строк текста, набранным 14 шрифтом текста» определяет один пользовательский критерий требования «быть дружественным к пользователю» в измеримых величинах. План проверок должны быть согласован и зафиксирован чтобы гарантировать, что требования к проверкам и их методы известны и запланированы на этапе принятия решения по разработке, называемого Проверка Предварительного Проекта (Preliminary Design Review (PDR)). Черновые процедуры проверок основанные на требованиях к проверкам, плане проверок, и предложенном проекте объекта должны быть известны на этапе принятия решения по созданию и программированию, обычно называемом Окончательная Оценка Разработки (Critical Design Review (CDR)). Это снижает вероятность того, что проверка, в том виде котором она определена не может, быть выполнена. Вертикальное размер Vee Объекта это зафиксированная разработка на выбранном уровне архитектуры и ядро Vee Объекта представляющее итоговую последовательность разработки объекта. Также включены (по аналогии с Vee Архитектурой) действия, связанные с управлением возможностями и рисками, осуществляемые в нисходящем от ядра объекта направлении до уровня необходимого для оценки и решения проблемы. Например, лабораторные испытания компьютерного чипа или программного обеспечения могут быть необходимы для подтверждения технической пригодности.

В отличие от широко распространенного мнения о Модели Водопада, не существует запрета на выполнение пробной разработки и её анализа в любой точке проектного цикла для исследования или доказательства производительности или пригодности. В отличие от Спиральной Модели, в Vee модели исследования возможностей и рисков могут быть выполнены последовательно или в параллель с основной разработкой, а не проводиться постфактум или до процесса разработки решения. Аппаратные и программные требования понимания моделей или технической пригодности моделей приветствуются в начале проектного цикла для реализации таких возможностей, как новые технологии, и для снижения риска. Например, для оценки концепции ручной корректировки текста в сравнении с полностью автоматической корректировкой, техническая пригодность обоих концепций может быть смоделирована и выбор может быть сделан на основе сравнения времени и стоимости решений. Согласие клиента может также обеспечить необходимое в проектном цикле утверждение более предпочтительного метода.

В правой ветви, нисходящие от основного процесса запросы применяются

для выполнения сбора и проверки отклонений. Это может подразумевать происходящие от проекта ошибки, дефекты при пайке или ошибка оператора и тому подобное. Восходящие от основного процесса пользовательские взаимодействия — получении пользовательского или клиентского подтверждения или отказ от достигнутых результатов. Обратите внимание на то, что в объекте Vee эти взаимодействия указывают на индивидуальные решения в модели, а не на интегрированную архитектуру, выполненную на Vee архитектуре. На любом уровне представленной модели клиент модели в то же время является управляющим нескольких более высоких уровней модели. Например, человек, управляющий подсистемой электропитания, является клиентом на уровне зарядных батарей, и он выполняет приёмку этих самых батарей.

V-модель обеспечивает поддержку в планировании и реализации проекта. В ходе проекта ставятся следующие задачи:

- Минимизация рисков.

  V-образная модель делает проект более прозрачным и повышает качество контроля проекта путем стандартизации промежуточных целей и описания соответствующих им результатов и ответственных лиц. Это позволяет выявлять отклонения в проекте и риски на ранних стадиях и улучшает качество управления проектов, уменьшая риски.

- Повышение и гарантии качества.

  V-Model — стандартизованная модель разработки, что позволяет добиться от проекта результатов желаемого качества. Промежуточные результаты могут быть проверены на ранних стадиях. Универсальное документирование облегчает читаемость, понятность и проверяемость.

- Уменьшение общей стоимости проекта.

  Ресурсы на разработку, производство, управление и поддержку могут быть заранее просчитаны и проконтролированы. Получаемые результаты также универсальны и легко прогнозируются. Это уменьшает затраты на последующие стадии и проекты.

- Повышение качества коммуникации между участниками проекта.

  Универсальное описание всех элементов и условий облегчает взаимопонимание всех участников проекта. Таким образом, уменьшаются

неточности в понимании между пользователем, покупателем, поставщиком и разработчиком.

Выделяют следующие особенности V-Model.

- Пользователи V-Model участвуют в разработке и поддержке V-модели. Комитет по контролю за изменениями поддерживает проект и собирается раз в год для обработки всех полученных запросов на внесение изменений в V-Model.
- На старте любого проекта V-образная модель может быть адаптирована под этот проект, так как эта модель не зависит от типов организаций и проектов.
- V-model позволяет разбить деятельность на отдельные шаги, каждый из которых будет включать в себя необходимые для него действия, инструкции к ним, рекомендации и подробное объяснение деятельности.

Следующие моменты не учитываются в V-модели, но могут быть рассмотрены отдельно, либо возможно адаптировать модель под них:

- Не регулируется размещение контрактов на обслуживание.
- Организация и выполнение управления, обслуживания, ремонта и утилизации системы не учитываются в V-модели. Однако, планирование и подготовка к этим операциям моделью рассматриваются.
- V-образная модель больше касается разработки программного обеспечения в проекте, чем всей организации процесса.

По мнения различных разработчиков о достоинствах и недостатках V-model, не входящих в официальную документацию, можно выделить следующие достоинства:

- В модели особое значение придается планированию, направленному на верификацию и аттестацию разрабатываемого продукта на ранних стадиях его разработки. Фаза модульного тестирования подтверждает правильность детализированного проектирования. Фазы интеграции и тестирования реализуют архитектурное проектирование или проектирование на высшем уровне. Фаза тестирования системы подтверждает правильность выполнения этапа требований к продукту и его спецификации.

- В модели предусмотрены аттестация и верификация всех внешних и внутренних полученных данных, а не только самого программного продукта.
- В V-образной модели определение требований выполняется перед разработкой проекта системы, а проектирование ПО — перед разработкой компонентов.
- Модель определяет продукты, которые должны быть получены в результате процесса разработки, причем каждые полученные данные должны подвергаться тестированию.
- Благодаря модели менеджеры проекта могут отслеживать ход процесса разработки, так как в данном случае вполне возможно воспользоваться временной шкалой, а завершение каждой фазы является контрольной точкой.

В тоже время, к недостаткам относят:

- Модель не предусматривает работу с параллельными событиями.
- В модели не предусмотрено внесение требования динамических изменений на разных этапах жизненного цикла.
- Тестирование требований в жизненном цикле происходит слишком поздно, вследствие чего невозможно внести изменения, не повлияв при этом на график выполнения проекта.
- В модель не входят действия, направленные на анализ рисков.
- Некоторый результат можно посмотреть только при достижении низа буквы V.

Спиральная модель разработки

Спиральная модель (spiral model) была предложена Барри Боэмом (Barry Boem) в середине 80-х годов XX века с целью сократить возможный риск разработки. Фактически, это была первая реакция на устаревание каскадной модели. Спиральная модель использует понятие прототипа — программы, реализующей частичную функциональность создаваемого программного продукта. Создание прототипов осуществляется за несколько витков спирали, каждый из которых состоит из "анализа риска", "некоторого процесса" и "верификации". Обращение к каждому процессу предваряет "анализ риска", причем, если риск превышения сроков и стоимости проекта оказывается существенным, то разработка заканчивается. Это позволяет предотвратить более крупные денежные потери в будущем.

Боэм формулирует десять наиболее распространённых (по приоритетам) рисков:

1. Дефицит специалистов.
2. Нереалистичные сроки и бюджет.
3. Реализация несоответствующей функциональности.
4. Разработка неправильного пользовательского интерфейса.
5. «Золотая сервировка», перфекционизм, ненужная оптимизация и оттачивание деталей.
6. Непрекращающийся поток изменений.
7. Нехватка информации о внешних компонентах, определяющих окружение системы или вовлечённых в интеграцию.
8. Недостатки в работах, выполняемых внешними (по отношению к проекту) ресурсами.
9. Недостаточная производительность получаемой системы.
10. «Разрыв» в квалификации специалистов разных областей знаний.

Большая часть этих рисков связана с организационными и процессными аспектами взаимодействия специалистов в проектной команде.

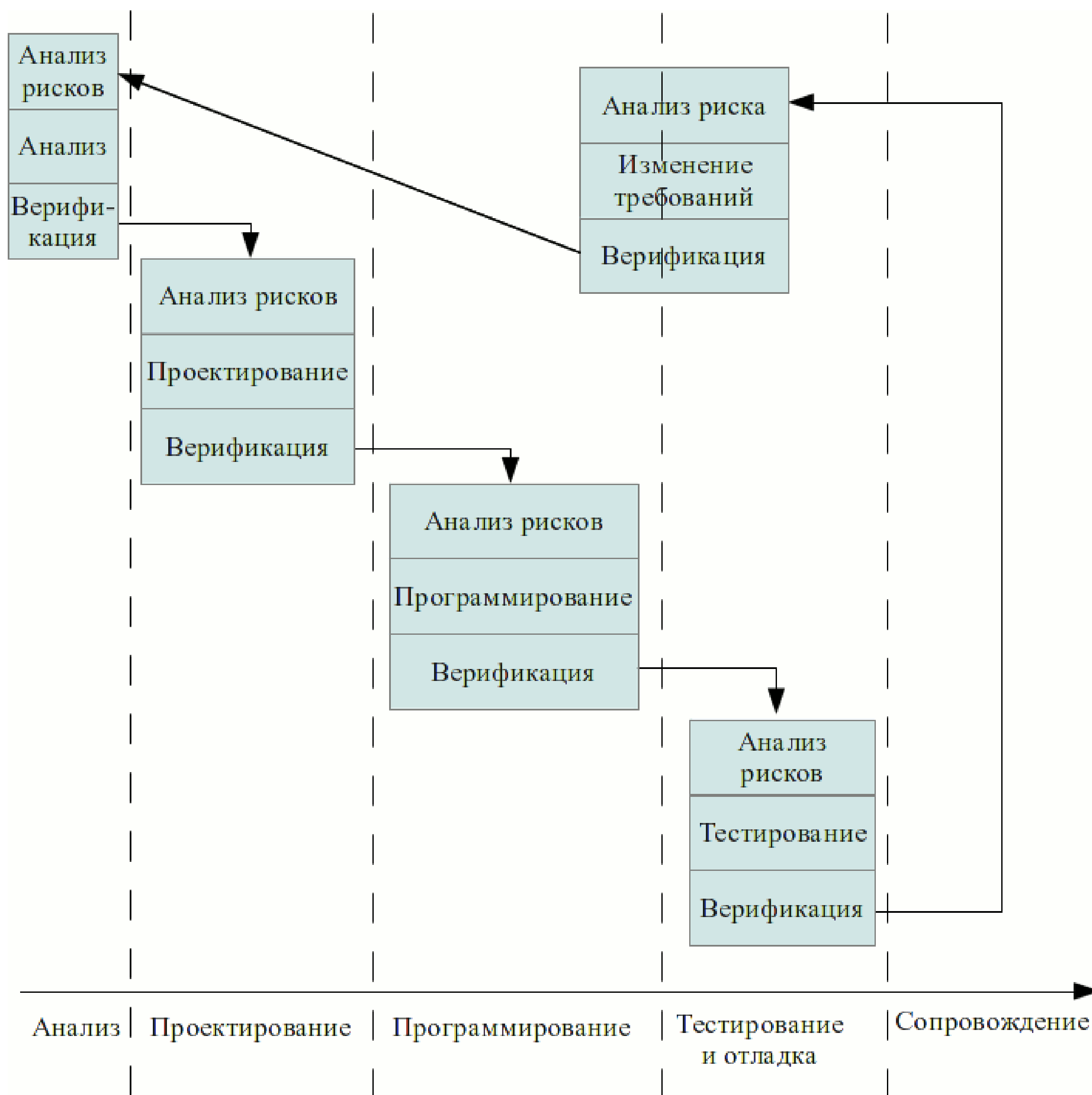

Рисунок 8: Спиральная модель

Особенность спиральной модели — в разработке итерациями. Причем каждый следующий итерационный прототип будет обладать большей функциональностью.

На каждом витке спирали могут применяться разные модели процесса разработки ПО. В конечном итоге на выходе получается готовый продукт. Модель сочетает в себе возможности модели прототипирования и водопадной модели. Разработка итерациями отражает объективно существующий спиральный цикл создания системы. Неполное завершение работ на каждом этапе позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем. При итеративном способе разработки недостающую работу можно будет выполнить на следующей итерации. Главная задача — как можно быстрее показать пользователям системы

работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований. Основная проблема спирального цикла — определение момента перехода на следующий этап. Для ее решения необходимо ввести временные ограничения на каждый из этапов жизненного цикла. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков. Одним из возможных подходов к разработке программного обеспечения в рамках спиральной модели жизненного цикла является получившая в последнее время широкое распространение методология быстрой разработки приложений RAD (Rapid Application Development). Под этим термином обычно понимается процесс разработки программного обеспечения, содержащий 3 элемента:

- небольшую команду программистов (от 2 до 10 человек);
- короткий, но тщательно проработанный производственный график (от 2 до 6 месяцев);
- повторяющийся цикл, при котором разработчики, по мере того, как приложение начинает обретать форму, запрашивают и реализуют в продукте требования, полученные через взаимодействие с заказчиком.

Жизненный цикл программного обеспечения по методологии RAD состоит из четырех фаз:

- фаза определения требований и анализа;
- фаза проектирования;
- фаза реализации;
- фаза внедрения.

Спиральная модель ориентирована на большие, дорогостоящие и сложные проекты. В условиях, когда бизнес цели таких проектов могут измениться, но требуется разработка стабильной архитектуры, удовлетворяющей высоким требованиям по нагрузке и устойчивости, имеет смысл применение Spiral Architecture Driven Development. Данная методология, включающая в себя лучшие идеи спиральной модели и некоторых других, позволяет существенно снизить архитектурные риски, что является немаловажным фактором успеха при разработке крупных систем.

Модель Хаоса

В компьютерных вычислениях Модель хаоса — это способ разработки программного обеспечения. Ее создатель Л.Б.С.Ракун отмечает, что такие модели управления проектами, как спиральная модель и каскадная модель, хотя и хороши в управлении расписаниями и персоналом, но не обеспечивают методами устранения ошибок и решениями других технических задач, не помогают, ни в управлении конечными сроками, ни в реагировании на запросы клиентов. Модель хаоса — это инструмент пытающийся помочь понять эти ограничения и восполнить пробелы.

Модель хаоса отмечает, что фазы жизненного цикла распространяются на все уровни проекта, от всего проекта в целом, до отдельной строки кода.

- В целом проект должен быть определен, реализован и интегрирован.
- Системы должны быть определены, реализованы и интегрированы.
- Модули должны быть определены, реализованы и интегрированы.
- Функции должны быть определены, реализованы и интегрированы.
- Строки кода должны быть определены, реализованы и интегрированы.

Одно важное изменение в перспективе — это могут ли проекты быть представлены, как цельные модули или должны быть представлены по частям. Никто не пишет десять тысяч строк кода в один присест. Все пишут небольшими частям, одну строку за раз, проверяя, чтобы этот небольшой кусочек работал. Затем уже над этим надстраивают следующие этажи. Поведение сложной системы исходит из комбинированных поведений составляющих ее меньших блоков.

Стратегия хаоса — это стратегия разработки программного обеспечения основанная на модели хаоса. Главное правило — это, всегда решать наиболее важную задачу первой.

- Задача это незавершенная частная задача программирования.
- Наиболее важная задача это комбинация большого размера, срочности и устойчивости.
  - Задачи большого размера ценны для пользователей настолько, насколько они функциональны.
  - Срочные задачи своевременны настолько, насколько должны быть,

иначе задерживается остальная работа.
    - Устойчивые задачи проверены и испытаны. Разработчики могут благополучно сфокусироваться на другом.
- Решить, означает привести в состояние стабильности.

Стратегия хаоса похожа на путь по которому программисты работают прямо в конце проекта, когда у них есть список ошибок для исправления и возможность для творчества. Обычно, кто-то расставляет приоритет оставшимся частным задачам и программисты устраняют их по одной. Стратегия хаоса утверждает, что это единственный корректный путь выполнения работы.

Каркасные технологические подходы

Такие подходы представляют собой каркас для процессов.

Рациональный унифицированный процесс

Рациональный унифицированный процесс (RUP) вобрал в себя все лучшее из технологических подходов каскадной группы. В нем выделяются четыре основные фазы, отражающие крупные временные отрезки:

1. начало — определение бизнес-целей проекта;
2. исследование — разработка плана и архитектуры проекта;
3. построение — постепенное создание системы;
4. внедрение — поставка системы конечным пользователям.

В период прохождения этих фаз функционируют процессы, такие как анализ, проектирование, тестирование, реализация и другие.

Основные особенности подхода заключаются в следующем:

1. итеративность с присущей ей гибкостью;
2. контроль качества, возможность выявить и устранить риски на как можно более ранних этапах;
3. предпочтение в первую очередь отдается моделям, а не бумажным документам с текстовым описанием;
4. основное внимание уделяется раннему определению архитектуры;
5. возможность конфигурирования, настройки и масштабирования.

OpenUP

OpenUP — это итеративно-инкрементальный метод разработки ПО. Позиционируется как легкий и гибкий вариант RUP.

В основу OpenUP положены следующие основные принципы:

- Совместная работа с целью согласования интересов и достижения общего понимания;
- Развитие с целью непрерывного обеспечения обратной связи и совершенствования проекта;
- Концентрация на архитектурных вопросах на ранних стадиях для минимизации рисков и организации разработки;
- Выравнивание конкурентных преимуществ для максимизации потребительской ценности для заинтересованных лиц.

OpenUP делит жизненный цикл проекта на четыре фазы:

1. начальная фаза,
2. фазы уточнения,
3. конструирования,
4. передачи.

Жизненный цикл проекта обеспечивает предоставление заинтересованным лицам и членам коллектива точек ознакомления и принятия решений на протяжении всего проекта. Это позволяет эффективно контролировать ситуацию и вовремя принимать решения о приемлемости результатов. План проекта определяет жизненный цикл, а конечным результатом является окончательное приложение.

OpenUP делит проект на итерации: планируемые, ограниченные во времени интервалы, длительность которых обычно измеряется неделями. План итерации определяет, что именно должно быть сдано по окончании итерации, а результатом является работоспособная версия. Коллективы разработчиков OpenUP строятся по принципу самоорганизации, решая вопросы выполнения задач итераций и передачи результатов. Для этого они сначала определяют, а затем решают хорошо детализированные задачи из списка элементов работ.

Базовый процесс OpenUP является независимым от инструментов, малорегламентированным процессом, который может быть расширен для удовлетворения потребностей широкого диапазона типов проектов.

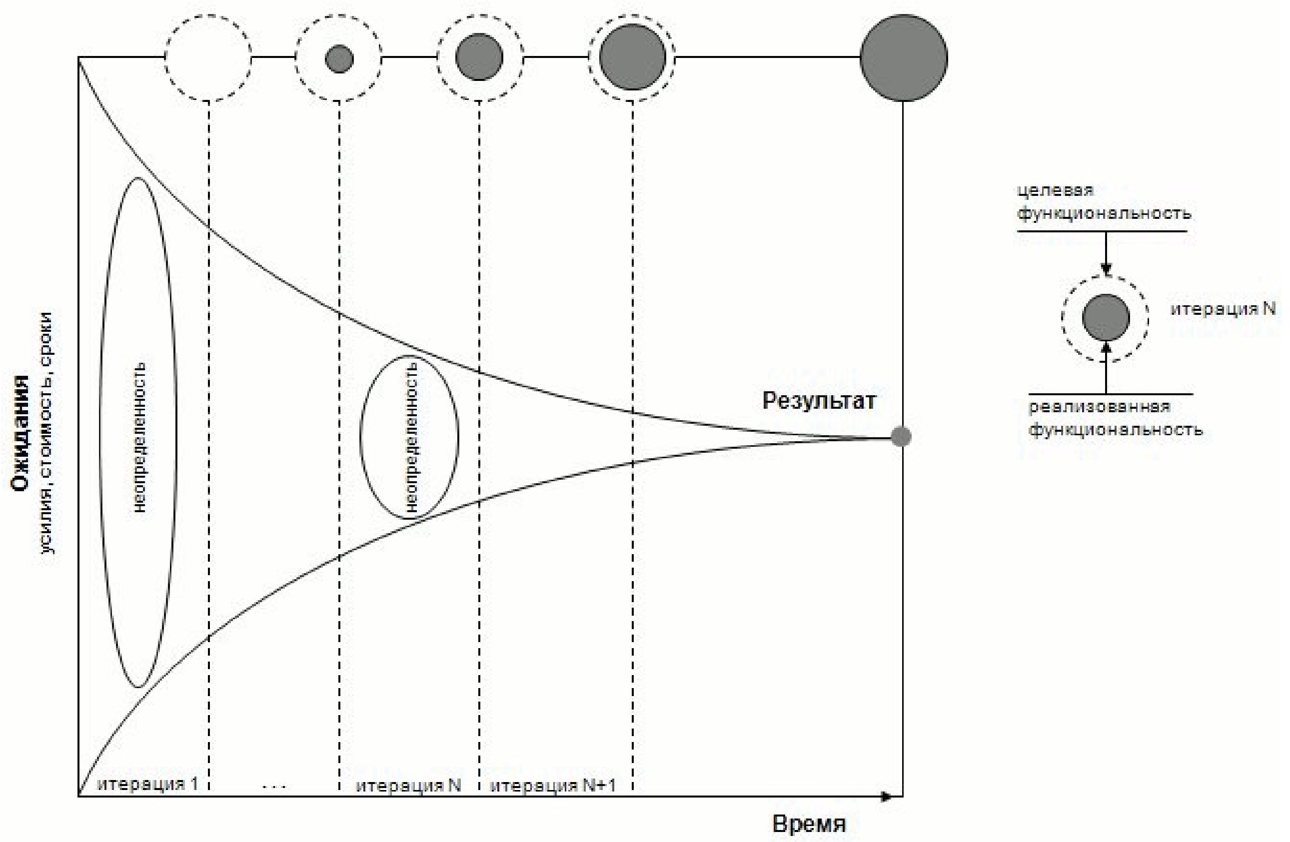

Рисунок 9: Снижение неопределенности и инкрементальное расширение функциональности при итеративной организация жизненного цикла.

Итерационный метод разработки

Каскадные методологии разработки программного обеспечения исходят из того, что разработка программного обеспечения делится на фазы, каждая из которых характеризуется своим набором работ.

Такой подход в среде быстро изменяющихся условий работы с заказчиком, стремительно меняющегося рынка и требований к разрабатываемому программному обеспечению оказывается неподходящим. Поэтому, в качестве развития и альтернативы каскадных подходов явилось создание группы подходов быстрой разработки.

Все эти подходы объединяют следующие основные черты:

1. итерационную разработку прототипа;
2. тесное взаимодействие с заказчиком.

Итерационный подход

Итеративный подход (iterative delivery) разбивает разработку на несколько итераций, в ходе каждой из которых выполняются практически все типы работ,

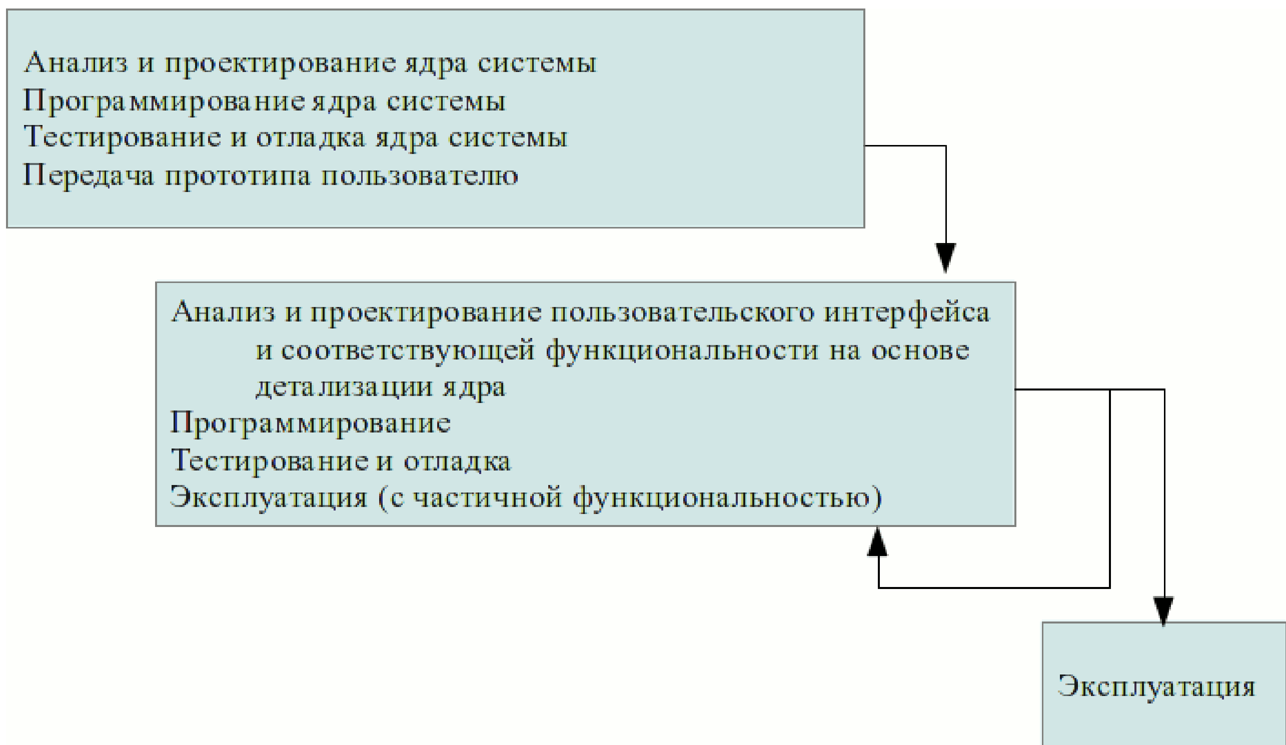

Рисунок 10: Итеративная разработка

и создается реальная работающая система с все более развитыми функциональными возможностями.

При каскадном подходе сначала происходит выявление всех требований к

проекту и их анализ. Затем проектная группа приступает к проектированию системы (чаще всего, "сверху вниз", разбив создаваемую систему на подсистемы и далее детализируя их до уровня программных процедур и функций). После этого начинается разработка кода и модульное тестирование. После этого идет сборка и системное тестирование.

При итерационном подходе разработка ПО разбивается на относительно короткие итерации. Практически во всех итерациях выполняется и выявление требований, и проектирование, и тестирование. Так, в самой первой итерации еще до выявления всех требований может начаться разработка прототипа, на котором проверяются основные архитектурные решения. По мере детализации требований на отдельные подсистемы или компоненты на последующих итерациях начинается их проектирование и кодирование. Разработанные "начерно" подсистемы и компоненты собираются в единую систему (не дожидаясь завершения разработки всех подсистем) и тут же начинается их системное тестирование.

Эволюционное прототипирование

Первый прототип при эволюционном прототипировании (evolutionary prototyping) обычно включает создание развитого пользовательского интерфейса. Он может быть сразу же продемонстрирован заказчику для получения от него отзывов и возможных корректив. Основное начальное внимание уделяется стороне системы, обращенной к пользователю. Далее, до тез пор, пока пользователь не сочтет программный продукт законченным, в него вносится необходимая функциональность.

Эволюционное прототипирование разумно применять в тех случаях, когда заказчик не может четко сформулировать свои требования к программному продукту на начальных этапах разработки или заказчик знает, что требования могут кардинально измениться.

Существенным недостатком этого подхода является невозможность определить продолжительность и стоимость проекта. Не является очевидным и количество итераций, по истечении которых заказчик сочтет программный продукт законченным.

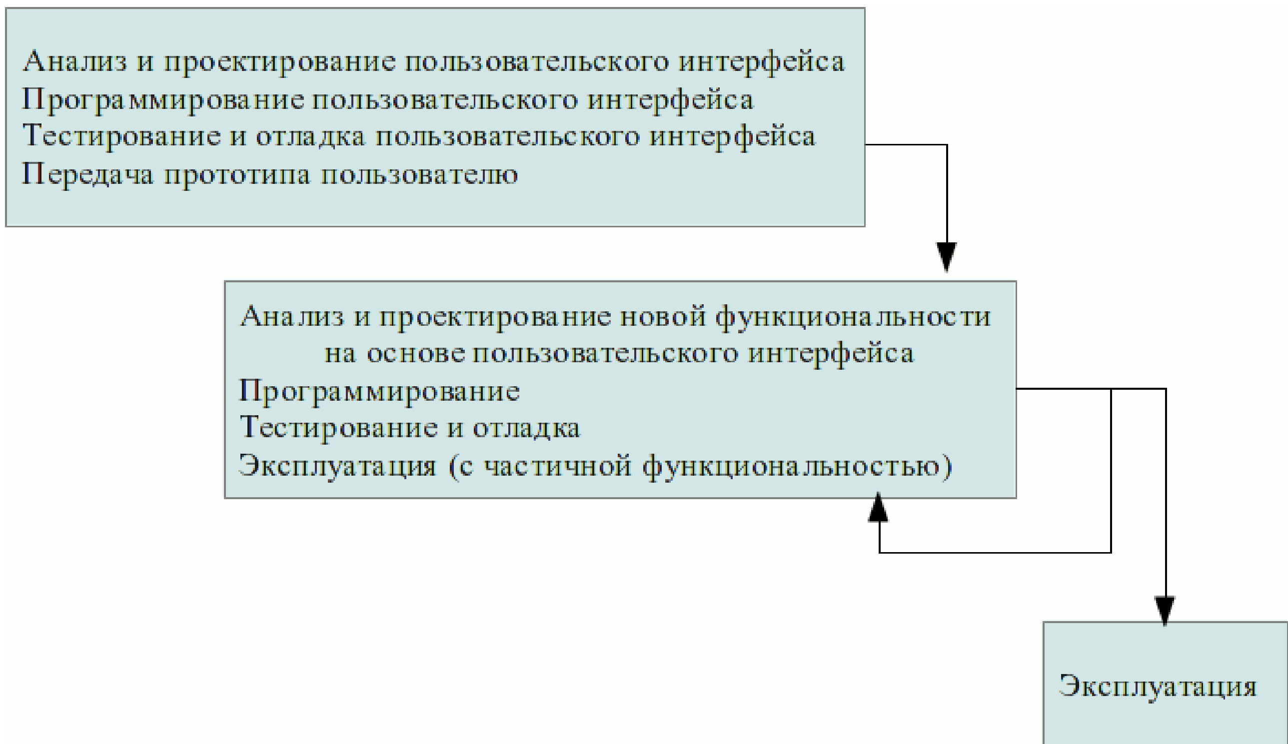

Рисунок 11: Эволюционное прототипирование

Гибкие методологии разработки

Гибкая методология разработки (agile software development) — набор подходов к разработке программного обеспечения, основанных на итеративной разработке и динамически формируемых требования, а также обеспечении их реализации благодаря взаимодействию внутри самоорганизующихся рабочих групп, состоящих из специалистов различного профиля.

Большинство гибких методологий нацелены на минимизацию рисков путем сведения разработки к серии итераций, длящихся обычно две-три недели. Каждая итерация содержит в себе все основные технологические процессы, такие как планирование, анализ требований, проектирование, кодирование, тестирование и документирование.

Гибкие методологии подразумевают, что программный продукт готов к выпуску в конце каждой итерации. И по завершению каждой итерации команда выполняет переоценку приоритетов разработки.

Особенностью гибких методов разработки является то, что основной упор делается на непосредственное общение лицом к лицу. Чаще всего члены agile-команды расположены в одном офисе, где присутствует и заказчик или его полномочный представитель, определяющий требования к продукту. Роль заказчика может выполнять менеджер проекта, бизнес-аналитик или непосредственно клиент. Офис может также включать тестировщиков, дизайнеров интерфейса, технических писателей и менеджеров.

В феврале 2001 в штате Юта США был выпущен «Манифест гибкой методологии разработки программного обеспечения». Он являлся альтернативой управляемым документацией, «тяжеловесным» практикам разработки программного обеспечения, таким как водопадная модель, являвшимся золотым стандартом разработки в то время. Данный манифест был одобрен и подписан представителями методологий экстремального программирования, Crystal Clear, DSDM, Feature driven development, Scrum, Adaptive software development, Pragmatic Programming. Гибкая методология разработки использовалась многими компаниями и до принятия манифеста, однако именно после этого события произошло вхождение agile-разработки в массы.

Часто к недостаткам гибких подходов относят следующий факт: при agile-методологии часто пренебрегают созданием «дорожной карты» развития продукта, равно как и управлением требованиями, в процессе которого и

формируется такая «карта». Гибкий подход к управлению требованиями не подразумевает далеко идущих планов (по сути, управления требованиями просто не существует в данной методологии), а подразумевает возможность заказчика вдруг и неожиданно в конце каждой итерации выставлять новые требования, часто противоречащие архитектуре уже созданного и поставляемого продукта. Такое иногда приводит к катастрофическим «авралам» с массовым рефакторингом и переделками практически на каждой очередной итерации.

Кроме того, считается, что работа в рамках гибкой методологии мотивирует разработчиков решать все поступившие задачи простейшим и быстрейшим возможным способом, при этом зачастую не обращая внимания на правильность кода с точки зрения требований нижележащей платформы (подход — «работает, и ладно», при этом не учитывается, что может перестать работать при малейшем изменении или же дать тяжелые к воспроизводству дефекты после реального развертывания у заказчика). Это приводит к снижению качества продукта и накоплению дефектов.

Экстремальное программирование

Экстремальное программирование — это гибкий подход, предложенный Кентом Беком, Мартином Фаулером и другими.

В основе этого подхода лежат 12 приемов, соблюдение которых необходимо для достижения эффективного результата. Эти приемы обычно объединяют в 4 группы:

Двенадцать основных приёмов экстремального программирования (по первому изданию книги *Extreme programming explained*) могут быть объединены в четыре группы:

- Короткий цикл обратной связи (Fine scale feedback)
  - Разработка через тестирование (Test driven development)
  - Игра в планирование (Planning game)
  - Заказчик всегда рядом (Whole team, Onsite customer)
  - Парное программирование (Pair programming)
- Непрерывный, а не пакетный процесс
  - Непрерывная интеграция (Continuous Integration)
  - Рефакторинг (Design Improvement, Refactor)
  - Частые небольшие релизы (Small Releases)
- Понимание, разделяемое всеми
  - Простота (Simple design)
  - Метафора системы (System metaphor)
  - Коллективное владение кодом (Collective code ownership) или выбранными шаблонами проектирования (Collective patterns ownership)
  - Стандарт кодирования (Coding standard or Coding conventions)
- Социальная защищенность программиста (Programmer welfare):
  - 40-часовая рабочая неделя (Sustainable pace, Forty hour week)

SCRUM

SRUM — гибкая методология, особо акцентирующая внимание на контроле процесса разработки.

Подход впервые описали Хиротака Такэути и Икудзиро Нонака в статье The New Product Development Game (Гарвардский Деловой Обзор, январь-февраль 1986). Они отметили, что проекты, над которыми работают небольшие, кросс-функциональные команды, обычно систематически производят лучшие результаты, и объяснили это как «подход регби». В 1991 году ДеГрейс и Шталь в книге «Злые проблемы, справедливые решения» ссылались на этот подход, как на Scrum (толкотня; схватка вокруг мяча (в регби)), спортивный термин, приведенный в статье Такэути и Нонакой. Позднее Кен Швабер объединил усилия с Майком Бидлом в 2001 году, чтобы детально описать метод в книге «Agile Software Development with SCRUM».

В основе этого подхода лежит набор принципов, на которых строится разработка. Эти принципы позволяют в жёстко фиксированные и небольшие по времени итерации, называемые спринтами (sprints), предоставлять конечному пользователю работающее программное обеспечение с новыми возможностями, для которых определен наибольший приоритет. Спринт — это жестко фиксированный промежуток времени, длиной обычно 2-4, иногда 6 недель. Считается, что чем короче спринт, тем более гибким является процесс разработки, релизы выходят чаще, быстрее поступают отзывы от потребителя, меньше времени тратится на работу в неправильном направлении.

Во общем случае процесс можно описать следующим способом:

1. в начале работы над проектом составляется набор требуемой фукнциональности — т.н. резерв проекта (product backlog), производится первичная оценка сроков выполнения проекта;
2. инициируется спринт, при этом заказчик составляет функциональность для данного временного промежутка из резерва проекта; выбранная функциональность разбита по задачам, каждая из которых оценивается скрам-командой; выбранная функциональность не может меняться в течении спринта;
3. скрам-команды проводят совещания, на которых оценивают объем работы, который необходимо совершить для завершения текущего спринта. После проведения этого совещания может проводиться совещание между разными скрам-командами для обеспечения связи

между компонентами.

Правила проведения совещаний строго регламентированы, внутри команд выстраивается строгая система ролей. По завершению каждого спринта производится строгая оценка результатов, составляются диаграммы и отчеты по ходу развития проекта, выставляются пожелания скрам-команд на следующий спринт.

LEAN

Бережливая разработка программного обеспечения — методология разработки программного обеспечения, использующая методы концепции бережливого производства. Возникла из среды сторонников концепции гибкой методологии разработки. Впервые была освещена в одноименной книге (Lean Software Development) Мэри Поппендик и Тома Поппендика. В книге представлены традиционные принципы бережливого производства применительно к разработке программного обеспечения, также набор из 22 инструментов (практик) и их сравнение с гибкой методологией разработки. Мэри и Том участвовали в ряде различных конференций, посвященных методике Agile, что объясняет известность концепции бережливого производства среди сообщества гибкой методология разработки.

Если говорить о Бережливом подходе вообще, то в его основе лежат следующие 10 принципов, раделенных на 4 группы:

Принципы бережливого производства, которые суммированы ниже, могут использоваться как основа или руководство для решения большинства проблем в мире разработки программного обеспечения:

- Создавать ценность и ничего более
    - 1. Устранять потери
    - 2. Сокращать уровень запасов
    - 3. Делать качественно с первого раза
- Акцент на тех, кто добавляет ценность
    - 4. Поддерживать (назначать, поощрять) тех, кто добавляет ценность
    - 5. Постоянно совершенствоваться
- Вытягивать ценность
    - 6. Соответствовать требованиям потребителей
    - 7. Вытягивать спросом (ориентируясь на спрос)
    - 8. Улучшать поток создания ценности
- Помогать окружающим в оптимизации
    - 9. Запрет локальной оптимизации
    - 10. Партнерство с поставщикам

Исходя из основных принципов Бережливого подхода в производстве были сформулированы следующие принципы:

- Исключение затрат.

  Затратами считается все, что не добавляет ценности для потребителя. В частности: излишняя функциональность; ожидание (паузы) в процессе разработки; нечёткие требования; бюрократизация; медленное внутреннее сообщение.

- Акцент на обучении.

  Короткие циклы разработки, раннее тестирование, частая обратная связь с заказчиком.

- Предельно отсроченное принятие решений.

  Решение следует принимать не на основе предположений и прогнозов, а после открытия существенных фактов.

- Предельно быстрая доставка заказчику.

  Короткие итерации разработки.

- Мотивация команды.

  Нельзя рассматривать людей исключительно как ресурс. Людям нужно нечто большее, чем просто список заданий.

- Интегрирование.

  Передать целостную информацию заказчику. Стремиться к целостной архитектуре. Рефакторинг.

- Целостное видение.

  Стандартизация, установление отношений между разработчиками. Разделение разработчиками принципов бережливости. «Мыслить широко, действовать мало, промахиваться быстро; учиться стремительно».

Test-Driven Development (TDD)

Разработка через тестирование - техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам. Кент Бек, считающийся изобретателем этой техники, утверждал в 2003 году, что разработка через тестирование поощряет простой дизайн и внушает уверенность (англ. inspires confidence).

В 1999 году при своём появлении разработка через тестирование была тесно связана с концепцией «сначала тест», test-first, применяемой в экстремальном программировании, однако позже выделилась как независимая методология.

Тест — это процедура, которая позволяет либо подтвердить, либо опровергнуть работоспособность кода. Когда программист проверяет работоспособность разработанного им кода, он выполняет тестирование вручную. В данном контексте тест состоит из двух этапов: стимулирование кода и проверка результатов его работы. Автоматический тест выполняется иначе: вместо программиста стимулированием кода и проверкой результатов занимается компьютер, который отображает на экране результат выполнения теста: код работоспособен или код неработоспособен. При этом происходит «инверсия ответственности»: от логики тестов и их качества зависит, будет ли код соответствовать техническому заданию. Методика разработки через тестирование заключается главным образом именно в организации автоматических тестов и выработке определенных навыков тестирования.

Разработка через тестирование требует от разработчика создания автоматизированных модульных тестов, определяющих требования к коду непосредственно перед написанием самого кода. Тест содержит проверки условий, которые могут либо выполняться, либо нет. Когда они выполняются, говорят, что тест пройден. Прохождение теста подтверждает поведение, предполагаемое программистом. Разработчики часто пользуются библиотеками для тестирования (testing frameworks) для создания и автоматизации запуска наборов тестов. На практике модульные тесты покрывают критические и нетривиальные участки кода. Это может быть код, который подвержен частым изменениям, код, от работы которого зависит работоспособность большого количества другого кода, или код с большим количеством зависимостей.

Среда разработки должна быстро реагировать на небольшие модификации кода. Архитектура программы должна базироваться на использовании множества сильно связанных компонентов, которые слабо сцеплены друг с другом, благодаря чему тестирование кода упрощается.

TDD не только предполагает проверку корректности, но и влияет на дизайн программы. Опираясь на тесты, разработчики могут быстрее представить, какая функциональность необходима пользователю. Таким образом, детали интерфейса появляются задолго до окончательной реализации решения. Разумеется, к тестам применяются те же требования стандартов кодирования, что и к основному коду.

Цикл разработки через тестирование по книге Кента Бека «Разработка через тестирование: на примере»:

- Добавление теста
- Запуск всех тестов: новые тесты не проходят
- Написание кода
- Запуск всех тестов: все тесты проходят
- Рефакторинг
- Повтор цикла

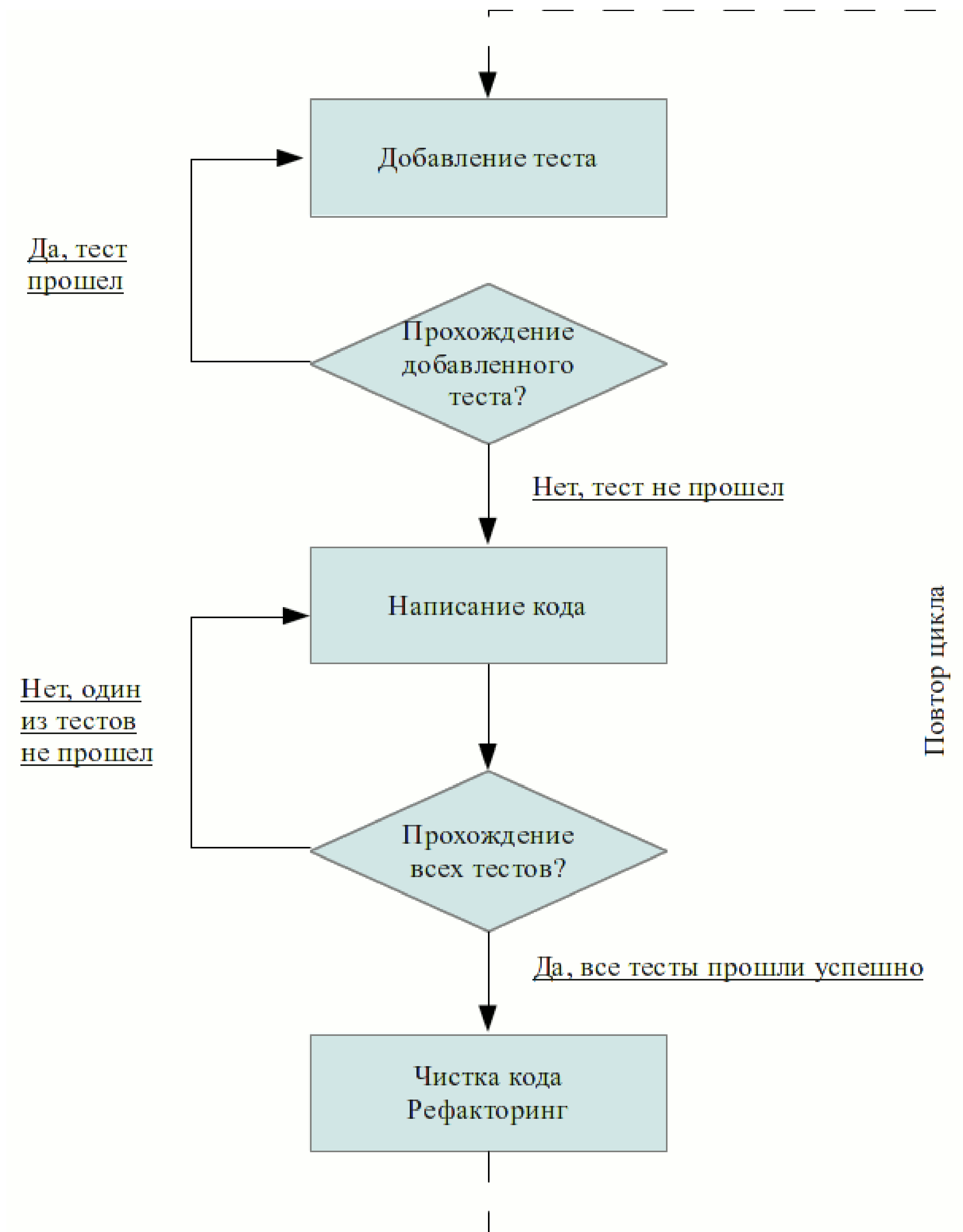

Рисунок 12: Цикл разработки функциональности через тестирование

Добавление теста

При разработке через тестирование, добавление каждой новой функциональности (feature) в программу начинается с написания теста. Неизбежно этот тест не будет проходить, поскольку соответствующий код еще не написан. (Если же написанный тест прошел, это означает, что либо предложенная «новая» функциональность уже существует, либо тест имеет недостатки). Чтобы написать тест, разработчик должен четко понимать предъявляемые к новой возможности требования. Для этого рассматриваются возможные сценарии использования и пользовательские истории. Новые требования могут также повлечь изменение существующих тестов. Это отличает разработку через тестирование от техник, когда тесты пишутся после того, как код уже написан: она заставляет разработчика сфокусироваться на требованиях до написания кода — тонкое, но важное отличие.

Запуск всех тестов: новые тесты не проходят

На этом этапе проверяют, что только что написанные тесты не проходят. Этот этап также проверяет сами тесты: написанный тест может проходить всегда и соответственно быть бесполезным. Новые тесты должны не проходить по объяснимым причинам. Это увеличит уверенность (хотя не будет гарантировать полностью), что тест действительно тестирует то, для чего он был разработан.

Написание кода

На этом этапе пишется новый код так, что тест будет проходить. Этот код не обязательно должен быть идеален. Допустимо, чтобы он проходил тест каким-то неэлегантным способом. Это приемлемо, поскольку последующие этапы улучшат и отполируют его.

Важно писать код, предназначенный именно для прохождения теста. Не следует добавлять лишней и, соответственно, не тестируемой функциональности.

Запуск всех тестов: все тесты проходят

Если все тесты проходят, программист может быть уверен, что код удовлетворяет всем тестируемым требованиям. После этого можно приступить к заключительному этапу цикла.

Рефакторинг

Когда достигнута требуемая функциональность, на этом этапе код может быть почищен. Рефакторинг — процесс изменения внутренней структуры программы, не затрагивающий ее внешнего поведения и имеющий целью облегчить понимание ее работы, устранить дублирование кода, облегчить внесение изменений в ближайшем будущем.

Повтор цикла

Описанный цикл повторяется, реализуя все новую и новую функциональность. Шаги следует делать небольшими, от 1 до 10 изменений между запусками тестов. Если новый код не удовлетворяет новым тестам или старые тесты перестают проходить, программист должен вернуться к отладке. При использовании сторонних библиотек не следует делать настолько небольшие изменения, которые буквально тестируют саму стороннюю библиотеку, а не код, её использующий, если только нет подозрений, что библиотека содержит ошибки.

Принципы

Разработка через тестирование тесно связана с такими принципами как KISS ("keep it simple, stupid") и YAGNI ("you ain't gonna need it"). Дизайн может быть чище и яснее, при написании лишь того кода, который необходим для прохождения теста. Кент Бек также предлагает принцип «подделай, пока не сделаешь» (fake it till you make it). Тесты должны писаться для тестируемой функциональности. Считается, что это имеет два преимущества. Это помогает убедиться, что приложение пригодно для тестирования, поскольку разработчику придется с самого начала обдумать то, как приложение будет тестироваться. Это также способствует тому, что тестами будет покрыта вся функциональность. Когда функциональность пишется до тестов, разработчики и организации склонны переходить к реализации следующей функциональности, не протестировав существующую.

Идея проверять, что вновь написанный тест не проходит, помогает убедиться, что тест реально что-то проверяет. Только после этой проверки следует приступать к реализации новой функциональности. Этот прием, известный как «красный/зеленый/рефакторинг», называют «мантрой разработки через тестирование». Под красным здесь понимают не прошедшие тесты, а под зеленым — прошедшие.

Отработанные практики разработки через тестирование привели к созданию техники «разработка через приемочное тестирование» (Acceptance Test-driven development, ATDD), в котором критерии описанные заказчиком автоматизируются в приемочные тесты, используемые потом в обычном процессе разработки через модульное тестирование (unit test-driven development, UTDD). Этот процесс позволяет гарантировать, что приложение удовлетворяет сформулированным требованиям. При разработке через приёмочное тестирование, команда разработчиков сконцентрирована на четкой задаче: удовлетворить приемочные тесты, которые отражают соответствующие требования пользователя.

Приемочные (функциональные) тесты (customer tests, acceptance tests) — тесты, проверяющие функциональность приложения на соответствие требованиям заказчика. Приемочные тесты проходят на стороне заказчика. Это помогает ему быть уверенным в том, что он получит всю необходимую функциональность.

К основным недостаткам данной методологии разработки относят:

- Главным недостатком TDD является то, что к нему сложно привыкнуть.
- Существуют задачи, которые невозможно (по крайней мере, на текущий момент) решить только при помощи тестов. В частности, TDD не позволяет механически продемонстрировать адекватность разработанного кода в области безопасности данных и взаимодействия между процессами. Безусловно, безопасность основана на коде, в котором не должно быть дефектов, однако она основана также на участии человека в процедурах защиты данных. Тонкие проблемы, возникающие в области взаимодействия между процессами, невозможно с уверенностью воспроизвести, просто запустив некоторый код.
- Разработку через тестирование сложно применять в тех случаях, когда для тестирования необходимо прохождение функциональных тестов. Примерами может быть: разработка интерфейсов пользователя, программ, работающих с базами данных, а также того, что зависит от специфической конфигурации сети. Разработка через тестирование не предполагает большого объёма работы по тестированию такого рода вещей. Она сосредотачивается на тестировании отдельно взятых модулей, используя mock-объекты для представления внешнего мира.
- Требуется больше времени на разработку и поддержку, а одобрение со стороны руководства очень важно. Если у организации нет уверенности в том, что разработка через тестирование улучшит качество продукта, то время, потраченное на написание тестов, может рассматриваться как потраченное впустую.
- Модульные тесты, создаваемые при разработке через тестирование, обычно пишутся теми же, кто пишет тестируемый код. Если разработчик неправильно истолковал требования к приложению, и тест, и тестируемый модуль будут содержать ошибку.
- Большое количество используемых тестов могут создать ложное ощущение надежности, приводящее к меньшему количеству действий по контролю качества.
- Тесты сами по себе являются источником накладных расходов. Плохо написанные тесты, например, содержат жёстко вшитые строки с сообщениями об ошибках или подвержены ошибкам, дороги при поддержке. Чтобы упростить поддержку тестов следует повторно использовать сообщения об ошибках из тестируемого кода.

- Уровень покрытия тестами, получаемый в результате разработки через тестирование, не может быть легко получен впоследствии. Исходные тесты становятся все более ценными с течением времени. Если неудачные архитектура, дизайн или стратегия тестирования приводят к большому количеству непройденных тестов, важно их все исправить в индивидуальном порядке. Простое удаление, отключение или поспешное изменение их может привести к необнаруживаемым пробелам в покрытии тестами.

Fake-, mock-объекты и интеграционные тесты

Модульные тесты тестируют каждый модуль по отдельности. Не важно, содержит ли модуль сотни тестов или только пять. Тесты, используемые при разработке через тестирование, не должны пересекать границы процесса, использовать сетевые соединения. В противном случае прохождение тестов будет занимать большое время, и разработчики будут реже запускать набор тестов целиком. Введение зависимости от внешних модулей или данных также превращает модульные тесты в интеграционные. При этом если один модуль в цепочке ведет себя неправильно, может быть не сразу понятно какой именно.

Когда разрабатываемый код использует базы данных, веб-сервисы или другие внешние процессы, имеет смысл выделить покрываемую тестированием часть. Это делается в два шага:

1. Везде, где требуется доступ к внешним ресурсам, должен быть объявлен интерфейс, через который этот доступ будет осуществляться. Здесь используется принцип инверсии зависимостей (dependency inversion) для обсуждения преимуществ этого подхода независимо от TDD.

2. Интерфейс должен иметь две реализации. Первая, собственно предоставляющая доступ к ресурсу, и вторая, являющаяся fake- или mock-объектом. Все, что делают fake-объекты, это добавляют сообщения вида «Объект person сохранен» в лог, чтобы потом проверить правильность поведения. Mock-объекты отличаются от fake- тем, что сами содержат утверждения (assertions), проверяющие поведение тестируемого кода. Методы fake- и mock-объектов, возвращающие данные, можно настроить так, чтобы они возвращали при тестировании одни и те же правдоподобные данные. Они могут эмулировать ошибки так, чтобы код обработки ошибок мог быть тщательно протестирован. Другими примерами fake-служб, полезными при разработке через тестирование, могут быть: служба кодирования, которая не кодирует данные, генератор случайных чисел, который всегда выдает единицу. Fake- или mock-реализации являются примерами внедрения зависимости (dependency injection).

Использование fake- и mock-объектов для представления внешнего мира приводит к тому, что настоящая база данных и другой внешний код не будут протестированы в результате процесса разработки через тестирование. Чтобы избежать ошибок, необходимы тесты реальных реализаций интерфейсов, описанных выше. Эти тесты могут быть отделены от остальных модульных тестов и реально являются интеграционными тестами. Их необходимо меньше,

чем модульных, и они могут запускаться реже. Тем не менее, чаще всего они реализуются используя те же библиотеки для тестирования (testing framework), что и модульные тесты.

Интеграционные тесты, которые изменяют данные в базе данных, должны откатывать состоянии базы данных к тому, которое было до запуска теста, даже если тест не прошёл. Для этого часто применяются следующие техники:

- Метод TearDown, присутствующий в большинстве библиотек для тестирования.
- try...catch...finally структуры обработки исключений, там где они доступны.
- Транзакции баз данных.
- Создание снимка (snapshot) базы данных перед запуском тестов и откат к нему после окончания тестирования.
- Сброс базы данных в чистое состояние перед тестом, а не после них. Это может быть удобно, если интересно посмотреть состояние базы данных, оставшееся после не прошедшего теста.

Feature driven development

Feature driven development (FDD, разработка, управляемая функциональностью) — итеративная методология разработки программного обеспечения, одна из гибких методологий разработки (agile). FDD представляет собой попытку объединить наиболее признанные в индустрии разработки программного обеспечения методики, принимающие за основу важную для заказчика функциональность (свойства) разрабатываемого программного обеспечения. Основной целью данной методологии является разработка реального, работающего программного обеспечения систематически, в поставленные сроки.

FDD включает в себя пять базовых видов деятельности:

- разработка общей модели;
- составление списка необходимых функций системы;
- планирование работы над каждой функцией;
- проектирование функции;
- реализация функции.

Первые два процесса относятся к началу проекта. Последние три осуществляются для каждой функции. Разработчики в FDD делятся на «хозяев классов» и «главных программистов». Главные программисты привлекают хозяев задействованных классов к работе над очередным свойством. Работа над проектом предполагает частые сборки и делится на итерации, каждая из которых предполагает реализацию определенного набора функций.

Разработка общей модели

Разработка начинается с высокоуровневого сквозного анализа широты решаемого круга задач и контекста системы. Далее для каждой моделируемой области делается более детальный сквозной анализ. Сквозные описания составляются в небольших группах и выносятся на дальнейшее обсуждение и экспертную оценку. Одна из предлагаемых моделей или их объединение становится моделью для конкретной области. Модели каждой области задач объединяются в общую итоговую модель, которая изменяется в ходе работы.

Составление списка возможностей (функций)

Информация, собранная при построении общей модели, используется для

составления списка функций. Это осуществляется разбиением областей (domain) на подобласти (предметные области, subject areas) с точки зрения функциональности. Каждая отдельная подобласть соответствует какому-либо бизнес-процессу, шаги которого становятся списком функций (свойств). В данном случае функции — это маленькие части понимаемых пользователем функций, представленных в виде «<действие> <результат> <объект>», например, «проверка пароля пользователя». Разработка каждой функции должна занимать не более 2 недель, иначе задачу необходимо разбить на несколько подзадач, каждая их которых сможет быть завершена за установленный двухнедельный срок.

План по свойствам (функциям)

После составления списка основных функций, наступает черед составления плана разработки программного обеспечения. Владение классами распределяется среди ведущих программистов путем упорядочивания и организации свойств (или наборов свойств) в классы.

Проектирование функций

Для каждого свойства создается проектировочный пакет. Ведущий программист выделяет небольшую группу свойств для разработки в течение двух недель. Вместе с разработчиками соответствующего класса ведущий программист составляет подробные диаграммы последовательности для каждого свойства, уточняя общую модель. Далее пишутся «болванки» классов и методов, и происходит критическое рассмотрение дизайна.

Реализация функции

После успешного рассмотрения дизайна, данная видимая клиенту функциональность реализуется до состояния готовности. Для каждого класса пишется программный код. После модульного тестирования каждого блока и проверки кода, завершенная функция включается в основной проект (build).

Так как функции малы, то их разработка — относительно легкая задача. Для мониторинга проекта по разработке ПО и предоставления точных данных о развитии проекта необходимо протоколировать разработку каждого свойства (функции). FDD выделяет шесть последовательных этапов для каждой функции (свойства). Первые три полностью завершаются в процессе проектирования, последние три — в процессе реализации. Для удобства контроля за

выполнением работ на каждом этапе показывается процент его готовности (выполнения).

FDD построен на основе набора передового опыта (набора наилучших практик), признанного в отрасли и полученного из инженерии программного обеспечения. Эти практические методы строятся с точки зрения важного для клиента функционала. Ниже дано краткое описание каждого метода:

- Объектное моделирование области. Объектное моделирование состоит из исследования и выяснения рамок предметной области решаемой задачи. Результатом является общий каркас, который можно в дальнейшем дополнять функциями.
- Разработка по функции. Любая функция, которая слишком сложна для разработки в течение двух недель, разбивается на меньшие подфункции до тех пор, пока каждая подзадача не может быть названа свойством (то есть, быть реализована за 2 недели). Это облегчает создание корректно работающих функций, расширение и модификацию системы.
- Индивидуальное владение классом (кодом). Означает, что каждый блок кода закреплен за конкретным владельцем-разработчиком. Владелец ответственен за согласованность, производительность и концептуальную целостность своих классов.
- Команда по разработке функций (свойств). Команда по разработке функций (свойств) — маленькая, динамически формируемая команда разработчиков, занимающаяся небольшой подзадачей. Позволяет нескольким разработчикам участвовать в дизайне свойства, а также оценивать дизайнерские решения перед выбором наилучшего.
- Проверка кода (code review) Проверки обеспечивают хорошее качество кода, в первую очередь путем выявления ошибок.
- Конфигурационное управление. Помогает с идентификацией исходного кода для всех функций (свойств), разработка которых завершена на текущий момент, и с протоколированием изменений, сделанных разработчиками классов.
- Регулярная сборка. Регулярная сборка гарантирует, что всегда есть продукт (система), которая может быть представлена заказчику, и помогает находить ошибки при объединении частей исходного кода на ранних этапах.

- Обозримость хода работ и результатов. Частые и точные отчеты о ходе выполнения работ на всех уровнях внутри и за пределами проекта о выполненной работе помогают менеджерам правильно руководить проектом.

Технологии коллективной разработки

Очевидно, что все разновидности разработок ПО зависит от количества участников и во общем случае может быть сведено к трем типам:

1. Авторская разработка
2. Коллективная разработка
3. Общинная разработка

Авторская разработка

Авторская разработка — принцип создания программных продуктов, при котором весь жизненный цикл разработки поддерживается одним программистом.

Этот принцип использовался в 70-80 годы. Наиболее известной разработкой является язык программирования Pascal, созданная Николаусом Виртом.

Коллективная разработка

Коллективная разработка — это принцип, в основе которого лежит разделение труда в ходе жизненного цикла ПО между ограниченным числом людей. Одним из самых важных вопросов является разделение труда — от равноправных соисполнителей проекта до сложной иерархической организационной структуры.

Равноправные исполнители

Обычно бригада равноправных исполнителей состоит из специалистов, занятых примерно подобными задачами в рамках одного проекта. Естественно, специализаций в рамках одной бригады может быть несколько:

1. инженер-разработчик;
2. технический писатель;
3. инженер тестирования;
4. инженер качества;
5. специалист по сопровождению продукта;
6. специалист по продажам продукта.

Тип работы определяет содержание и природу выполняемой работы.

Приведем список типов работ и областей специализации на основе классификацииКонгер [Conger 1994]:

- разработка приложений
  - программист;
  - специалист по инженерии ПО;
  - специалист по инженерии знаний;
- работа с прилоежениями
  - специалист по приложениям;
  - администратор базы данных;
- техническая поддержка:
  - системный администратор;
  - сетевой администратор;
  - администратор коммуникаций;
- обеспечение качества продукта
  - технический писатель;
  - инженер тестирования;
  - инженер качества;
- маркетинг
  - специалист по сопровождению продукта;
  - специалист по продажам;
- системное интегрирование
  - системный интегратор.

Особое внимание следует обратить на специализацию системного интегратора. Его задачи — предложить заказчику вариант решения его проблем, выбрав наиболее приемлемый по цене и технике, и реализовать его. Т.о., интегратор продает решения и несет ответственность за их реализацию. Он должен обладать знаниями из совершенно различных областей — прикладное и системное программное обеспечение, администрирование, сети, экономика, уметь разбираться в предметной области заказчика и многое другое.

Бригада главного программиста

Харланом Миллзом было предложено организовывать команды программистов, подобные хирургическим бригадам, где один участник занимается основной работой, а остальные оказывают ему всевозможную поддержку. Такой подход получил название бригады главного программиста (chief programmer team). В бригаду входит десять человек, выполняющих специализированные роли в команде:

- Главный программист — выполняет анализ, проектирование, создание и отладку кода, написание документации. Должен обладать большим опытом работы и обширными знаниями.
- Дублер — выполняет любую работу главного программиста, подстраховывает его, может заниматься написанием кода, но не несет ответственности за проект.
- Администратор — осуществляет контроль денежных средств, людей, помещений, ресурсов, контактирует с другими группами и руководством.
- Редактор — осуществляет переработку черновиков документации, занимается их оформлением и публикацией.
- Языковед — является знатоком языков программирования, может найти эффективные способы использования языков программирования для решения сложных задач.
- Инструментальщик — занимается разработкой утилит и скриптов, поддерживает основной инструментарий и оказывает по нему консультации.
- Отладчик — разрабатывает тесты и проводит тестирование программного обеспечения.
- Делопроизводитель — отвечает за регистрацию всех технических данных команды. В настоящее время функции делопроизводителя осуществляет автоматически репозиторий проекта.

Программирование в парах

Программирование в парах предусматривает разработку, при которой два человека в одно и то же время занимаются программированием одной задачи за одним компьютером, используя одну клавиатуру, одну мышь и один монитор. В каждой паре существуют две роли.

- Первый партнер решает задачу непосредственной реализации одного методов наилучшим образом. Именно в его руках находится клавиатура и мышь.
- Второй партнер решает стратегические задачи:
  - будет ли работать используемый подход в целом;
  - какими могут быть дополнительные тестовые случаи;
  - существуют ли способы упростить всю систему так, что текущая проблема просто исчезнет.

Состав пар обычно меняется динамически, возможно несколько раз в день.

Ядерная модель

Ядерная модель предполагает наличие первого исполнителя, олицетворяющего ядро команды и создающего прототип системы. На основе прототипа командой разработчиков создается программный продукт. Наиболее сложным действием здесь является передача работы от исполнителя прототипа к команде, которая будет доводить работу до состояния программного продукта.

Общинная модель разработки

Идеология общинной (“базарной”) модели разработки сформулирована в программной статье Эрика Раймонда (Eric Raymond) “Собор и Базар”. Общинная модель характеризуется тремя основными факторами.

1. Децентрализованность разработки. Не существует ограничения сверху на количество людей, принимающих участие в проекте. Как правило, разработки такого типа ведутся на базе сети Интернет и могут включать любого заинтересованного разработчика.
2. Разработка на базе открытых исходных текстов. По ним можно разобраться с сутью задачи и в любой момент подключиться к разработке.
3. Большое количество внешних тестеров (бета-тестеров), позволяющих быстро обнаруживать ошибки и проблемы в программе.

Эрик Раймонд сформулировал несколько уроков, которые позволяют лучше понять особенности общинной разработки:

- Каждая хорошая программа начинается с энтузиазма разработчика.
- Хорошие программисты знают, что можно написать, а великие — можно переписать.
- При правильном отношении интересная проблема найдет вас сама.
- Когда вы теряете интерес к программе, ваша последняя обязанность передать ее компетентному преемнику.
- Следует выпускать ранние и частые версии программ.
- Обнаружить проблему и исправить ее могут разные люди.
- Иногда использовать идеи пользователей лучше, чем свои.

Офшорное программирование

Офшорное программирование — это выполнение внутренних работ компании сторонними специалистами, вне ее офиса и, как правило, на территории другой страны. Эта разновидность коллективного программирования получила известность в начале 1990-х годов. Формы предоставления услуг офшорного программирования прошли следующие этапы эволюции:

- Аутстаффинг — использование программистов «поштучно» для конкретной работы под руководством менеджера заказчика.
- Аутсорсинг — передача исполнителю разработки отдельных модулей компонентов систем, полная сборка которых производится заказчиком.
- Полная разработка — выполнение проекта по полной разработке и внедрению системы.

Довольно широкое распространение офшорного программирования в настоящее время обусловлено состоянием мирового рынка заказного программного обеспечения. По очень приблизительным данным разработкой программного обеспечения в мире занято от 7 до 20 миллионов человек.

В России — от 5 до 10 тысяч. В мире существует огромный неудовлетворенный спрос на услуги профессионального программирования (например, в США дефицит профессиональных программистов составил в 2003 году около 1,5 миллионов). Следовательно, для некоторых компаний передача части работ исполнителям в другой стране обусловлена естественной необходимостью.

Поскольку стран, в которые могут быть переданы офшорные заказы, достаточно много, мы можем сформулировать для них три основных желательных условия.

- Оплата профессионалов ниже, чем в стране-заказчике.
- Присутствуют высокие стандарты образования и доступны технические эксперты.
- Доступны передовые технологии, повышение технической квалификации.

Легко видеть, основные страны, удовлетворяющие данным условиям, — Россия, Индия, Китай. Среди лидеров офшорного программирования уже находятся Уругвай и Израиль.

Некоторый идеальный портрет офшорного программиста и офшорной компании может выглядеть следующим образом:

- Высокое качество работы каждого сотрудника. В офшорных компаниях в первую очередь “люди решают все”. На уровне компании должны быть созданы системы отбора и оценки сотрудников, системы карьерного роста, обучения, мотивации, социальной защиты и решения многих других задач.
- Высокий уровень ведения проекта. В основе любого проекта лежат несколько ключевых процессов, грамотное выполнение которых должен обеспечить менеджер проекта.
- Оптимальная структура управления компанией. Такая структура подразумевает возможность быстрого формирования и переформирования команды, а также отсутствие жесткой и сложной иерархии.

Для России, конечно, положительных сторон в офшорном программировании много, и они достаточно хорошо известны. В первую очередь, это фундаментальная подготовка, позволяющая браться за большие проекты. Немаловажную роль может играть территориальная близость и близость культурных сред. С тоже время, среди слабых сторон можно выделить следующие:

- Небольшое количество профессионально подготовленных менеджеров, способных грамотно управлять программными проектами. Большинство компаний, передающих заказы, считает это основной проблемой, препятствующей резкому росту офшорного программирования в России.
- Недостаточное знание естественного языка той страны, из которой поступает работа. Последнее время ситуация начинает меняться к лучшему благодаря росту интереса к иностранным языкам в школах и их грамотному преподаванию.
- Большие сложности поездок в страну фирмы, передавшей заказ, известно много случаев отказа в выдаче виз по надуманным причина инженерам, выезжающим на короткий срок для осуществления консультаций и участия в совещаниях рабочих групп. Особенно “славятся” такой политикой США.
- Относительно высокая стоимость качественных телекоммуникационных услуг. Это может стать причиной дополнительных расходов.

- Низкий уровень сертификации на соответствие стандартам качеств Для ряда зарубежных компаний наличие такого сертификата у российской компании, в которую передается заказ, является обязательным.
- Повышенная осторожность зарубежных клиентов при взаимодействии с российскими компаниями. В ряде случаев эта осторожность обусловлена историческими и политическими причинами. В других случаях — предыдущим неудачным опытом, связанным, например, с традиционным российским упованием на «авось».

Обеспечение качества программного продукта

Понятие об обеспечении качества и тестировании. Историческое развитие представлений

Качество программного обеспечения

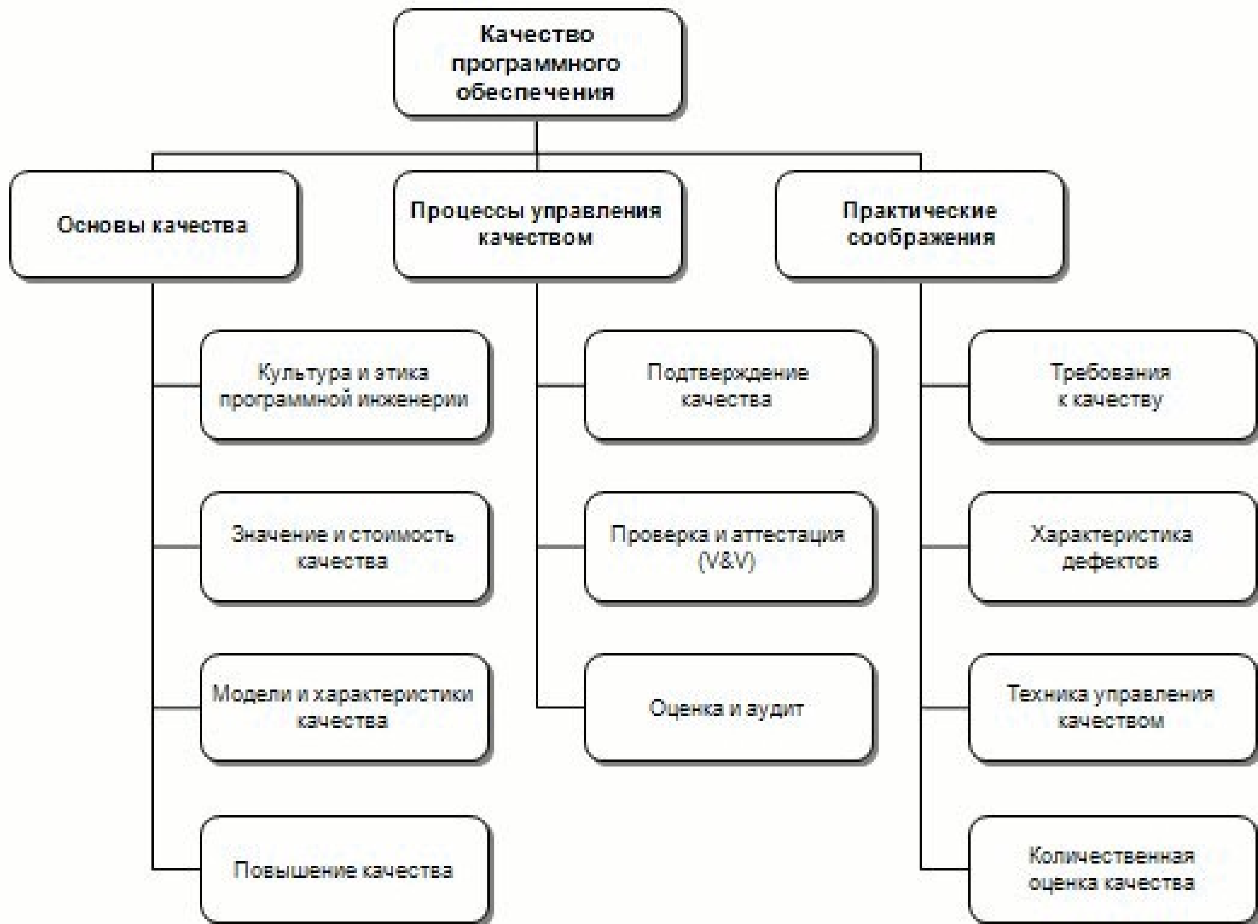

Рисунок 13: Качество программного обеспечения [SWEBOK 2004]

На протяжении многих лет отдельные авторы и целые организации определяли термин “качество” по-разному. Фил Кросби (Phil Crosby) в 1979 году дал определение качеству как «соответствие пользовательским требованиям». Уотс Хемпфри (Watts Hamphrey, оригинальный автор концепции модели оценки зрелости CMM, а также PSP и TSP — People Software Process и Team Software Process) описывает качество как «достижение отличного уровня пригодности к использованию». Компания IBM, в свою очередь, ввела в оборот фразу «качество, управляемое рыночными потребностями» («market-driven quality»). Критерий Бэлдриджа (Baldrige) для организационного качества (см. NIST - National Institute of Standards and Technology, «Baldrige National Quality Program», http://www.quality.nist.gov) использует похожую фразу - «качество,

задаваемое потребителем» («customer-driven quality»), рассматривая удовлетворение потребителя в качестве главного соображения в отношении качества. Чаще, понятие качества используется в соответствии с определением системы менеджмента качества ISO 9001 как «степень соответствия присущих характеристик требованиям» (именно так это сформулировано в официальном переводе ИСО 9000-2000 «Системы менеджмента качества. Основные положения и словарь»).Эти взгляды можно сформулировать как «Приемлемое качество», или «достаточное хорошее». Такое понятие может иметь место в связи с связи с тем, что качество определяется не только уровнем запросов конечных потребителей в отношении параметров создаваемого продукта, но и заданным контекстом/ограничениями проекта. Это не значит, что «приемлемое качество» противопоставляется «качеству, диктуемому заказчиком». Конечно, не стоит и проводить параллель «приемлемого качества» с «продуктом второй свежести». Введение категории «приемлемости» в отношении качества является лишь прагматичным взглядом на желаемую степень совершенства создаваемого продукта (услуги), способную удовлетворить пользователей и достижимую в рамках заданных проектных ограничений.

Говоря о качестве часто выделяют два подхода к критериям оценки качества:

1. Измерение направлено на продукт, либо на процесс. Для повышения качества программного обеспечения можно сосредоточиться на качестве самого продукта, например, делая упор на пользовательский интерфейс. Альтернативный подход заключается в совершенствовании процесса разработки, полагаю при этом, что чем лучше процесс, тем выше качество разрабатываемого продукта.
2. Иной подход к измерению связан либо с соответствием, либо с усовершенствованием. Под соответствием подразумевается соответствие какому-либо стандарту. Усовершенствование имеет своей целью переход на более совершенные методы и лучшую практику для повышения качества.

В качестве примера этих четырех подходов можно привести:

- ISO 9126 является стандартом на качество продукта, определяющим атрибуты и характеристики качества, включая измерения количественной оценки этих характеристик;
- примером «усовершенствования практики» является усовершенствование управления конфигурацией программного обеспечения, тестирования,

инспекций и т.п.;

- ISO 9000 является совокупностью стандартов, декларирующих требования для качественных систем.
- Методы усовершенствования процесса разработки ПО предлагают некоторую шкалу уровней и требования соответствия, согласно которым можно определить место компьютерной компании на этой шкале. Наибольшую популярность получили 2 метода:
    - модель зрелости процесса разработки ПО — Capability Maturity Model for Software (CMM), предложенная Software Engineering Institute (SEI)$
    - определение возможностей и улучшение процесса создания программного обеспечения. ISO/IEC 15504 Software Process Improvement and Capability determintaion (SPICE).

Если смотреть на вопрос качества программного обеспечения с точки зрения процесса, то можно утверждать, что качество начинается с удовлетворения потребностей разработчиков и доказывается удовлетворением потребностей пользователей.

Основные подходы к достижению качества таковы:

- качество достигается с помощью квалифицированных разработчиков, точного соблюдения процессов и удачных технологических подходов;
- качество достигается путем полного понимания всех действий и изменений. Но одна строки в программе не должна быть ни добавлена, ни изменена без полного понимания — что, зачем и как делается;
- качество достигается путем тщательного тестирования программы перед тем, как она будет доступна пользователю;
- достижение качества должно планироваться;
- достижение качества — обязанность каждого разработчика.

Характеристики качества программного обеспечения

Классическое определение качества заключается в том, что разработанный программный продукт подтверждает свою спецификацию, при этом спецификация должна быть ориентирована на характеристики, которые желательны заказчику.

Современные стандарты уточняют понятие качества, ввода совокупность черт и характеристик. Вот некоторые из них:

- Функциональность — способность программного продукта выполнять набор функций, удовлетворяющих заданным или подразумеваемым потребностям пользователей. Набор таких функций определяется во внешнем описании программного продукта. Характеристику функциональность еще называют пригодность, точность, интероперабельность, согласованность.
- Надежность — способность программы безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью. Надежный программный продукт не исключает наличия в нем ошибок — важно, чтобы ошибка при практическом применении в заданных условиях проявлялись достаточно редко. Степень надежности характеризуется вероятностью работы программного продукта без отказа в течении определенного периода времени. Существуют следующие подходы по обеспечению надежности:
    - предупреждение ошибок;
    - самообнаружение ошибок;
    - самоисправление ошибок;
    - обеспечение устойчивости к ошибкам.
- Удобство — это характеристики программного продукта, которые позволяют минимизировать усилия пользователя по подготовке исходных данных, применению программного продукта и оценке полученных результатов, а также вызывать положительные эмоции определенного или подразумеваемого пользователя. Удобство еще называют понимаемостью, эффективностью освоения, эргономичностью.
- Эффективность — это отношение уровня услуг, предоставляемых программным продуктом пользователю при заданных условиях к общему

использованных ресурсов.

- Сопровождаемость — характеристика, позволяющая минимизировать усилия по внесению изменений для устранения ошибок и его модификации в соответствии с изменяющимися потребностями пользователя.
- Переносимость — способность программного продукта быть перенесенным из одной среды в другую — например с одной аппаратной платформы на другую.
- Добротность — характеристика, по которой программа разумно и рационально организована, с достаточной продуманной организацией потоков управления и информационных потоков, не слишком переусложнена. Выделяют 4 класса критериев добротности программы:
  - Количественные критерии, связанные с различными метриками сложности программы.
    - Меры Холстеда, включающие рад формул, оценивающих длину, объем, уровень и интеллектуальное содержание программы.
    - Оценка сложности управляющего графа программы. Фрагмент программы может быть оценен цикломатическим числом ее управляющего графа, равное m-n+2, где m — число дуг, а n — число вершин управляющего графа. Считается, что цикломатическое число не должно превышать 10.
    - Оценка модульного разбиения программы. Такая оценка должна состоять из множества критериев, таких как , например, совокупность сложности определяемых процедур, сложности связей между модулями по импорту и экспорту определяемых сущностей.
  - Генетические критерии, связанные с происхождением программы и дисциплиной ее создания.
  - Структурные критерии, связанные с оценкой организации управления в программе и отражением организации управления в программном тексте.
  - Прагматические критерии, связанные с оценкой того, насколько программный текст соответствует цели программы. Формулируется список излишеств, которых не должно быть в добротных программах,

к примеру — вычислительной избыточности.

Качество процесса разработки

Оценку качества программного обеспечения можно через тестирование и эксплуатацию. В отличие от оценки качества процесса разработки, который должен стать частью долговременной стратегии компании.

CMM (Capability Maturity Model) — модель зрелости процесса разработки программного обеспечения определяет 5 уровней зрелости организации:

1. Начальный уровень. На этом уровне процесс разработки характеризуется практическим отсутствием процессов управления. Успех проекта зависит от индивидуальных усилий, личных качеств участников проекта.
2. Повторяемый уровень. Но этом уровне зрелости в компании должны быть внедрены основные процессы управления для отслеживания стоимости, графика проекта и его функциональности. Уровень характеризуется тем, что управление проектами основывается на накопленном опыте и ранее достигнутые успехи будут повторены на подобных приложениях.
3. Определенный уровень. Процесс разработки программного обеспечения (как на уровне управленческой, так и и инженерной деятельности) документирован, стандартизирован и интегрирован на уровне всей организации. Процесс перестает зависеть от индивидуальных качеств отдельных разработчиков и не может скатиться на более низкие уровни в кризисных ситуациях.
4. Управляемый уровень. В компании устанавливаются детальные количественные показатели на процесс разработки и качество продукта. И процесс разработки, и продукты — понимаемы и контролируемы.
5. Оптимизирующий уровень. Продолжающееся совершенствование процесса разработки на основе анализа текущих результатов процесса и применения инновационных идей и технологий.

Определение возможностей и улучшение процесса создания программного обеспечения

Данная модель очень близка к модели зрелости, однако уровни зрелостей (возможностей) могут быть применены не только к организации в целом, но и отдельным процессам. Модели такого типа называют непрерывными, в таких моделях один процесс может находиться на низком уровне зрелости, а другой — на высоком. Стандарт определяет 6 уровней зрелости процесса:

1. Уровень 0. Процесс не выполняется
2. Уровень 1. Выполняемый процесс.
3. Уровень 2. Управляемый процесс.
4. Уровень 3. Установленный процесс.
5. Уровень 4. Предсказуемый процесс.
6. Уровень 5. Оптимизирующий процесс.

Для оценки и улучшения качества процессов выполняются следующие задачи:

- Сравнение процесса разработки, существующего в данной организации с описанной в стандарте моделью. Результат анализа дает возможность определить сильные и слабые стороны процесса, его внутренние риски.
- Оценка возможности улучшения данного процесса на основе определения текущих возможностей.
- Техническая реализация поставленных задач на основе сформулированных целей совершенствования процесса.
- После этого весь цикл работ начинаю сначала.

Качество баз данных

Современные базы данных (БД) являются одними из массовых специфических объектов в сфере информатизации, для которых в ряде областей необходимо особенно высокое качество и его квалифицированное системное проектирование. Естественно возникают вопросы, что означает качество таких объектов, какие требования следует предъявлять к их качеству, какими характеристиками нужно описывать качество, как их задавать и оценивать. Для этого полезны, как прототипы, методы и стандарты, разработанные для анализа качества сложных программных средств.

Базу данных можно рассматривать как два компонента: систему программ управления данными и совокупность данных, упорядоченных по некоторым правилам. Поэтому при анализе качества базу данных целесообразно делить на два компонента:

- программные средства системы управления базой данных (СУБД), независимые от сферы их применения, структуры и смыслового содержания накапливаемых и обрабатываемых данных;
- информацию базы данных (ИБД), доступную для накопления, упорядочивания, обработки и использования в конкретной проблемно-ориентированной сфере применения.

При комплексном анализе качества баз данных, не всегда удается четко разделить требования и значения характеристик качества для каждого из этих объектов. При этом одна и та же система управления базой данных (СУБД) может обрабатывать различные по структуре, составу и содержанию данные, а одни и те же данные могут управляться программными средствами различных СУБД. Хотя эти компоненты тесно взаимодействуют при реализации конкретной прикладной БД, первоначально при проектировании они создаются или выбираются практически независимо и могут рассматриваться в их жизненном цикле (ЖЦ) как два объекта, которые различаются:

- номенклатурой и содержанием показателей качества, определяющих их назначение, функции и потребительские свойства;
- технологией и средствами автоматизации разработки и обеспечения всего ЖЦ каждого объекта;
- категориями специалистов, обеспечивающих: создание, эксплуатацию или применение компонентов БД;

- комплектами эксплуатационной и технологической документации, поддерживающими жизненный цикл объектов.

Первым компонентом для системного анализа и требований к качеству является комплекс программ СУБД. Практически весь набор характеристик и атрибутов качества ПС, изложенный в стандарте ISO- 9126, в той или иной степени, может использоваться при формировании требований к качеству СУБД. Особенности состоят в адаптации и изменении акцентов при выборе и упорядочении этих показателей. Во всех случаях важнейшими характеристиками качества СУБД являются требования функциональной пригодности для процессов формирования и изменения информационного наполнения БД администраторами, а также доступа к данным и представления результатов пользователям БД. Качество интерфейса специалистов с БД, обеспечиваемого средствами СУБД, определяется, в значительной степени, субъективно, однако имеется ряд характеристик, которые можно оценивать достаточно корректно.

Различия требований к характеристикам качества привели к созданию весьма широкого спектра локальных, специализированных и распределенных СУБД. Значения ряда показателей качества ПС, составляющих СУБД, существенно зависят от характеристик и организации информации в БД. Специализированные СУБД характеризуются относительно узкой сферой применения и более четким выделением группы требований к приоритетным показателям качества. В универсальных СУБД спектр характеристик качества шире, что позволяет соответственно расширять сферу применения конкретного типа СУБД. Однако и для них существуют области приоритетного, наиболее эффективного использования.

За основу принята номенклатура и содержание стандартизированных характеристик сложных комплексов программ, которые адаптируются применительно к понятиям и особенностям компонентов баз данных. В зависимости от конкретной проблемно-ориентированной области применения СУБД, приоритет при системном анализе требований качеству может отдаваться различным, конструктивным характеристикам: либо надежности и защищенности применения (финансовая сфера), либо удобству использования малоквалифицированными пользователями (социальная сфера), либо эффективности использования ресурсов (сфера материально-технического снабжения). Однако, практически во всех случаях сохраняется некоторая роль ряда других конструктивных показателей качества. Для каждого из них

необходимо анализировать и определять его приоритет для конкретной сферы применения, меры и шкалы необходимых и допустимых характеристик качества.

Вторым компонентом БД является собственно накапливаемая и обрабатываемая информация. В системах баз данных доминирующее значение приобретают сами данные, их хранение и обработка. Ниже сделан акцент на системный анализ требований и составляющих характеристик качества этого объекта - на информацию баз данных с предположением, что средства СУБД способны их обеспечить. Для оценивания качества информации БД может сохраняться общий, методический подход к выделению адекватной номенклатуры стандартизированных в ISO 9126 базовых характеристик и субхарактеристик качества ПС. Однако их содержание для применения к качеству ИБД при проектировании требуется уточнить и пояснить. Выделяемые показатели качества должны иметь практический интерес для пользователей БД и быть упорядочены в соответствии с приоритетами практического применения. Кроме того, каждый выделяемый показатель качества ИБД должен быть пригоден для достаточно достоверного оценивания или измерения, а также для сравнения с требуемым значением при испытаниях.

При проектировании каждой БД в контракте, техническом задании и в спецификации должны селектироваться и формализоваться представительный набор функциональных требований к качеству ИБД, адекватный ее назначению и области применения, а также требованиям заказчика и потенциальных пользователей. Так же как для ПС, характеристики качества ИБД можно разделить на функциональные и конструктивные. Их номенклатура, содержание и субхарактеристики ниже базируются на описаниях, рекомендуемых стандартом ISO 9126. Они представляются достаточно универсальными и применимыми для систематизации характеристик качества информации баз данных. Тем самым может быть заложена основа для стандартизированного формирования требований к качеству баз данных. Однако номенклатура показателей качества не всегда может ограничиваться только характеристиками информации в БД, а должна включать ряд уточнений, отражающих комплексную эффективность и функциональную пригодность совместного применения СУБД и ИБД пользователями в реальных условиях.

Функциональная пригодности ИБД при системном проектировании может представлять сложную проблему для определения соответствия требований реальным значениям необходимых атрибутов качества особенно, для больших распределенных БД при циркулировании разнообразной и сложной

информации об анализируемых объектах. Мерой качества функциональной пригодности может быть степень покрытия целей, назначения и функций БД доступной пользователям информацией. Так же как для ПС, для баз данных в составе функциональной пригодности целесообразно использовать группу субхарактеристик, определяющих функциональные и структурные требования к базам данных, основное содержание которых подобно приведенным для ПС выше. Дополнительно функциональная пригодность многих ИБД может отражаться:

- полнотой накопленных описаний объектов – относительным числом объектов или документов, имеющихся в БД, к общему числу объектов по данной тематике или по отношению к числу объектов в аналогичных БД того же назначения;
- идентичностью данных - относительным числом описаний объектов, не содержащих дефекты и ошибки, к общему числу документов об объектах в ИБД;
- актуальностью данных - относительным числом устаревших данных об объектах в ИБД к общему числу накопленных и обрабатываемых данных.

Разнообразие назначения и функций ИБД ограничивает возможность стандартизации требований к ним только общими правилами их организации и структурирования, которые достаточно подробно изложены выше и далее не рассматриваются.

К конструктивным характеристикам качества информации БД в целом можно отнести, с некоторой корректировкой и уточнением понятий, субхарактеристик и атрибутов, практически все стандартизированные показатели качества ПС, которые представлены в ISO 9126. Требования к информации баз данных так же должны содержать особенности обеспечения ее

- надежности,
- эффективности использования ресурсов,
- практичности,
- применимости,
- сопровождаемости,
- мобильности.

Содержание и атрибуты этих конструктивных характеристик в данном

случае несколько отличаются от применяемых для программ, однако, их сущность, как базовых понятий и характеристик качества объектов, целесообразно использовать при проектировании для систематизации и регламентированного формирования требований к этим компонентам информационных систем. Меры и шкалы для оценивания конструктивных характеристик, в значительной степени, могут применяться те же, что при анализе качества программных средств.

Корректность или достоверность данных - это степень соответствия информации об объектах в БД реальным объектам вне ЭВМ в данный момент времени, определяющаяся изменениями самих объектов, некорректностями записей о их состоянии или некорректностями расчетов их характеристик. При системном проектировании выбор и установление требований к корректности данных в БД, можно оценивать по степени покрытия накопленными, актуальными и достоверными данными состояния и изменения внешних объектов, которые они отражают. Кроме того, к корректности БД можно отнести некоторые объемно-временные характеристики сохраняемых и обрабатываемых данных:

- объем базы данных - относительное число записей описаний объектов или документов в базе данных, доступных для хранения и обработки, по сравнению с полным числом реальных объектов во внешней среде;
- оперативность - степень соответствия динамики изменения описаний данных в процессе сбора и обработки, состояниям реальных объектов или величина допустимого запаздывания между появлением или изменением характеристик реального объекта, относительно его отражения в базе данных;
- глубина ретроспективы - максимальный интервал времени от даты выпуска и/или записи в базу данных самого раннего документа до настоящего времени;
- динамичность - относительное число изменяемых описаний объектов к общему числу записей в БД за некоторый интервал времени, определяемый периодичностью издания версий БД.

Защищенность информации БД реализуется, в основном, программными средствами СУБД, однако в сочетании с поддерживающими их средствами организации и защиты данных. Цели, назначение и функции защиты тесно связаны с особенностями функциональной пригодности каждой ИБД. При

проектировании свойства защищать информацию баз данных от негативных воздействий описываются обычно составом и номенклатурой методов и средств, используемых для защиты от внешних и внутренних угроз. Косвенным показателем ее качества может служить относительная доля вычислительных ресурсов, используемых непосредственно средствами защиты информации БД.

Основное внимание в практике обеспечения безопасности применения БД сосредоточено на защите от злоумышленных разрушений, искажений и хищений информации баз данных. Основой такой защиты является аудит санкционирования доступа, а также контроль организации и эффективности ограничений доступа. В реальных БД возможны и не всегда учитываются катастрофические последствия и аномалии информации БД, отражающиеся на безопасности применения, при которых их источниками являются случайные, непредсказуемые, дестабилизирующие факторы или дефекты, и отсутствуют непосредственно заинтересованные лица в подобных нарушениях. Качество защиты ИБД можно характеризовать величиной предотвращенного ущерба, возможного при проявлении дестабилизирующих факторов и реализации конкретных угроз безопасности, а также средним временем между возможными проявлениями угроз, преодолевающих защиту данных.

Надежность информации баз данных может основываться на применении при системном проектировании понятий и методов теории надежности, которая позволяет получить ряд четких, измеряемых интегральных показателей их качества. Надежная ИБД, прежде всего, должна обеспечивать достаточно низкую вероятность потери работоспособности - отказа, в процессе ее функционирования в реальном времени. Быстрое реагирование на потерю или искажение данных и восстановление их достоверности и работоспособности за время меньшее, чем порог между сбоем и отказом, обеспечивают высокую надежность БД. Если в этих ситуациях происходит достаточно быстрое восстановление, такое что не фиксируется отказ, то такие события не влияют на основные показатели надежности – наработку на отказ и коэффициент готовности ИБД. Непредсказуемость вида, места и времени проявления дефектов ИБД в процессе эксплуатации приводит необходимости создания специальных, дополнительных систем оперативной защиты от непредумышленных, случайных искажений данных. Надежность должна повышаться за счет средств обеспечения помехоустойчивости, оперативного контроля и восстановления ИБД.

Стандартом ISO 9126 рекомендуется анализировать и учитывать надежность комплексов программ четырьмя субхарактеристиками, которые

могут быть применены также для формирования требований к характеристикам качества информации БД. Завершенность - свойство ИБД, состоящее в способности не попадать в состояния отказов вследствие потерь, искажений, ошибок и дефектов в данных. Они могут быть обусловлены не полным тестовым покрытием при испытаниях компонентов и ИБД в целом, а также недостаточной завершенностью их тестирования и защищенностью от искажений.

Устойчивость к дефектам и ошибкам - свойство ИБД автоматически поддерживать заданный уровень качества данных в случаях проявления дефектов и ошибок или нарушения установленного интерфейса по данным с внешней средой. Для этого в ИБД рекомендуется вводить оперативное обнаружение дефектов и ошибок информации, их идентификацию и автоматическое восстановление (рестарт) нормального функционирования ИБД. Относительная доля вычислительных ресурсов, используемых непосредственно для быстрой ликвидации последствий отказов и оперативного восстановления данных (рестарт) определяет значение устойчивости и снижается на повышении надежности ИБД.

Восстанавливаемость - свойство ИБД в случае отказа возобновлять требуемый уровень качества информации, а также корректировать поврежденные данные. Для этого необходимы вычислительные ресурсы и время на выявление неработоспособного состояния, диагностику причин отказа, а также на реализацию процессов восстановления. Основными показателями процесса восстановления данных являются его длительность и вероятностные характеристики ИБД в процессе ручного или автоматического их перезапуска — рестарта.

Доступность или готовность - свойство ИБД быть в состоянии полностью выполнять требуемую функцию в данный момент времени при заданных условиях использования информации базы данных. Обобщением характеристик отказов и восстановления производится в критерии коэффициент готовности ИБД. Этот показатель отражает вероятность иметь восстанавливаемые данные в работоспособном состоянии в произвольный момент времени. Нижние границы шкал атрибутов надежности могут быть отражены значениями, при которых использование конкретной ИБД становится неудобным, опасным или нерентабельным.

Эффективность использования ресурсов ЭВМ при системном анализе реального функционирования БД отражается временными характеристиками

взаимодействия конечных пользователей и администраторов ИБД в процессе эксплуатации базы данных по прямому назначению.

Временная эффективность БД определяется длительностью выполнения заданных функций и ожидания результатов от ИБД в средних и/или наихудших случаях, с учетом приоритетов задач. Она зависит от объема, структуры и скорости обработки данных, влияющих непосредственно на интервал времени завершения конкретного вычислительного процесса, и от пропускной способности - производительности, т.е. от числа заданий, которое можно реализовать на данной ЭВМ в заданном интервале времени.

Используемость ресурсов или ресурсная экономичность в стандартах отражается занятостью ресурсов центрального процессора, оперативной, внешней и виртуальной памяти, каналов ввода-вывода, терминалов и каналов сетей связи. Эта величина определяется структурой, функциями и объемом ИБД, а также архитектурными особенностями и доступными ресурсами ЭВМ. В зависимости от конкретных задач и особенностей ИБД и ЭВМ при системном проектировании и выборе атрибутов качества ИБД может доминировать либо абсолютная величина занятости ресурсов различных видов, либо относительная величина использования ресурсов каждого вида при нормальном функционировании ИБД.

Практичность-применимость - зачастую значительно определяет функциональную пригодность и полезность применения ИБД для квалифицированных пользователей. В число пользователей могут быть включены администраторы, конечные и косвенные пользователи, которые находятся под влиянием или зависят от качества информации БД. В эту группу показателей качества входят субхарактеристики и атрибуты с различных сторон отражающие функциональную понятность, удобство освоения, системную эффективность и простоту использования данных. Некоторые субхарактеристики можно оценивать экономическими показателями - затратами труда и времени специалистов на реализацию некоторых функций взаимодействия с данными. Практичность зависит не только от собственных характеристик ИБД, но также от организации и адекватности документирования процессов их эксплуатации.

Понятность зависит от качества документации и субъективных впечатлений потенциальных пользователей от функций и характеристик ИБД. В системном проекте ее можно представить качественно четкостью функциональной концепции, широтой демонстрационных возможностей,

полнотой, комплектностью и наглядностью представления в эксплуатационной документации возможных функций и особенностей реализации данных в БД. Она должна обеспечиваться корректностью и полнотой описания исходной и результирующей информации, а также всех деталей применения ИБД для пользователей.

Простота использования ИБД - возможность удобно и комфортно ее эксплуатировать и управлять данными. Для этого должны быть обеспечены: достаточный объем параметров управления, реализуемых по умолчанию, информативность сообщений пользователям, наглядность унифицированность управления экраном, а также доступность изменений функций ИБД в соответствии с квалификацией пользователей и минимум операций, необходимых для реализации определенного задания и анализа результатов. Некоторые атрибуты этой субхарактеристики доступны при установлении количественных требований путем указания трудоемкости длительности соответствующих процессов подготовки и обучения квалифицированных пользователей к эффективной эксплуатации ИБД.

Изучаемость может определяться требованиями ограниченной трудоемкости и длительности подготовки пользователя к полноценной эксплуатации информации БД. Изучаемость ИБД зависит от внутренних свойств и сложности структуры информации БД, а также от субъективных характеристик квалификации конкретных пользователей. Она может также характеризоваться объемом эксплуатационной документации и/или объемом и качеством электронных учебников.

Сопровождаемость информации БД в системном проекте может отражаться удобством и эффективностью исправления, усовершенствования или адаптации структуры и содержания описаний данных в зависимостисти от изменений во внешней среде применения, а также в требованиях и функциональных спецификациях заказчика. Обобщенно качество сопровождаемости ИБД можно представить потребностью трудовых и временных ресурсов для ее обеспечения и для реализации. Возможные затраты экономических, трудовых и временных ресурсов на развитие и совершенствование качества ИБД зависят не только от внутренних свойств данных, но также и от запросов и потребностей пользователей на новые функции и от готовности заказчика и разработчика удовлетворить эти потребности. По объему предполагаемых изменений, а также вновь вводимых в очередную версию данных с учетом сложности и новизны их разработки могут быть сформулированы требования на их реализацию.

Совокупность субхарактеристик сопровождаемости ПС, представленная в стандарте ISO 9126, вполне применима для описания требований к этому показателю качества информации БД, в основном, теми же организационно-технологическими субхарактеристиками. Анализируемость ИБД зависит от стройности архитектуры, унифицированности интерфейса, полноты и корректности технологической и эксплуатационной документации на БД. Изменяемость состоит в приспособленности структуры и содержания данных к реализации специфицированных изменений и к управлению конфигурацией данных. Изменяемость зависит не только от внутренних свойств ИБД, но также от организации и инструментальной оснащенности процессов сопровождения и конфигурационного управления, на которые ориентирована в проекте архитектура, внешние и внутренние интерфейсы данных.

Тестируемость зависит от величины области влияния изменений, которые необходимо тестировать при модификациях структуры и содержания данных в ИБД, от сложности тестов для проверки их характеристик. Ее атрибуты зависят от четкости формализации в системном проекте правил структурного построения компонентов и всего комплекса ИБД, от унификации межмодульных и внешних интерфейсов, от полноты и корректности технологической документации. Субхарактеристики изменяемость и тестируемость данных доступны количественному определению по величине трудоемкости и длительности реализации этих функций при типовых операциях с данными при применении различных методов и средств автоматизации.

Мобильность данных БД, так же как для программ, можно характеризовать в системном проекте в основном длительностью и трудоемкостью их инсталляции, адаптации и замещаемости при переносе ИБД на иные аппаратные и операционные платформы. Информация о процессах, происходящих во внешней среде, может иметь большие объем и трудоемкость первичного накопления и актуализации, что определяет необходимость ее тщательного хранения и регламентированного изменения. Формирование и заполнение информацией баз данных достаточно сложный и трудоемкий процесс, технико-экономические показатели которого сильно зависит от структуры, состава, объема, связности и других характеристик исходной информации. Особенности и трудоемкость переноса ИБД зависят, прежде всего, от характеристик совместимости архитектуры и содержания информации переносимой БД между рассматриваемыми платформами:

- форматная совместимость характеризуется степенью соответствия данных в ИБД анализируемых платформ требованиям стандартов на форматы представления данных для документальных, фактографических, словарных и иных БД;
- лингвистическая совместимость определяется степенью использования в рассматриваемых ИБД единых лингвистических средств (классификаторов, рубрикаторов, словарей), формализованных соответствующими стандартами этих платформ;
- физическая совместимость заключается в степени соответствия кодировки информации БД одинаковым стандартам на машиночитаемые носители информации.

Так как перенос БД часто обусловлен необходимостью увеличения ресурсов ЭВМ, доступных для решения новых перспективных задач, их системный проект становится естественным расширением функций ИБД относительно исходной версии проекта. Для оценки качества и определения требований к мобильности ИБД, так же как для ПС, следует решать задачу сравнения достигаемого эффекта и затрат для методов переноса или повторной разработки компонентов и наполнения базы данных в конкретных условиях с учетом всех перечисленных факторов и затрат. Эти задачи значительно упрощаются при одновременном сокращении затрат при применении идеологии и концепции открытых компьютерных систем, поддержанных комплексом международных стандартов, а также современных версий ОС и СУБД, как стандартов де-факто.

Тестирование программного обеспечения

Тестирование (software testing) — деятельность, выполняемая для оценки и улучшения качества программного обеспечения. Эта деятельность, в общем случае, базируется на обнаружении дефектов и проблем в программных системах.

Тестирование программных систем состоит из динамической верификации поведения программ на конечном (ограниченном) наборе тестов (set of test cases), выбранных соответствующим образом из обычно выполняемых действий прикладной области и обеспечивающих проверку соответствия ожидаемому поведению системы.

В данном определении тестирования выделены слова, определяющие основные вопросы, которым адресуется данная область знаний:

- Динамичность (dynamic) — этот термин подразумевает, что тестирование

всегда предполагает выполнение тестируемой программы с заданными входными данными. При этом, величины, задаваемые на вход тестируемому программному обеспечению, не всегда достаточны для определения теста. Сложность и недетерминированность систем приводит к тому, что система может по разному реагировать на одни и те же входные параметры, в зависимости от состояния системы. С точки зрения процесса тестирования термин «вход» (input) обычно используется в рамках соглашения о том, что вход может также специфицировать состояние системы, в тех случаях, когда это необходимо. Кроме динамических техник проверки качества, то есть тестирования, существуют также и статические техники, рассматриваемые в рамках оценки качества программного обеспечения.

- Конечность (ограниченность, finite) — даже для простых программ теоретически возможно столь большое количество тестовых сценариев, что исчерпывающее тестирование может занять многие месяцы и даже годы. Именно поэтому, с практической точки зрения, всестороннее тестирование считается бесконечным. Тестирование всегда предполагает компромисс между ограниченными ресурсами и заданными сроками, с одной стороны, и практически неограниченными требованиями по тестированию, с другой. То есть мы снова говорим об определении характеристик «приемлемого» качества, на основе которых планируем необходимы объем тестирования.

- Выбор (selection) — многие предлагаемые техники тестирования отличаются друг от друга в том, как выбираются сценарии тестирования. Инженеры по программному обеспечению должны обладать представлением о том, что различные критерии выбора тестов могут давать разные результаты, с точки зрения эффективности тестирования. Определение подходящего набора тестов для заданных условий является очень сложной проблемой. Обычно, для выбора соответствующих тестов совместно применяют техники анализа рисков, анализ требований и соответствующую экспертизу в области тестирования и заданной прикладной области.

- Ожидаемое поведение (expected behaviour) — хотя это не всегда легко, все же необходимо решить, какое наблюдаемое поведение программы будет приемлемо, а какое – нет. В противном случае, усилия по тестированию – бесполезны. Наблюдаемое поведение может рассматриваться в контексте

пользовательских ожиданий (подразумевая «тестирования для проверки» - testing for validation), спецификации («тестирование для аттестации» - testing for verification) или, наконец, в контексте предсказанного поведения на основе неявных требований или обоснованных ожиданий.

Общий взгляд на тестирование программного обеспечения в последние годы активно эволюционировал, становясь все более конструктивным, прагматичным и приближенным к реалиям современных проектов разработки программных систем. Тестирование более не рассматривается как деятельность, начинающаяся только после завершения фазы конструирования. Как об этом уже говорилось в различных методологиях разработки, тестирование рассматривается как деятельность, которую необходимо проводить на протяжении всего процесса разработки и сопровождения и является важной частью конструирования программных продуктов. Действительно, планирование тестирования должно начинаться на ранних стадиях работы с требованиями, необходимо систематически и постоянно развивать и уточнять планы тестов и соответствующие процедуры тестирования. Даже сами по себе сценарии тестирования оказываются очень полезными для тех, кто занимается проектированием, позволяя выделять те аспекты требований, которые могут неоднозначно интерпретироваться или даже быть противоречивыми.

Не секрет, что легче предотвратить проблему, чем бороться с ее последствиями. Тестирование, наравне с управлением рисками, является тем инструментом, который позволяет действовать именно в таком ключе. Причем действовать достаточно эффективно. С другой стороны, необходимо осознавать, что даже если приемочные тесты показали положительные результаты, это совсем не означает, что полученный продукт не содержит ошибок. Однако, адекватное внимание вопросам тестирования качественно снижает риск возникновения ошибок на этапе эксплуатации, обеспечивая более высокую удовлетворенность пользователей, что и является, по существу, целью любого проекта.

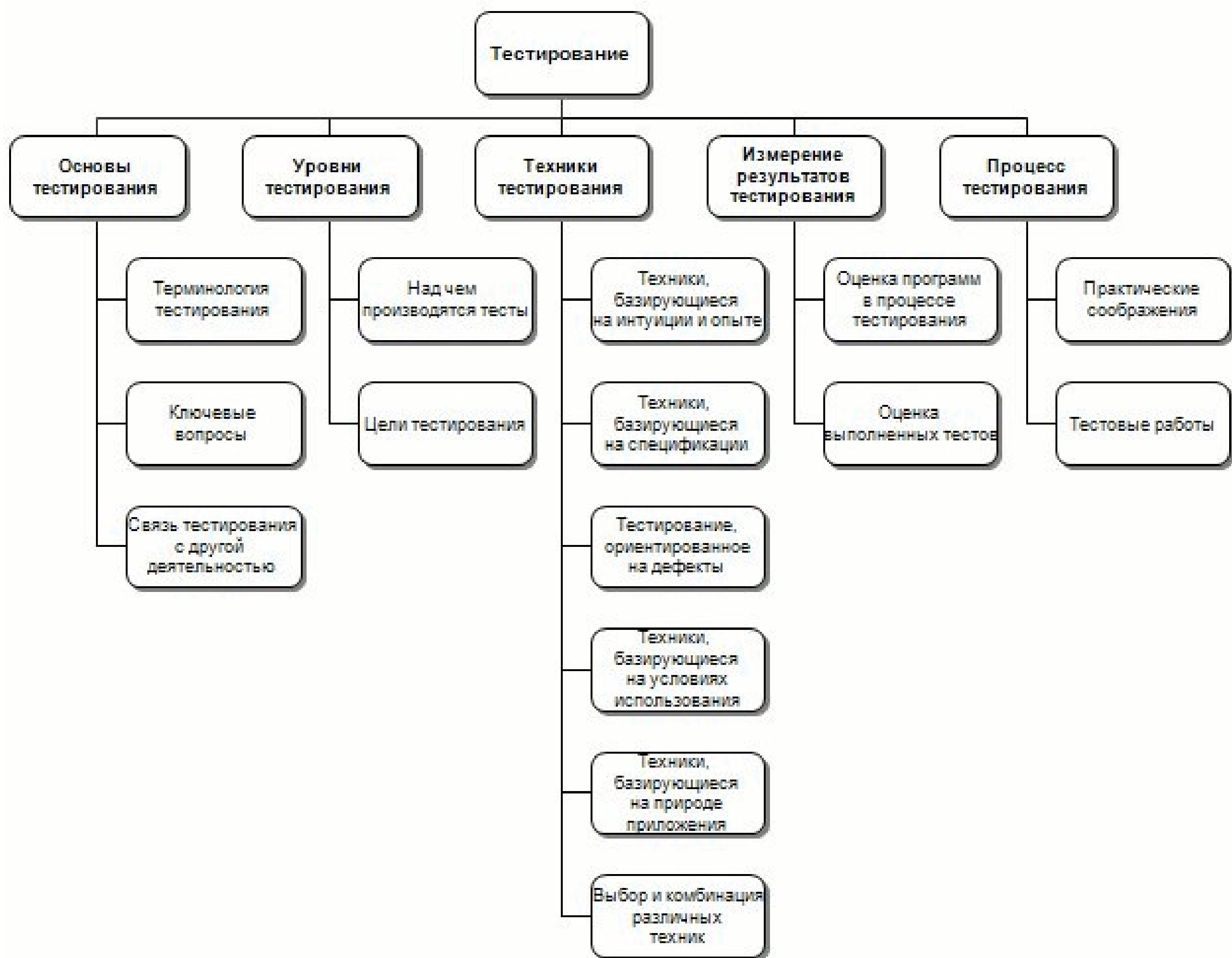

Рисунок 14: Тестирование программного обеспечения [SWEBOK 2004]

С точки зрения управления качеством программного обеспечения, четко разделяются статические (без выполнения кода) и динамические (с выполнением кода) проверки. Обе эти категории важны. Тестирование рассматривает только динамические (с выполнением кода) техники.

Цели тестирования

Тестирование проводится в соответствии с определенными целями (Objectivies of Testing, могут быть заданы явно или неявно) и различным уровнем точности. Определение цели точным образом, выражаемым количественно, позволяет обеспечить контроль результатов тестирования.

Тестовые сценарии могут разрабатываться как для проверки функциональных требований (известны как функциональные тесты), так и для оценки нефункциональных требований. При этом, существуют такие тесты, когда количественные параметры и результаты тестов могут лишь опосредованно говорить об удовлетворении целям тестирования (например, «usability» — легкость, простота использования, в большинстве случаев, не может быть явно описана количественными характеристиками).

Можно выделить следующие, наиболее распространенные и обоснованные цели (а, соответственно, виды) тестирования:

1. Приемочное тестирование (Acceptance/qualification testing) . Проверяет поведение системы на предмет удовлетворения требований заказчика. Это возможно в том случае, если заказчик берет на себя ответственность, связанную с проведением таких работ, как сторона «принимающая» программную систему, или специфицированы типовые задачи, успешная проверка (тестирование) которых позволяет говорить об удовлетворении требований заказчика. Такие тесты могут проводиться как с привлечением разработчиков системы, так и без них.
2. Установочное тестирование (Installation testing) . Данные тесты проводятся с целью проверки процедуры инсталляции системы в целевом окружении.
3. Альфа- и бета-тестирование (Alpha and beta testing) . Перед тем, как выпускается программное обеспечение, как минимум, оно должно проходить стадии альфа (внутреннее пробное использование) и бета (пробное использование с привлечением отобранных внешних пользователей) версий. Отчеты об ошибках, поступающие от пользователей этих версий продукта, обрабатываются в соответствии с определенными процедурами, включающими подтверждающие тесты (любого уровня), проводимые специалистами группы разработки. Данный вид тестирования не может быть заранее спланирован.
4. Функциональные тесты/тесты соответствия (Conformance

testing/Functional testing/Correctness testing) . Эти тесты могут называться по разному, однако, их суть проста — проверка соответствия системы, предъявляемым к ней требованиям, описанным на уровне спецификации поведенческих характеристик.

5. Достижение и оценка надежности (Reliability achievement and evaluation) . Помогая идентифицировать причины сбоев, тестирование подразумевает и повышение надежности программных систем. Случайно генерируемые сценарии тестирования могут применяться для статистической оценки надежности. Обе цели — повышение и оценка надежности — могут достигаться при использовании моделей повышения надежности.
6. Регрессионное тестирование (Regression testing) . Определение успешности регрессионных тестов (IEEE 610-90 "Standard Glossary of Software Engineering Terminology") гласит: «повторное выборочное тестирование системы или компонент для проверки сделанных модификаций не должно приводить к непредусмотренным эффектам». На практике это означает, что если система успешно проходила тесты до внесения модификаций, она должна их проходит и после внесения таковых. Основная проблема регрессионного тестирования заключается в поиске компромисса между имеющимися ресурсами и необходимостью проведения таких тестов по мере внесения каждого изменения. В определенной степени, задача состоит в том, чтобы определить критерии «масштабов» изменений, с достижением которых необходимо проводить регрессионные тесты.
7. Тестирование производительности (Performance testing) . Специализированные тесты проверки удовлетворения специфических требований, предъявляемых к параметрам производительности. Существует особый подвид таких тестов, когда делается попытка достижения количественных пределов, обусловленных характеристиками самой системы и ее операционного окружения.
8. Нагрузочное тестирование (Stress testing) . Необходимо понимать отличия между рассмотренным выше тестированием производительности с целью достижения ее реальных (достижимых) возможностей производительности и выполнением программной системы с повышением нагрузки, вплоть до достижения запланированных характеристик и далее, с отслеживанием поведения на всем протяжении повышения загрузки системы.

9. Сравнительное тестирование (Back-to-back testing) . Единичный набор тестов, позволяющих сравнить две версии системы.
10. Восстановительные тесты (Recovery testing) . Цель — проверка возможностей рестарта системы в случае непредусмотренной катастрофы (disaster), влияющей на функционирование операционной среды, в которой выполняется система.
11. Конфигурационное тестирование (Configuration testing) . В случаях, если программное обеспечение создается для использования различными пользователями (в терминах «ролей»), данный вид тестирования направлен на проверку поведения и работоспособности системы в различных конфигурациях.
12. Юзабилити-тестирование (Usability testing) . Цель — в проверить, насколько легко конечный пользователь системы может ее освоить, включая не только функциональную составляющую, но и ее документацию; насколько эффективно пользователь может выполнять задачи, автоматизация которых осуществляется с использованием данной системы; наконец, насколько хорошо система застрахована (с точки зрения потенциальных сбоев) от ошибок пользователя.

Разработка, управляемая тестированием (Test-driven development). По-сути, это не столько техника тестирования, сколько стиль организации процесса разработки, жизненного цикла, когда тесты являются неотъемлемой частью требований (и соответствующих спецификаций) вместо того, чтобы рассматриваться независимой деятельностью по проверке удовлетворения требований программной системой.

Иногда говорят о таком стиле разработки как о самостоятельной методологии — TDD. Однако, принято говорить о TDD не как о методологии, а скорее как о технике, практике или стиле организации работы.

В меньшей степени это относится к FDD — Feature-Driven Development (разработка на основе функциональных возможностей). TDD может естественно рассматриваться как составная часть XP или, как минимум Agile-методов. В свою очередь, FDD может рассматриваться как один из методов гибкой разработки.

Столь разительное отличие между двумя столь схожими подходами заключается в том, что тесты суть инструмент достижения характеристик системы, удовлетворяющей заданным требованиям, то есть потребностям

пользователей, а «возможности» (features) — практически сами (чаще – функциональные) требования, воплощенные (в идеальном случае) в коде.

Классификация методов тестирования

Говоря о тестировании, важно четко разделять причину нарушения работы прикладных систем, обычно описываемую терминами недостаток (fault) или дефект, и наблюдаемый нежелательный эффект, вызываемый этими причинами – сбой (failure). Термин ошибка, в зависимости от контекста, может описывать как причину сбоя, так и сам сбой. Тестирование позволяет обнаружить дефекты, приводящие к сбоям.

Необходимо понимать, что причина сбоя не всегда может быть однозначно определена. Не существует теоретических критериев, позволяющих гарантированно определить какой именно дефект приводит к наблюдаемому сбою.

Говоря о тестировании обычно выделяют следующие признаки, на основе которых проводят классификацию тестирования.

Классификация по форме тестирования

- Статические тестирование (static code analisys)
- Динамическое тестирование (testing)

Классификация по объекту тестирования

- Функциональное тестирование (functional testing)
- Тестирование производительности (performance testing)
    - Нагрузочное тестирование (load testing)
    - Стресс-тестирование (stress testing)
- Тестирование стабильности (stability / endurance / soak testing)
- Юзабилити-тестирование (usability testing)
- Тестирование интерфейса пользователя (UI testing)
- Тестирование безопасности (security testing)
- Тестирование локализации (localization testing)
- Тестирование совместимости (compatibility testing)

Классификация по знанию системы

- Тестирование черного ящика (black box)

- Тестирование белого ящика (white box)
- Тестирование серого ящика (grey box)

Классификация по степени автоматизации

- Ручное тестирование (manual testing)
- Автоматизированное тестирование (automated testing)
- Полуавтоматизированное тестирование (semiautomated testing)

Классификация по степени изолированности компонентов

- Компонентное (модульное) тестирование (component/unit testing)
- Интеграционное тестирование (integration testing)
- Системное тестирование (system/end-to-end testing)

Классификация по времени проведения тестирования

- Альфа-тестирование (alpha testing)
- Тестирование при приёмке (smoke testing)
    - Тестирование новой функциональности (new feature testing)
    - Регрессионное тестирование (regression testing)
    - Тестирование при сдаче (acceptance testing)
- Бета-тестирование (beta testing)

Классификация по признаку позитивности сценариев

- Позитивное тестирование (positive testing)
- Негативное тестирование (negative testing)

Классификация по степени подготовленности к тестированию

- Тестирование по документации (formal testing)
- Тестирование ad hoc или интуитивное тестирование (ad hoc testing)

Уровни тестирования (юнит-, интеграционное, системное тестирование)

Тестирование обычно производится на протяжении всей разработки и сопровождения на разных уровнях. Уровень тестирования определяет «над чем» производятся тесты: над отдельным модулем, группой модулей или системой, в целом. При этом ни один из уровней тестирования не может считаться приоритетным. Важны все уровни тестирования, вне зависимости от используемых моделей и методологий.

Модульное тестирование (Unit testing)

Этот уровень тестирования позволяет проверить функционирование отдельно взятого элемента системы. Что считать элементом-модулем системы определяется контекстом. Наиболее полно данный вид тестов описан в стандарте IEEE 1008-87 «Standard for Software Unit Testing», задающем интегрированную концепцию систематического и документированного подхода к модульному тестированию.

Цель модульного тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

Этот тип тестирования обычно выполняется программистами.

Данный вид тестирования позволяет осуществлять широкое поле действий в ходе дальнейших шагов жизненном цикле программного обеспечения:

- Поощрение изменений . Модульное тестирование позже позволяет программистам проводить рефакторинг, будучи уверенными, что модуль по-прежнему работает корректно (регрессионное тестирование). Это поощряет программистов к изменениям кода, поскольку достаточно легко проверить, что код работает и после изменений.
- Упрощение интеграции . Модульное тестирование помогает устранить сомнения по поводу отдельных модулей и может быть использовано для подхода к тестированию «снизу вверх»: сначала тестируются отдельные части программы, затем программа в целом.
- Документирование кода . Модульные тесты можно рассматривать как «живой документ» для тестируемого класса. Клиенты, которые не знают, как использовать данный класс, могут использовать юнит-тест в качестве примера.

- Отделение интерфейса от реализации . Поскольку некоторые классы могут использовать другие классы, тестирование отдельного класса часто распространяется на связанные с ним. Например, класс пользуется базой данных; в ходе написания теста программист обнаруживает, что тесту приходится взаимодействовать с базой. Это ошибка, поскольку тест не должен выходить за границу класса. В результате разработчик абстрагируется от соединения с базой данных и реализует этот интерфейс, используя свой собственный т.н. mock-объект. Это приводит к менее связанному коду, минимизируя зависимости в системе.

Однако, как любая технология тестирования, модульное тестирование не позволяет отловить все ошибки программы. В самом деле, это следует из практической невозможности трассировки всех возможных путей выполнения программы, за исключением простейших случаев. Кроме того, происходит тестирование каждого из модулей по отдельности. Это означает, что ошибки интеграции, системного уровня, функций, исполняемых в нескольких модулях, не будут определены. Кроме того, данная технология бесполезна для проведения тестов на производительность. Таким образом, модульное тестирование более эффективно при использовании в сочетании с другими методиками тестирования.

Тестирование программного обеспечения — комбинаторная задача. Например, каждое возможное значение булевской переменной потребует двух тестов: один на вариант TRUE, другой — на вариант FALSE. В результате на каждую строку исходного кода потребуется 3-5 строк тестового кода.

Для получения выгоды от модульного тестирования требуется строго следовать технологии тестирования на всем протяжении процесса разработки программного обеспечения. Нужно хранить не только записи обо всех проведенных тестах, но и обо всех изменениях исходного кода во всех модулях. С этой целью следует использовать систему контроля версий ПО. Таким образом, если более поздняя версия ПО не проходит тест, который был успешно пройден ранее, будет несложным сверить варианты исходного кода и устранить ошибку. Также необходимо убедиться в неизменном отслеживании и анализе неудачных тестов. Игнорирование этого требования приведет к лавинообразному увеличению неудачных тестовых результатов.

Интеграционное тестирование (Integration testing)

Данный уровень тестирования является процессом проверки взаимодействия между программными компонентами/модулями.

Классические стратегии интеграционного тестирования – «сверху-вниз» и «снизу-вверх» – используются для традиционных, иерархически структурированных систем и их сложно применять, например, к тестированию слабосвязанных систем, построенных в сервисно-ориентированных архитектурах (SOA).

Современные стратегии в большей степени зависят от архитектуры тестируемой системы и строятся на основе идентификации функциональных «потоков» (например, потоков операций и данных).

С технологической точки зрения интеграционное тестирование является количественным развитием модульного, поскольку так же, как и модульное тестирование, оперирует интерфейсами модулей и подсистем и требует создания тестового окружения, включая заглушки на месте отсутствующих модулей. Основная разница между модульным и интеграционным тестированием состоит в целях, то есть в типах обнаруживаемых дефектов, которые, в свою очередь, определяют стратегию выбора входных данных и методов анализа. В частности, на уровне интеграционного тестирования часто применяются методы, связанные с покрытием интерфейсов, например, вызовов функций или методов, или анализ использования интерфейсных объектов, таких как глобальные ресурсы, средства коммуникаций, предоставляемых операционной системой.

Интеграционное тестирование – постоянно проводимая деятельность, предполагающая работу на достаточно высоком уровне абстракции. Наиболее успешная практика интеграционного тестирования базируется на инкрементальном подходе, позволяющем избежать проблем проведения разовых тестов, связанных с тестированием результатов очередного длительного этапа работ, когда количество выявленных дефектов приводит к серьезной переработке кода (традиционно, негативный опыт выпуска и тестирования только крупных релизов называют «big bang»).

Интеграционное тестирование применяется на этапе сборки модульно-оттестированных модулей в единый комплекс. Известны два метода сборки модулей:

- Монолитный, характеризующийся одновременным объединением всех

модулей в тестируемый комплекс

- Инкрементальный, характеризующийся пошаговым (помодульным) наращиванием комплекса программ с пошаговым тестированием собираемого комплекса. В инкрементальном методе выделяют две стратегии добавления модулей:
  - «Сверху вниз» и соответствующее ему восходящее тестирование.
  - «Снизу вверх» и соответственно нисходящее тестирование.

Особенности монолитного тестирования заключаются в следующем: для замены неразработанных к моменту тестирования модулей, кроме самого верхнего, необходимо дополнительно разрабатывать драйверы (test driver) и/или заглушки (stub), замещающие отсутствующие на момент сеанса тестирования модули нижних уровней.

Сравнение монолитного и интегрального подхода дает следующее:

- Монолитное тестирование требует больших трудозатрат, связанных с дополнительной разработкой драйверов и заглушек и со сложностью идентификации ошибок, проявляющихся в пространстве собранного кода.
- Пошаговое тестирование связано с меньшей трудоемкостью идентификации ошибок за счет постепенного наращивания объема тестируемого кода и соответственно локализации добавленной области тестируемого кода.
- Монолитное тестирование предоставляет большие возможности распараллеливания работ особенно на начальной фазе тестирования.

Особенности нисходящего тестирования заключаются в следующем: организация среды для исполняемой очередности вызовов оттестированными модулями тестируемых модулей, постоянная разработка и использование заглушек, организация приоритетного тестирования модулей, содержащих операции обмена с окружением, или модулей, критичных для тестируемого алгоритма.

Особенности восходящего тестирования заключаются в организации порядка сборки и перехода к тестированию модулей, соответствующему порядку их реализации.

Недостатки восходящего тестирования:

- Запаздывание проверки концептуальных особенностей тестируемого

комплекса .

- Необходимость в разработке и использовании драйверов.

Системное тестирование (System testing)

Системное тестирование охватывает целиком всю систему. Большинство функциональных сбоев должно быть идентифицировано еще на уровне модульных и интеграционных тестов. В свою очередь, системное тестирование, обычно фокусируется на нефункциональных требованиях — безопасности, производительности, точности, надежности т.п.

На этом уровне также тестируются интерфейсы к внешним приложениям, аппаратному обеспечению, операционной среде и т.д.

Системное тестирование относится к методам тестирования черного ящика, и, тем самым, не требует знаний о внутреннем устройстве системы.

Основной задачей системного тестирования является проверка как функциональных, так и не функциональных требований к системе в целом. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д. Для минимизации рисков, связанных с особенностями поведения системы в той или иной среде, во время тестирования рекомендуется использовать окружение максимально приближенное к тому, на которое будет установлен продукт после выдачи.

Можно выделить два подхода к системному тестированию:

- на базе требований (requirements based) . Для каждого требования пишутся тестовые случаи (test cases), проверяющие выполнение данного требования.
- на базе случаев использования (use case based) .

Альфа-тестирование и бета-тестирование являются подкатегориями системного тестирования.

Альфа- и бета-тестирование

Альфа-тестирование — имитация реальной работы с системой штатными разработчиками, либо реальная работа с системой потенциальными пользователями/заказчиком. Чаще всего альфа-тестирование проводится на ранней стадии разработки продукта, но в некоторых случаях может применяться для законченного продукта в качестве внутреннего приемочного тестирования. Иногда альфа-тестирование выполняется под отладчиком или с

использованием окружения, которое помогает быстро выявлять найденные ошибки. Обнаруженные ошибки могут быть переданы тестировщикам для дополнительного исследования в окружении, подобном тому, в котором будет использоваться программным обеспечением.

Бета-тестирование — это интенсивное использование почти готовой версии продукта (как правило, программного или аппаратного обеспечения) с целью выявления максимального числа ошибок в его работе для их последующего устранения перед окончательным выходом (релизом) продукта на рынок, к массовому потребителю.

В отличие от альфа-тестирования, проводимого силами штатных разработчиков или инженеров по тестированию, бета-тестирование предполагает привлечение добровольцев из числа обычных будущих пользователей продукта, которым доступна упомянутая предварительная версия продукта (так называемая бета-версия).

Такими добровольцами (их называют бета-тестерами) часто движет любопытство к новому продукту — любопытство, ради удовлетворения которого они вполне согласны мириться с возможностью испытать последствия еще не найденных (а потому и не исправленных) ошибок. Кроме любопытства, мотивация может быть обусловлена желанием повлиять на процесс разработки и в итоге получать более удовлетворяющий их нужды продукт и многим другим.

Кроме того, открытие бета-тестирования может использоваться как часть стратегии продвижения продукта на рынок (например, бесплатная раздача бета-версий позволяет привлечь широкое внимание потребителей к окончательной дорогостоящей версии продукта), а также для получения предварительных отзывов о нем от широкого круга будущих пользователей.

Бета-версия не является финальной версией продукта, поэтому разработчик не гарантирует полного отсутствия ошибок, которые могут нарушить работу компьютера и/или привести к потере данных. Хотя и в финальных версиях таких гарантий разработчики, как правило, не дают.

Другие виды тестирования

Статическое тестирование

Статическое тестирование в противовес динамическому — процесс, направленный на выявление ошибок без реального запуска приложения. Статический анализ кода — анализ программного обеспечения, производимый без реального выполнения исследуемых программ (анализ, производимый с выполнением программ, называется динамический анализ кода). В большинстве случаев анализ производится над какой-либо версией исходного кода, хотя иногда анализу подвергается какой-нибудь вид объектного кода, например P-код или код на MSIL. Термин обычно применяют к анализу, производимому специальным программным обеспечением, тогда как ручной анализ называют пониманием или постижением программы.

В зависимости от используемого инструмента глубина анализа может варьироваться от определения поведения отдельных операторов до анализа, включающего весь имеющийся исходный код. Способы использования полученной в ходе анализа информации также различны — от выявления мест, возможно содержащих ошибки, до формальных методов, позволяющих математически доказать какие-либо свойства программы (например, соответствие поведения спецификации). Некоторые люди считают программные метрики и обратное проектирование формами статического анализа.

Обычно с помощью статического анализа выявляют следующие виды ошибок:

- Неопределенное поведение — неинициализированные переменные, обращение к NULL-указателям. Часто о простейших случаях сигнализируют и компиляторы.
- Нарушение блок-схемы пользования библиотекой. Например, для каждого fopen нужен fclose. И если файловая переменная теряется раньше, чем файл закрывается, анализатор может сообщить об ошибке.
- Типичные сценарии, приводящие к недокументированному поведению. Стандартная библиотека языка Си известна большим количеством неудачных технических решений. Некоторые функции, например, gets, в принципе небезопасны. sprintf и strcpy безопасны лишь при определенных условиях.

- Переполнение буфера — когда компьютерная программа записывает данные за пределами выделенного в памяти буфера.
- Типичные сценарии, мешающие аппаратной кроссплатформенности.
- Ошибки в повторяющемся коде. Многие программы исполняют несколько раз одно и то же с разными аргументами. Обычно повторяющиеся фрагменты не пишут с нуля, а размножают и исправляют.
- Неизменный параметр, передаваемый в функцию — признак изменившихся требований к программе. Когда-то параметр был задействован, но сейчас он уже не нужен. В таком случае программист может вообще избавиться от этого параметра — и от связанной с ним логики.
- Прочие ошибки — многие функции из стандартных библиотек не имеют побочного эффекта, и вызов их как процедур не имеет смысла.

Программное обеспечение для статического анализа

- BLAST (Berkeley Lazy Abstraction Software Verification Tool) — статический анализатор кода для языка программирования C.
- T-SQL Analyzer — инструмент, который может просматривать программные модули в базах данных под управлением Microsoft SQL Server.
- ReSharper — плагин для Microsoft Visual Studio, обладающий огромным количеством вспомогательных функций.
- SourceAnalyzer — статический анализатор кода, предназначенный для поиска и анализа зависимости функций препроцессированного кода.
- FindBugs — статический анализатор кода для Java.

Юзабилити-тестирование

Удобство использования пользовательского интерфейса (usability) — показатель его качества, который определяет количество усилий, необходимых для изучения принципов работы с программной системой при помощи данного интерфейса, ее использования, подготовки входных данных и интерпретации выходных. Иначе говоря, удобство использования определяет степень простоты доступа пользователя к функциям системы, предоставляемым через человеко-машинный (пользовательский) интерфейс.

Тестирование удобства использования пользовательского интерфейса, вообще говоря, не относится к классическим методам тестирования программных систем. Специалист по тестированию пользовательского интерфейса должен сочетать в себе знания как в области программной инженерии, так и в физиологии, психологии и эргономике.

На удобство использования пользовательского интерфейса влияют следующие факторы:

- легкость обучения — быстро ли человек учится использовать систему;
- эффективность обучения — быстро ли человек работает после обучения;
- запоминаемость обучения — легко ли запоминается все, чему человек научился;
- ошибки — часто ли человек допускает ошибки в работе;
- общая удовлетворенность — является ли общее впечатление от работы с системой положительным.

Все эти факторы, несмотря на свою неформальность, могут быть измерены. Для таких измерений выбирается группа типичных пользователей системы, и в процессе их работы измеряются показатели их работы с системой (например, количество допущенных ошибок), а также им предлагается высказать собственные впечатления от системы при помощи заполнения опросных листов.

Выделяют следующие этапы тестирования удобства использования пользовательского интерфейса.

- Исследовательское.

  Проводится после формулирования требований к системе и разработки прототипа интерфейса. Основная цель на этом этапе - провести

высокоуровневое обследование интерфейса и выяснить, позволяет ли он с достаточной степенью эффективности решать задачи пользователя.

- Оценочное.

  Проводится после разработки низкоуровневых требований и детализированного прототипа пользовательского интерфейса. Оценочное тестирование углубляет исследовательское и имеет ту же цель. На данном этапе уже проводятся количественные измерения характеристик пользовательского интерфейса: измеряются количество обращений к системе помощи по отношению к количеству совершенных операций, количество ошибочных операций, время устранения последствий ошибочных операций и т.п.

- Валидационное.

  Проводится ближе к этапу завершения разработки. На этом этапе проводится анализ соответствия интерфейса программной системы стандартам, регламентирующим вопросы удобства интерфейса (например ISO 13407, ISO 9126), проводится общее тестирование всех компонент пользовательского интерфейса с точки зрения конечного пользователя. Под компонентами интерфейса здесь понимается как его программная реализация, так и система помощи и руководство пользователя. Также на данном этапе проверяется отсутствие дефектов удобства использования интерфейса, выявленных на предыдущих этапах.

- Сравнительное.

  Данный вид тестирования может проводиться на любом этапе разработки интерфейса. В ходе сравнительного тестирования сравниваются два или более вариантов реализации пользовательского интерфейса.

Как правило, при тестировании удобства использования пользовательского интерфейса используются некоторые эвристические критерии и характеристики, которые заменяют точные оценки в классическом тестировании программных систем.

Например, Якоб Нильсен выделил 10 эвристических характеристик удобного пользовательского интерфейса, которые с его точки зрения должны проверяться при тестировании удобства использования интерфейса.

1. Наблюдаемость состояния системы.

   Система всегда должна оповещать пользователя о том, что она в данный

момент делает, причем через разумные промежутки времени.

2. Соотнесение с реальным миром.

   Терминология, использованная в интерфейсе системы должна соотноситься с пользовательским миром, т.е. это должна быть терминология проблемной области пользователя, а не техническая терминология.

3. Пользовательское управление и свобода действий.

   Пользователи часто выбирают отдельные интерфейсные элементы и используют функции системы по ошибке. В этом случае необходимо предоставлять четко определенный "аварийный выход", при помощи которого можно вернуться к предыдущему нормальному состоянию. К таким "аварийным выходам" относятся, например, функции отката и обратного отката.

4. Целостность и стандарты.

   Для обозначения одних и тех же объектов, ситуаций и действий должны использоваться одинаковые слова во всех частях интерфейса. Более того, терминология сообщений в пользовательском интерфейсе должна учитывать соглашения конкретной платформы.

5. Помощь пользователям в распознавании, диагностике и устранении ошибок.

   Сообщения об ошибках должны быть написаны на естественном языке, а не заменяться кодами ошибок. Сообщения об ошибках должны четко определять суть возникшей проблемы и предлагать ее конструктивное решение.

6. Предотвращение ошибок.

   Продуманный дизайн пользовательского интерфейса, предотвращающий появление ошибок пользователя, всегда лучше хорошо продуманных сообщений об ошибках. При проектировании интерфейса необходимо либо полностью устранить элементы, в которых могут возникать ошибки пользователя, либо проверять ввод пользователя в этих элементах и сообщать ему о потенциально возможном возникновении проблемы.

7. Распознавание, а не вспоминание.

   При создании интерфейса необходимо минимизировать нагрузку на память пользователя, делая объекты, действия и опции ясными,

доступными и явно видимыми. Пользователь не должен запоминать информацию при переходе от одного диалогового окна к другому. Во всех необходимых местах должны быть доступны контекстные инструкции по использованию интерфейса.

8. Гибкость и эффективность использования.

   В интерфейсе должны быть предусмотрены горячие клавиши (не обязательные к использованию начинающим пользователем) - они часто значительно ускоряют работу опытного пользователя. Иными словами, система должна предоставлять два способа работы - для новичков и для опытных пользователей. Желательно при этом давать возможность пользователю автоматизировать часто повторяющиеся действия.

9. Эстетичный и минимально необходимый дизайн.

   Окна не должны содержать не относящуюся к делу или редко используемую информацию. Каждый интерфейсный элемент, содержащий бесполезную информацию, играет роль информационного шума и отвлекает пользователя от действительно полезных интерфейсных элементов.

10. Помощь и документация.

    Несмотря на то, что в идеальном случае лучше, когда системой можно пользоваться без документации, таковая все равно необходима - как в виде системы помощи, так и, возможно, в виде печатного руководства. Информация в документации должна быть структурирована таким образом, чтобы пользователь мог легко найти нужный раздел, посвященный решаемой им задаче. Каждый такой ориентированный на конкретную задачу раздел должен помимо общей информации содержать пошаговые руководства по выполнению задачи и не должен быть слишком длинным.

Все эти эвристики могут использоваться при тестировании удобства использования пользовательского интерфейса. Достаточно очевидно, что при тестировании удобства слабо применимы способы автоматизации тестирования при помощи сценариев и подобные методы. Один из наиболее эффективных методов проверки интерфейса на удобство — использование формальной инспекции. Вопросы в бланке инспекции могут быть как общего характера (так, например, можно использовать в качестве вопросов перечисленные выше 10 эвристик), так и вполне конкретными.

Автоматизированное тестирование

Существует два основных подхода к автоматизации тестирования: тестирование на уровне кода и тестирование пользовательского интерфейса (в частности, GUI-тестирование). К первому типу относится, в частности, модульное тестирование. Ко второму — имитация действий пользователя с помощью специальных тестовых систем.

Наиболее распространенной формой автоматизации является тестирование приложений через графический пользовательский интерфейс. Популярность такого вида тестирования объясняется двумя факторами: во-первых, приложение тестируется тем же способом, которым его будет использовать человек, во-вторых, можно тестировать приложение, не имея при этом доступа к исходному коду.

GUI-автоматизация развивалась в течение 4 поколений инструментов и техник:

- Утилиты записи и воспроизведения (capture/playback tools) записывают действия тестировщика во время ручного тестирования. Они позволяют выполнять тесты без прямого участия человека в течение продолжительного времени, значительно увеличивая продуктивность и устраняя «тупое» повторение однообразных действий во время ручного тестирования. В то же время, любое малое изменение тестируемого программного обеспечения требует перезаписи ручных тестов. Поэтому это первое поколение инструментов не эффективно и не масштабируемо.
- Сценарии (Scripting) — форма программирования на языках, специально разработанных для автоматизации тестирования программного обеспечения — смягчает многие проблемы capture/playback tools. Но разработкой занимаются программисты высокого уровня, которые работают отдельно от инженеров по тестированию, непосредственно запускающих тесты. К тому же скрипты более всего подходят для тестирования GUI и не могут быть внедренными, пакетными или вообще каким-либо образом объединены в систему. Наконец, изменения в тестируемом программном обеспечении требуют сложных изменений в соответствующих скриптах, и поддержка все возрастающей библиотеки тестирующих скриптов становится в конце концов непреодолимой задачей.
- Data-driven testing — методология, которая используется в автоматизации

тестирования. Особенностью является то, что тестовые скрипты выполняются и верифицируются на основе данных, которые хранятся в центральном хранилище данных или базе данных. Роль БД могут выполнять ODBC-ресурсы, csv или xls файлы и т. д. Data-driven testing — это объединение нескольких взаимодействующих тестовых скриптов и их источников данных в фреймворк, используемый в методологии. В этом фреймворке переменные используются как для входных значений, так и для выходных проверочных значений: в тестовом скрипте обычно закодированы навигация по приложению, чтение источников данных, ведение логов тестирования. Таким образом, логика, которая будет выполнена в скрипте, также зависит от данных.

- Keyword-based автоматизация подразумевает разделение процесса создания кейсов на 2 этапа: этап планирования и этап реализации.

Одной из главных проблем автоматизированного тестирования является его трудоемкость: несмотря на то, что оно позволяет устранить часть рутинных операций и ускорить выполнение тестов, большие ресурсы могут тратиться на обновление самих тестов. Это относится к обоим видам автоматизации. При рефакторинге часто бывает необходимо обновить и модульные тесты, и изменение кода тестов может занять столько же времени, сколько и изменение основного кода. С другой стороны, при изменении интерфейса приложения необходимо заново переписать все тесты, которые связаны с обновленными окнами, что при большом количестве тестов может отнять значительные ресурсы.

Чаще всего автоматизированное тестирование, не связанное с тестированием интерфейса используется при регрессионном тестировании.

Результат тестирования — баг

Целью процесса тестирование является нахождение ошибок в испытуемой системе. Результаты труда инженера по тестированию должны быть зафиксированы и переданы программисту. Для регистрации и ведения найденных ошибок системы используют системы отслеживания и регистрации ошибок.

Система отслеживания ошибок (bug tracking system) — прикладная программа, разработанная с целью помочь разработчикам программного обеспечения (программистам, инженерам по тестированию и др.) учитывать и контролировать ошибки (баги), найденные в программах, пожелания пользователей, а также следить за процессом устранения этих ошибок и выполнения или невыполнения пожеланий.

Главный компонент такой системы — база данных, содержащая сведения об обнаруженных дефектах. Эти сведения чаще всего включают в себя следующую информацию :

- номер (идентификатор) дефекта;
- кто сообщил о дефекте;
- дата и время, когда был обнаружен дефект;
- версия продукта, в которой обнаружен дефект;
- серьёзность (критичность) дефекта и приоритет решения;
- описание шагов для выявления дефекта (воспроизведения неправильного поведения программы);
- кто ответственен за устранение дефекта;
- обсуждение возможных решений и их последствий;
- текущее состояние (статус) дефекта;
- версия продукта, в которой дефект исправлен.

Кроме того, развитые системы предоставляют возможность прикреплять файлы, помогающие описать проблему (например, дамп памяти или скриншот).

Как правило, система отслеживания ошибок использует тот или иной вариант «жизненного цикла» ошибки, стадия которого определяется текущим состоянием, или статусом, в котором находится ошибка.

Типичный жизненный цикл дефекта:

- Новый — дефект зарегистрирован инженером по тестированию
- Назначен — назначен ответственный за исправление дефекта
- Разрешен — дефект переходит обратно в сферу ответственности инженера по тестириванию. Как правило, сопровождается резолюцией, например:
    - Исправлено (исправления включены в версию такую-то)
    - Дубль (повторяет дефект, уже находящийся в работе)
    - Не исправлено (работает в соответствии со спецификацией, имеет слишком низкий приоритет, исправление отложено до следующей версии и т.п.)
    - «У меня все работает» (запрос дополнительной информации об условиях, в которых дефект проявляется)
- Далее инженер по тестированию проводит проверку исправления, в зависимости от чего дефект либо снова переходит в статус Назначен (если он описан как исправленный, но не исправлен), либо в статус Закрыт.
- Открыт повторно — дефект вновь найден в другой версии.

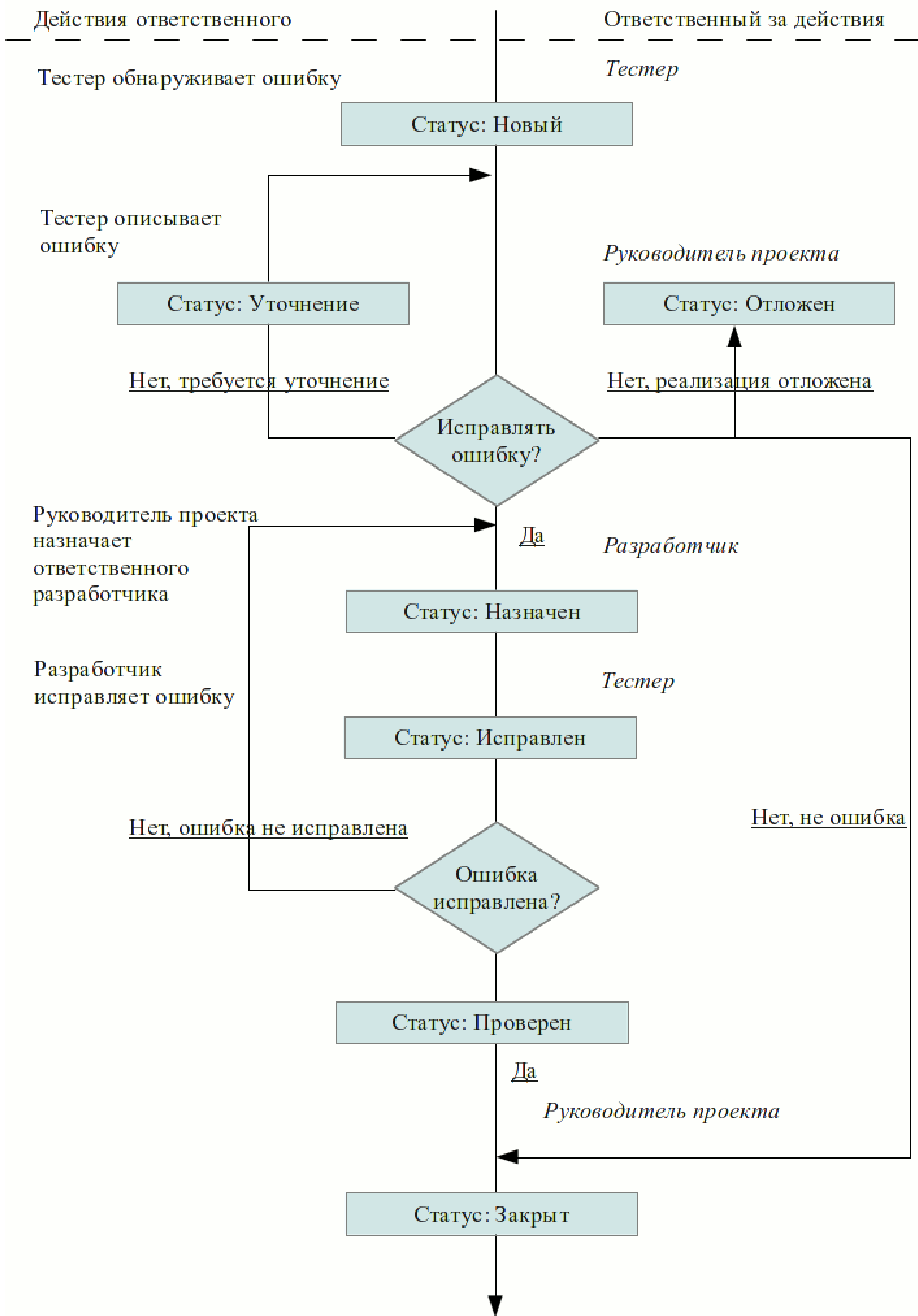

Рисунок 15: Жизненный цикл ошибки в системе

Система может предоставлять администратору возможность настроить,

какие пользователи могут просматривать и редактировать ошибки в зависимости от их состояния, переводить их в другое состояние или удалять.

В корпоративной среде, система отслеживания ошибок может использоваться для получения отчетов, показывающих продуктивность программистов при исправлении ошибок. Однако, часто такой подход не дает достаточно точных результатов, из-за того что разные ошибки имеют различную степень серьезности и сложности. При этом серьезность проблемы не имеет прямого отношения к сложности устранения ошибки.

Программные продукты для Unit-тестирования

Для большинства популярных языков программирования высокого уровня существуют инструменты и библиотеки модульного тестирования. Некоторые из них:

- Для Java
  - JUnit (http://JUnit.org)
  - TestNG (http://testNG.org)
  - JavaTESK (http://UniTESK.ru)
- Для C
  - Cunit (http://cunit.sourceforge.net)
  - CTESK (http://www.unitesk.ru)
  - cfix (http://sourceforge.net/projects/cfix)
  - API Sanity Autotest — для динамических C/C++ библиотек в Unix-системах (http://ispras.linux-foundation.org/index.php/API_Sanity_Autotest)
  - Unity (http://sourceforge.net/projects/unity) — для встраиваемых приложений
- Для Objective-C
  - OCUnit (http://www.sente.ch/software/ocunit)
- Для C++
  - CxxTest (http://cxxtest.com)
  - CPPUnit (http://apps.sourceforge.net/mediawiki/cppunit)
  - Boost Test (http://www.boost.org/doc/libs/1_38_0/libs/test/doc/html)
  - Google C++ Testing Framework (http://code.google.com/p/googletest)
  - Symbian (http://www.symbianosunit.co.uk) — фреймворк для Symbian OS всех версий.
  - API Sanity Autotest (http://ispras.linux-foundation.org/index.php/API_Sanity_Autotest)— для динамических C/C++ библиотек в Unix-подобных операционных системах.

- Для .NET
  - Nunit
  - XUnit
  - MbUnit
- DUnit — для Delphi
- EUnit — Erlang
- Для Perl
  - Test
  - Test::Simple
  - Test::More
  - Test::Unit
  - Test::Unit::Lite
- Для PHP
  - SimpleTest
  - PHPUnit
- Для Python
  - PyUnit
  - PyTest
  - Nose
- vbUnit — для языка программирования Visual Basic
- utPLSQL — PL/SQL
- Для T-SQL
  - TSQLUnit
  - SPUnit
- Для ActionScript 2.0 — язык сценариев, используемый виртуальной машиной Adobe Flash Player версии 7 и 8
  - AsUnit
  - AS2Unit

- Для ActionScript 3.0 — скриптовый язык, используемый виртуальной машиной Adobe Flash Player версии 9 и выше
  - FlexUnit
  - AsUnit
- Test::Unit — для Ruby
- Для JavaScript
  - Jasmine
  - D.O.H

Системы регистрирования ошибок

Часто системы регистрирования ошибок совмещены с системами управления проектами:

- Свободно распространяемые
  - Redmine
  - BUGS - the Bug Genie
  - Bugzilla
  - eTraxis
  - GNATS
  - Mantis bug tracking system
  - Trac
  - EmForge
  - Picket
  - Flyspray
  - DEVPROM
  - YouTrack
- Проприетарные
  - Atlassian JIRA
  - Bontq

- PVCS Tracker
- Project Kaiser
- TrackStudio Enterprise

Разновидности автоматических тестов

Регрессионное тестирование

Регрессионное тестирование (regression testing) — собирательное название для всех видов тестирования программного обеспечения, направленных на обнаружение ошибок в уже протестированных участках исходного кода. Такие ошибки — когда после внесения изменений в программу перестает работать то, что должно было продолжать работать, — называют регрессионными ошибками (regression bugs).

Регрессионное тестирование (по некоторым источникам) включает:

- new bug-fix - проверка исправления вновь найденного дефекта,
- old bug-fix - проверка, что исправленный ранее и верифицированный дефект не воспроизводится в системе снова
- side-effect - проверка того, что не нарушилась работоспособность работающей ранее функциональности, если ее код мог быть затронут при исправлении некоторых дефектов в другой функциональности.

Обычно используемые методы регрессионного тестирования включают повторные прогоны предыдущих тестов, а также проверки, не попали ли регрессионные ошибки в очередную версию в результате слияния кода.

Чаще всего выделяют следующие виды регрессии в порядке их важности (обычно в таком порядке их и выполняют):

1. Тесты верификации версии (Build Verification Test). Нужны эти тесты для проверки основной функциональности каждой версии программы. Только будучи уверенным в том, что основная функциональность программы не нарушена, можно продолжать разработку. При нахождении ошибки с помощью таких тестов необходимо пересмотреть соответствующую часть кода на предмет ошибок.
2. Тесты верификации багов (Bug Verification Test). Если некоторый тест выявил баг, необходимо после исправления провести этот тест еще раз. Хотя проведение этих тестов и является логичным, многие программисты пренебрегают такого вида тестированием.
3. Тесты регрессии (Regression Test Pass). К этим тестам относятся те, которые уже проводились с предыдущими версиями программного обеспечения и не выявляли ошибок. Иногда при отсутствии времени

некоторые из тестов можно пропустить (желательно только тогда, когда не были внесены изменения в соответствующие участки кода). Если ранее такие тесты уже проводились более 3 раз, процесс неплохо было бы автоматизировать.

4. Тесты регрессии на исправленных багах (тесты на закрытых багах). Баг может снова проявиться по ряду причин (особенно при модификации кода). Поэтому время от времени нужно возвращаться к этому месту программы.

Из опыта разработки программного обеспечения известно, что повторное появление одних и тех же ошибок — случай достаточно частый. Иногда это происходит из-за слабой техники управления версиями или по причине человеческой ошибки при работе с системой управления версиями. Но настолько же часто решение проблемы бывает «недолго живущим»: после следующего изменения в программе решение перестает работать. И наконец, при переписывании какой-либо части кода часто всплывают те же ошибки, что были в предыдущей реализации.

Поэтому считается хорошей практикой при исправлении ошибки создать тест на нее и регулярно прогонять его при последующих изменениях программы. Хотя регрессионное тестирование может быть выполнено и вручную, но чаще всего это делается с помощью специализированных программ, позволяющих выполнять все регрессионные тесты автоматически. В некоторых проектах даже используются инструменты для автоматического прогона регрессионных тестов через заданный интервал времени. Обычно это выполняется после каждой удачной компиляции (в небольших проектах) либо каждую ночь или каждую неделю.

Регрессионное тестирование является неотъемлемой частью экстремального программирования. В этой методологии проектная документация заменяется на расширяемое, повторяемое и автоматизированное тестирование всего программного пакета на каждой стадии процесса разработки программного обеспечения.

Регрессионное тестирование может быть использовано не только для проверки корректности программы, часто оно также используется для оценки качества полученного результата. Так, при разработке компилятора, при прогоне регрессионных тестов рассматривается размер получаемого кода, скорость его выполнения и время компиляции каждого из тестовых примеров.

Тестирование безопасности

Тестирование безопасности (Security and Access Control Testing) — это стратегия тестирования, используемая для проверки безопасности системы, а также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения, атак хакеров, вирусов, несанкционированного доступа к конфиденциальным данным.

Общая стратегия безопасности основывается на трех основных принципах:

- Конфиденциальность.

  Это сокрытие определенных ресурсов или информации. Под конфиденциальностью можно понимать ограничение доступа к ресурсу некоторой категории пользователей, или другими словами, при каких условиях пользователь авторизован получить доступ к данному ресурсу.

- Целостность .

  Существует два основных критерия при определении понятия целостности:

  - Доверие. Ожидается, что ресурс будет изменен только соответствующим способом определенной группой пользователей.
  - Повреждение и восстановление. В случае когда данные повреждаются или неправильно меняются авторизованным или не авторизованным пользователем, вы должны определить на сколько важной является процедура восстановления данных.

- Доступность .

  Доступность представляет собой требования о том, что ресурсы должны быть доступны авторизованному пользователю, внутреннему объекту или устройству. Как правило, чем более критичен ресурс тем выше уровень доступности должен быть.

В настоящее время наиболее распространенными видами уязвимости в безопасности программного обеспечения являются:

- XSS (Cross-Site Scripting)

  Это вид уязвимости программного обеспечения (Web приложений), при которой, на генерированной сервером странице, выполняются вредоносные скрипты, с целью атаки клиента.

Сами по себе XSS атаки могут быть очень разнообразными. Злоумышленники могут попытаться украсть куки, перенаправить пользователя на сайт, где произойдет более серьезная атака, загрузить в память какой-либо вредоносный объект и т.д., всего навсего разместив вредоносный скрипт, к примеру, на сайте.

- XSRF / CSRF (Request Forgery)

  Это вид уязвимости, позволяющий использовать недостатки HTTP протокола, при этом злоумышленники работают по следующей схеме: ссылка на вредоносный сайт установливается на странице, пользующейся доверием у пользователя, при переходе по вредоносной ссылке выполняется скрипт, сохраняющий личные данные пользователя (пароли, платежные данные и т.д.), либо отправляющий СПАМ сообщения от лица пользователя, либо изменяет доступ к учетной записи пользователя, для получения полного контроля над ней.

  Наиболее частыми CSRF атаками являются атаки использующие HTML <IMG> тэг или Javascript объект image. Чаще всего атакующий добавляет необходимый код в электронное письмо или выкладывает на веб-сайт, таким образом, что при загрузке страницы осуществляется запрос, выполняющий вредоносный код.

- Code injections (SQL, PHP, ASP и т.д.)

  Это вид уязвимости, при котором становится возможно осуществить запуск исполняемого кода с целью получения доступа к системным ресурсам, несанкционированного доступа к данным либо выведения системы из строя.

  Наиболее простым примером может служить SQL-инъекция, осуществляющаяся через запрос к параметризованной веб-странице.

- Server-Side Includes (SSI) Injection

  Это вид уязвимости, использующий вставку серверных команд в HTML код или запуск их напрямую с сервера.

  В зависимости от типа операционной системы команды могут быть разными, например, команда, которая выводит на экран список файлов.

- Authorization Bypass

  Это вид уязвимости, при котором возможно получить несанкционированный доступ к учетной записи или документам другого

пользователя .

Простейший пример — просмотр страницы профиля пользователя, при котором в URL страницы передается userID. В этом случае злоумышленник может подставить вместо своего userID номер другого пользователя.

Примеров уязвимостей и атак существует огромное количество. Даже проведя полный цикл тестирования безопасности, нельзя быть на 100% уверенным, что система по-настоящему безопасна. Но можно быть уверенным в том, что процент несанкционированных проникновений, краж информации, потерь данных будет на несколько порядков ниже, чем до проведения испытаний.

Тестирование на отказ и восстановление

Тестирование на отказ и восстановление (Failover and Recovery Testing) проверяет тестируемый продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками программного обеспечения, отказами оборудования или проблемами связи (например, отказ сети). Целью данного вида тестирования является проверка систем восстановления (или дублирующих основной функционал систем), которые, в случае возникновения сбоев, обеспечат сохранность и целостность данных тестируемого продукта.

Тестирование на отказ и восстановление очень важно для систем, работающих по принципу «24x7». Если разрабатывается продукт, который будет работать, например в интернете, то без проведения данного вида тестирования просто не обойтись. Т.к. каждая минута простоя или потеря данных в случае отказа оборудования, может привести к убыткам, стоить потери клиентов и репутации на рынке.

Методика подобного тестирования заключается в симулировании различных условий сбоя и последующем изучении и оценке реакции защитных систем. В процессе подобных проверок выясняется, была ли достигнута требуемая степень восстановления системы после возникновения сбоя.

Объектом тестирования в большинстве случаев являются весьма вероятные эксплуатационные проблемы, такие как:

- Отказ электричества на компьютере-сервере
- Отказ электричества на компьютере-клиенте
- Незавершенные циклы обработки данных (прерывание работы фильтров данных, прерывание синхронизации).
- Объявление или внесение в массивы данных невозможных или ошибочных элементов.
- Отказ носителей данных

Данные ситуации могут быть воспроизведены, как только достигнута некоторая точка в разработке, когда все системы восстановления или дублирования готовы выполнять свои функции. Технически реализовать тесты можно следующими путями:

- Симулировать внезапный отказ электричества на компьютере (обесточить

компьютер).

- Симулировать потерю связи с сетью (выключить сетевой кабель, обесточить сетевое устройство)
- Симулировать отказ носителей (обесточить внешний носитель данных)
- Симулировать ситуацию наличия в системе неверных данных (специальный тестовый набор или база данных).

При достижении соответствующих условий сбоя и по результатам работы систем восстановления, можно оценить продукт с точки зрения тестирования на отказ. Во всех вышеперечисленных случаях, по завершении процедур восстановления, должно быть достигнуто определенное требуемое состояние данных продукта:

- Потеря или порча данных в допустимых пределах.
- Отчет или система отчетов с указанием процессов или транзакций, которые не были завершены в результате сбоя.

Стоит заметить, что тестирование на отказ и восстановление – это весьма продукт-специфичное тестирование. Разработка тестовых сценариев должна производиться с учетом всех особенностей тестируемой системы. Принимая во внимание довольно жесткие методы воздействия, стоит также оценить целесообразность проведения данного вида тестирования для конкретного программного продукта.

Тестирование производительности

Тестирование производительности — это вид тестирования, который проводится с целью определения, как быстро работает система или ее часть под определенной нагрузкой. Также часто служит для проверки и подтверждения других атрибутов качества системы, таких как масштабируемость, надежность и потребление ресурсов.

Тестирование производительности подразумевает под собой различные виды испытаний:

- Нагрузочное тестирование — это простейшая форма тестирования производительности. Нагрузочное тестирование обычно проводится для того, чтобы оценить поведение приложения под заданной ожидаемой нагрузкой. Этой нагрузкой может быть, например, ожидаемое количество одновременно работающих пользователей приложения, совершающих заданное число транзакций за интервал времени. Такой тип тестирования обычно позволяет получить время отклика всех самых важных бизнес-транзакций. В случае наблюдения за базой данных, сервером приложений, сетью и т. д., этот тип тестирования может также идентифицировать некоторые узкие места приложения.
- Стресс-тестирование обычно используется для понимания пределов пропускной способности приложения. Этот тип тестирования проводится для определения надежности системы во время экстремальных или диспропорциональных нагрузок и отвечает на вопросы о достаточной производительности системы в случае, если текущая нагрузка сильно превысит ожидаемый максимум.
- Тестирование стабильности проводится с целью убедиться в том, что приложение выдерживает ожидаемую нагрузку в течение длительного времени. При проведении этого вида тестирования осуществляется наблюдение за потреблением приложением памяти, чтобы выявить потенциальные утечки. Также важным, но часто незамеченным, фактором является деградация производительности, смысл которого сводится к тому, чтобы скорость обработки информации и/или время ответа приложения через длительное время работы были такими же или лучше, чем в самом начале теста.
- Конфигурационное тестирование — еще один из видов традиционного тестирования производительности. В этом случае вместо того, чтобы

тестировать производительность системы с точки зрения подаваемой нагрузки, тестируется эффект влияния на производительность изменений в конфигурации. Хорошим примером такого тестирования могут быть эксперименты с различными методами балансировки нагрузки. Конфигурационное тестирование также может быть совмещено с нагрузочным, стресс или тестированием стабильности.

В общих случаях тестирование производительности может служить разным целям:

- С целью демонстрации того, что система удовлетворяет критериям производительности.
- С целью определения производительность какой из двух или нескольких систем лучше.
- С целью определения, какой элемент нагрузки или часть системы приводит к снижению производительности.

Многие тесты на производительность делаются без попытки осмыслить их реальные цели. Перед началом тестирования всегда должен быть задан бизнес-вопрос: «Какую цель мы преследуем, тестируя производительность?». Ответы на этот вопрос являются частью технико-экономического обоснования (или business case) тестирования. Цели могут различаться в зависимости от технологий, используемых приложением, или его назначения, однако, чаще всего направлены на определение следующих характеристик:

- Параллелизм / Пропускная способность.

  Если конечными пользователями приложения считаются пользователи, выполняющие логин в систему в любой форме, то в этом случае крайне желательно достижение параллелизма. По определению это максимальное число параллельных работающих пользователей приложения, поддержка которого ожидается от приложения в любой момент времени. Модель поведения пользователя может значительно влиять на способность приложения к параллельной обработке запросов, особенно если он включает в себя периодически вход и выход из системы. Если концепция приложения не заключается в работе с конкретными конечными пользователями, то преследуемая цель для производительности будет основана на максимальной пропускной способности или числе транзакций в единицу времени. Хорошим примером в данном случае будет являться просмотр веб-страниц на

портале.

- Время ответа сервера.

  Эта концепция строится вокруг времени ответа одного узла приложения на запрос, посланный другим. Простым примером является HTTP 'GET' запрос из браузера рабочей станции на веб-сервер. Практически все приложения, разработанные для нагрузочного тестирования работают именно по этой схеме измерений. Иногда целесообразно ставить задачи по достижению производительности времени ответа сервера среди всех узлов приложения.

- Время отображения — одно из самых сложных для приложения для нагрузочного тестирования понятий, так как в общем случае они не используют концепцию работы с тем, что происходит на отдельных узлах системы, ограничиваясь только распознаванием периода времени в течение которого нет сетевой активности. Для того, чтобы замерить время отображения, в общем случае требуется включать функциональные тестовые сценарии в тесты производительности, но большинство приложений для тестирования производительности не включают в себя такую возможность.

- Требования к производительности.

  Очень важно детализировать требования к производительности и документировать их в каком-либо плане тестирования производительности. В идеальном случае это делается на стадии разработки требований при разработке системы, до проработки деталей ее дизайна. Однако тестирование производительности часто не проводится согласно спецификации, так как нет зафиксированного понимания о максимальном времени ответа для заданного числа пользователей. Тестирование производительности часто используется как часть процесса профилирования производительности. Его идея заключается в том, чтобы найти «слабое звено» — такую часть системы, соптимизировав время реакции которой, можно улучшить общую производительность системы. Определение конкретной части системы, стоящей на этом критическом пути, иногда очень непростая задача, поэтому некоторые приложения для тестирования включают в себя (или могут быть добавлены с помощью add-on’ов) инструменты, запущенные на сервере (агенты) и наблюдающие за временем выполнения транзакций, временем доступа к базе данных, перегрузками сети и другими

показателями серверной части системы, которые могут быть проанализированы вместе с остальной статистикой по производительности. Тестирование производительности может проводиться с использованием глобальной сети и даже в географически удаленных местах, если учитывать тот факт, что скорость работы сети Интернет зависит от местоположения. Оно также может проводиться и локально, но в этом случае необходимо настроить сетевые маршрутизаторы таким образом, чтобы появилась задержка, присутствующая во всех публичных сетях. Нагрузка, прилагаемая к системе, должна совпадать с реальным положением дел.

Требования к производительности должны адресовать следующие, как минимум, вопросы:

- Что охватывается тестом производительности? Какие подсистемы, компоненты, интерфейсы и т. д. должны быть протестированы?
- Если в тест включаются пользовательские интерфейсы, то сколько одновременно работающих в системе пользователей ожидается для каждого интерфейса (необходимо определить пиковые и нормальные значения).
- Как выглядит аппаратная составляющая тестируемой системы? (Необходимо описать все сервера и сетевое оборудование)
- Каков сценарий использования каждого компонента системы? (например, 20 % запросов составляет вход в систему, 40 % — поиск, 30 % — выбор элемента, 10 % — выход из системы)
- Каков сценарий использования системы? (в одном тесте на производительность могут быть задействованы разные сценарии использования каждого компонента)
- Каковы требования ко времени выполнения серии операций серверной части приложения?

Нагрузочное тестирование

Нагрузочное тестирование (Load Testing) — определение или сбор показателей производительности и времени отклика программно-технической системы или устройства в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к тестируемой системе.

Термин нагрузочное тестирование может быть использован в различных значениях в профессиональной среде тестирования программного обеспечения. В общем случае он означает практику моделирования ожидаемого использования приложения с помощью эмуляции работы нескольких пользователей одновременно. Таким образом, подобное тестирование больше всего подходит для мультипользовательских систем, чаще — использующих клиент-серверную архитектуру (например, веб-серверов). Однако и другие типы систем программного обеспечения могут быть протестированы подобным способом. Например, текстовый или графический редактор можно заставить прочесть очень большой документ; а финансовый пакет — сгенерировать отчет на основе данных за несколько лет. Наиболее адекватно спроектированный нагрузочный тест дает более точные результаты.

Основная цель нагрузочного тестирования заключается в том, чтобы, создав определенную ожидаемую в системе нагрузку (например, посредством виртуальных пользователей) и, обычно, использовав идентичное программное и аппаратное обеспечение, наблюдать за показателями производительности системы.

В идеальном случае в качестве критериев успешности нагрузочного тестирования выступают требования к производительности системы, которые формулируются и документируются на стадии разработки функциональных требований к системе до начала программирования основных архитектурных решений. Однако часто бывает так, что такие требования не были четко сформулированы или не были сформулированы вовсе. В этом случае первое нагрузочное тестирование будет являться пробным (exploratory load testing) и основываться на разумных предположениях об ожидаемой нагрузке и потреблении аппаратной части ресурсов.

Одним из оптимальных подходов в использовании нагрузочного тестирования для измерений производительности системы является тестирование на стадии ранней разработки. Нагрузочное тестирование на первых стадиях готовности архитектурного решения с целью определить его состоятельность называется 'Proof-of-Concept' тестированием.

Говоря о нагрузочном тестировании, рассматривают принципы, используемые при тестировании производительности в целом и применимые к любому типу тестирования производительности (в частности и к нагрузочному тестированию).

1. Уникальность запросов. Даже сформировав реалистичный сценарий работы с системой на основе статистики ее использования, необходимо понимать, что всегда найдутся исключения из этого сценария.
2. Время отклика системы. В общем случае время отклика системы подчиняется функции нормального распределения. В частности это означает, что имея достаточное количество измерений, можно определить вероятность с которой отклик системы на запрос попадет в тот или иной интервал времени.
3. Зависимость времени отклика системы от степени распределенности этой системы. Дисперсия нормального распределения времени отклика системы на запрос пропорциональна отношению количества узлов системы, параллельно обрабатывающих такие запросы и количеству запросов, приходящихся на каждый узел. То есть, на разброс значений времени отклика системы влияет одновременно количество запросов приходящихся на каждый узел системы и само количество узлов, каждый из которых добавляет некоторую случайную величину задержки при обработке запросов.
4. Разброс времени отклика системы. Можно также заключить, что при достаточно большом количестве измерений величины времени обработки запроса в любой системе всегда найдутся запросы, время обработки которых превышает определенные в требованиях максимумы; причем, чем больше суммарное время проведения эксперимента тем выше окажутся новые максимумы. Этот факт необходимо учитывать при формировании требований к производительности системы, а также при проведении регулярного нагрузочного тестирования.
5. Точность воспроизведения профилей нагрузки. Необходимая точность воспроизведения профилей нагрузки тем дороже, чем больше компонент содержит система. Часто невозможно учесть все аспекты профиля нагрузки для сложных систем, так как чем сложнее система, тем больше времени будет затрачено на проектирование, программирование и поддержку адекватного профиля нагрузки для нее, что не всегда является необходимостью. Оптимальный подход в данном случае заключается в

балансировании между стоимостью разработки теста и покрытием функциональности системы, в результате которого появляются допущения о влиянии на общую производительность той или иной части тестируемой системы.

Одним из результатов, получаемых при нагрузочном тестировании и используемых в дальнейшем для анализа, являются показатели производительности приложения:

1. Потребление ресурсов центрального процессора (CPU, %). Метрика, показывающая сколько времени из заданного определенного интервала было потрачено процессором на вычисления для выбранного процесса. В современных системах важным фактором является способность процесса работать в нескольких потоках, для того, чтобы процессор мог производить вычисления параллельно. Анализ истории потребления ресурсов процессора может объяснять влияние на общую производительность системы потоков обрабатываемых данных, конфигурации приложения и операционной системы, мультипоточности вычислений, и других факторов.
2. Потребление оперативной памяти (Memory usage, Mb). Метрика, показывающая количество памяти, использованной приложением. Использованная память может делиться на три категории:
    1. Virtual — объем виртуального адресного пространства, которое использует процессор. Этот объем не обязательно подразумевает, использование соответствующего дискового пространства или оперативной памяти. Виртуальное пространство конечно и процесс может быть ограничен в возможности загружать необходимые библиотеки.
    2. Private — объем адресного пространства, занятого процессором и не разделяемого с другими процессами.
    3. Working Set — набор страниц памяти, недавно использованных процессом. В случае, когда свободной памяти достаточно, страницы остаются в наборе, даже если они не используются. В случае, когда свободной памяти остается мало, использованные страницы удаляются. При работе приложения память заполняется ссылками на объекты, которые, в случае неиспользования, могут быть очищены специальным автоматическим процессом, «сборщиком мусора» (Garbage Collector). Время затрачиваемое процессором на очистку

памяти таким способом может быть значительным, в случае, когда процесс занял всю доступную память (в Java — так называемый «постоянный Full GC») или когда процессу выделены большие объемы памяти, нуждающиеся в очистке. На время, требующееся для очистки памяти, доступ процесса к страницам выделенной памяти может быть заблокирован, что может повлиять на конечное время обработки этим процессом данных.

3. Потребление сетевых ресурсов. Эта метрика не связана непосредственно с производительностью приложения, однако ее показатели могут указывать на пределы производительности системы в целом.
4. Работа с дисковой подсистемой (I/O Wait). Работа с дисковой подсистемой может значительно влиять на производительность системы, поэтому сбор статистики по работе с диском может помогать выявлять узкие места в этой области. Большое количество чтений или записей может приводить к простаиванию процессора в ожидании обработки данных с диска и в итоге увеличению потребления CPU и увеличению времени отклика.
5. Время выполнения запроса (request response time, ms). Время выполнения запроса приложением остается одним из самых главных показателей производительности системы или приложения. Это время может быть измерено на серверной стороне, как показатель времени, которое требуется серверной части для обработки запроса; так и на клиентской, как показатель полного времени, которое требуется на сериализацию/десериализацию, пересылку и обработку запроса. Надо заметить, что не каждое приложение для тестирования производительности может измерить оба этих времени.

Стресс-тестирование

Стресс-тестирование (Stress Testing) — один из видов тестирования программного обеспечения, которое оценивает надежность и устойчивость системы в условиях превышения пределов нормального функционирования. Стресс-тестирование особенно необходимо для «критически важного» программного обеспечения, однако также используется и для остального программного обеспечения. Обычно стресс-тестирование лучше обнаруживает устойчивость, доступность и обработку исключений системой под большой нагрузкой, чем то, что считается корректным поведением в нормальных условиях.

В общем случае методология стресс-тестирования основана на снятии и анализе показателей производительности приложения при нагрузках, значительно превышающих ожидаемые на стадии сопровождения и несет в себе цель определить выносливость или устойчивость приложения на случай всплеска активности по его использованию.

Необходимость стресс-тестирования диктуется следующими факторами:

- Большая часть всех систем разрабатываются с допущением о функционировании в нормальном режиме и даже в случае, когда допускается возможность увеличения нагрузки, реальные объемы ее увеличения не принимаются во внимание.
- В случае SLA-контракта (соглашения об уровне услуг) стоимость отказа системы в экстремальных условиях может быть очень велика.
- Обнаружение некоторых ошибок или дефектов в функционировании системы не всегда возможно с использованием других типов тестирования.
- Тестирования, проведенного разработчиком, может быть недостаточно для эмуляции условий при которых происходит отказ системы.
- Предпочтительнее быть готовым к обработке экстремальных условий системы, чем ожидать ее отказа.

Основные направления применения стресс-тестирования:

- Общее исследование поведения системы при пиковых нагрузках.
- Исследование обработки ошибок и исключительных ситуаций системой при пиковых нагрузках.

- Исследование узких мест системы или отдельных компонент при диспропорциональных нагрузках.
- Тестирование емкости системы.

Стресс-тестирование, как и нагрузочное тестирование также может быть использовано для регулярной оценки изменений производительности с целью получения для дальнейшего анализа динамики изменения поведения системы за длительный период.

В ходе проведения стресс-тестирования производят различные виды нагрузки системы:

- Пропорциональная нагрузка . Стресс-тестирование может применяться как для обособленных приложений, так и для распределенных систем с клиент-серверной архитектурой. Простейшим примером стресс-тестирования обособленного приложения может являться открытие файла размером в 50 Мб программой Notepad, входящей в комплект ОС Windows. Условия стресс-тестирования приложения обычно формируются исходя из критических бизнес-процессов его функциональности, определенными на стадии разработки требований и анализа рисков группой, ответственной за производительность. В общем случае в качестве условий для стресс-тестирования может использоваться линейно увеличенная ожидаемая нагрузка.
- Диспропорциональная нагрузка . В случае тестирования многозвенных распределенных систем необходимо учитывать уже не только фактический объем нагрузки, состоящей из множества элементов, но и их пропорции в общем объеме. Использование диспропорциональной нагрузки в стресс-тестах может также применяться для выявления узких мест отдельных компонент системы.
- Тестирование емкости (Capacity Testing) является одним из самых важных с точки зрения развития бизнеса направлений стресс-тестирования и самых сложных с точки зрения проведения и анализа. Тестирование емкости проводится с целью определить запас прочности системы при полном соответствии требованиям к производительности. При моделировании нагрузки для тестирования емкости системы учитывается как текущая нагрузка в виде количества и пропорций одновременно поступающих в систему запросов, так и ожидаемая в перспективе. Результатом тестирования емкости приложения или системы является

набор максимально допустимых характеристик нагрузки системы, при которых приложение или система отвечает требованиям к производительности, разработанным и документированным на этапе проектирования архитектуры.

Тестирование стабильности

Тестирование стабильности или надежности (Stability / Reliability Testing) — один из видов нагрузочного тестирования программного обеспечения, целью которого является проверка работоспособности приложения при длительном тестировании со средним (ожидаемым) уровнем нагрузки.

Перед тем как подвергать программное обеспечение экстремальным нагрузкам обычно проводят проверку стабильности в предполагаемых условиях работы, то есть погрузить продукт в полную рабочую атмосферу. При тестировании, длительность его проведения не имеет первостепенного значения, основная задача - наблюдая за потреблением ресурсов, выявить утечки памяти и проследить чтобы скорость обработки данных и/или время отклика приложения в начале теста и с течением времени не уменьшалась. В противном случае вероятны сбои в работе продукта и перезагрузки системы.

Конфигурационное тестирование

Конфигурационное тестирование (Configuration Testing) — специальный вид тестирования, направленный на проверку работы программного обеспечения при различных конфигурациях системы (заявленных платформах, поддерживаемых драйверах, при различных конфигурациях компьютеров и т.д.).

В зависимости от типа проекта конфигурационное тестирование может иметь разные цели:

- Проект по профилированию работы системы, целью которого является определить оптимальную конфигурацию оборудования, обеспечивающую требуемые характеристики производительности и времени реакции тестируемой системы.
- Проект по миграции системы с одной платформы на другую, целью которого является проверка объекта тестирования на совместимость с объявленным в спецификации оборудованием, операционными системами и программными продуктами третьих фирм.

Для клиент-серверных приложений конфигурационное тестирование можно условно разделить на два уровня (для других типов приложений может быть актуален только один):

1. Серверный
2. Клиентский

На первом (серверном) уровне, тестируется взаимодействие выпускаемого программного обеспечения с окружением, в которое оно будет установлено:

- Аппаратные средства (тип и количество процессоров, объем памяти, характеристики сети / сетевых адаптеров и т.д.)
- Программные средства (ОС, драйвера и библиотеки, стороннее программное обеспечение, влияющее на работу приложения и т.д.)

Основной упор здесь делается на тестирование с целью определения оптимальной конфигурации оборудования, удовлетворяющего требуемым характеристикам качества (эффективность, портативность, удобство сопровождения, надежность).

На следующем (клиентском) уровне, программное обеспечение

тестируется с позиции его конечного пользователя и конфигурации его рабочей станции. На этом этапе будут протестированы следующие характеристики: удобство использования, функциональность. Для этого необходимо будет провести ряд тестов с различными конфигурациями рабочих станций:

- Тип, версия и битность операционной системы (подобный вид тестирования называется кросс-платформенное тестирование)
- Тип и версия Web-барузера, в случае если тестируется Web приложение (подобный вид тестирования называется кросс-браузерное тестирование)
- Тип и модель видео адаптера (при тестировании игр это очень важно)
- Работа приложения при различных разрешениях экрана
- Версии драйверов, библиотек и т.д. (для JAVA приложений версия JAVA машины очень важна, тоже можно сказать и для .NET приложений касательно версии .NET библиотеки)
- и т.д.

Перед началом проведения конфигурационного тестирования рекомендуется:

- создавать матрицу покрытия (матрица покрытия - это таблица, в которую заносят все возможные конфигурации),
- проводить приоритезацию конфигураций (на практике, скорее всего, все желаемые конфигурации проверить не получится),
- шаг за шагом, в соответствии с расставленными приоритетами, проверяют каждую конфигурацию.

Уже на начальном этапе становится очевидно, что чем больше требований к работе приложения при различных конфигурациях рабочих станций, тем больше тестов необходимо будет провести. В связи с этим, становится очевидной необходимость автоматизации данного процесса тестирования с целью экономии времени и ресурсов.

Непрерывная интеграция

Непрерывная интеграция (Continuous Integration) — это практика разработки программного обеспечения, которая заключается в выполнении частых автоматизированных сборок проекта для скорейшего выявления и решения интеграционных проблем. В обычном проекте, где над разными частями системы разработчики трудятся независимо, стадия интеграции является заключительной. Она может непредсказуемо задержать окончание работ. Переход к непрерывной интеграции позволяет снизить трудоемкость интеграции и сделать ее более предсказуемой за счет наиболее раннего обнаружения и устранения ошибок и противоречий.

Непрерывная интеграция является одним из основных экстремального программирования.

Данная практика прелъявляет следующие требования к проекту:

- исходные коды и все, что необходимо для сборки и тестирования проекта, хранится в репозитории системы управления версиями;
- операции копирования из репозитория, сборки и тестирования всего проекта автоматизированы и легко вызываются из внешней программы.

Для обеспечения непрерывной интеграции на выделенном сервере организуется служба, в задачи которой входят:

- получение исходного кода из репозитория;
- сборка проекта;
- выполнение тестов;
- развёртывание готового проекта;
- отправка отчетов.

Локальная сборка может осуществляться:

- по внешнему запросу
- по расписанию,
- по факту обновления репозитория и по другим критериям.

Сборка по расписанию

В случае сборки по расписанию (daily build — ежедневная сборка), они, как правило, проводятся каждой ночью в автоматическом режиме — ночные сборки (чтобы к началу рабочего дня были готовы результаты тестирования). Для различения дополнительно вводится система нумерации сборок — обычно, каждая сборка нумеруется натуральным числом, которое увеличивается с каждой новой сборкой. Исходные тексты и другие исходные данные при взятии их из репозитория системы контроля версий помечаются номером сборки. Благодаря этому, точно такая же сборка может быть точно воспроизведена в будущем — достаточно взять исходные данные по нужной метке и запустить процесс снова. Это дает возможность повторно выпускать даже очень старые версии программы с небольшими исправлениями.

Преимущества:

- проблемы интеграции выявляются и исправляются быстро, что оказывается дешевле;
- немедленный прогон модульных тестов для свежих изменений;
- постоянное наличие текущей стабильной версии вместе с продуктами сборок — для тестирования, демонстрации, и т. п.
- немедленный эффект от неполного или неработающего кода приучает разработчиков к работе в итеративном режиме с более коротким циклом.

Недостатки

- затраты на поддержку работы непрерывной интеграции;
- потенциальная необходимость в выделенном сервере под нужды непрерывной интеграции;
- немедленный эффект от неполного или неработающего кода отучает разработчиков от выполнения периодических резервных включений кода в репозиторий.
- в случае использования системы управления версиями исходного кода с поддержкой ветвления, эта проблема может решаться созданием отдельной «ветки» (branch) проекта для внесения крупных изменений (код, разработка которого до работоспособного варианта займет несколько дней, но желательно более частое резервное копирование в репозиторий).

По окончании разработки и индивидуального тестирования такой ветки, она может быть объединена (merge) с основным кодом или «стволом» (trunk) проекта.

Методология жизненного цикла автоматизированного тестирования

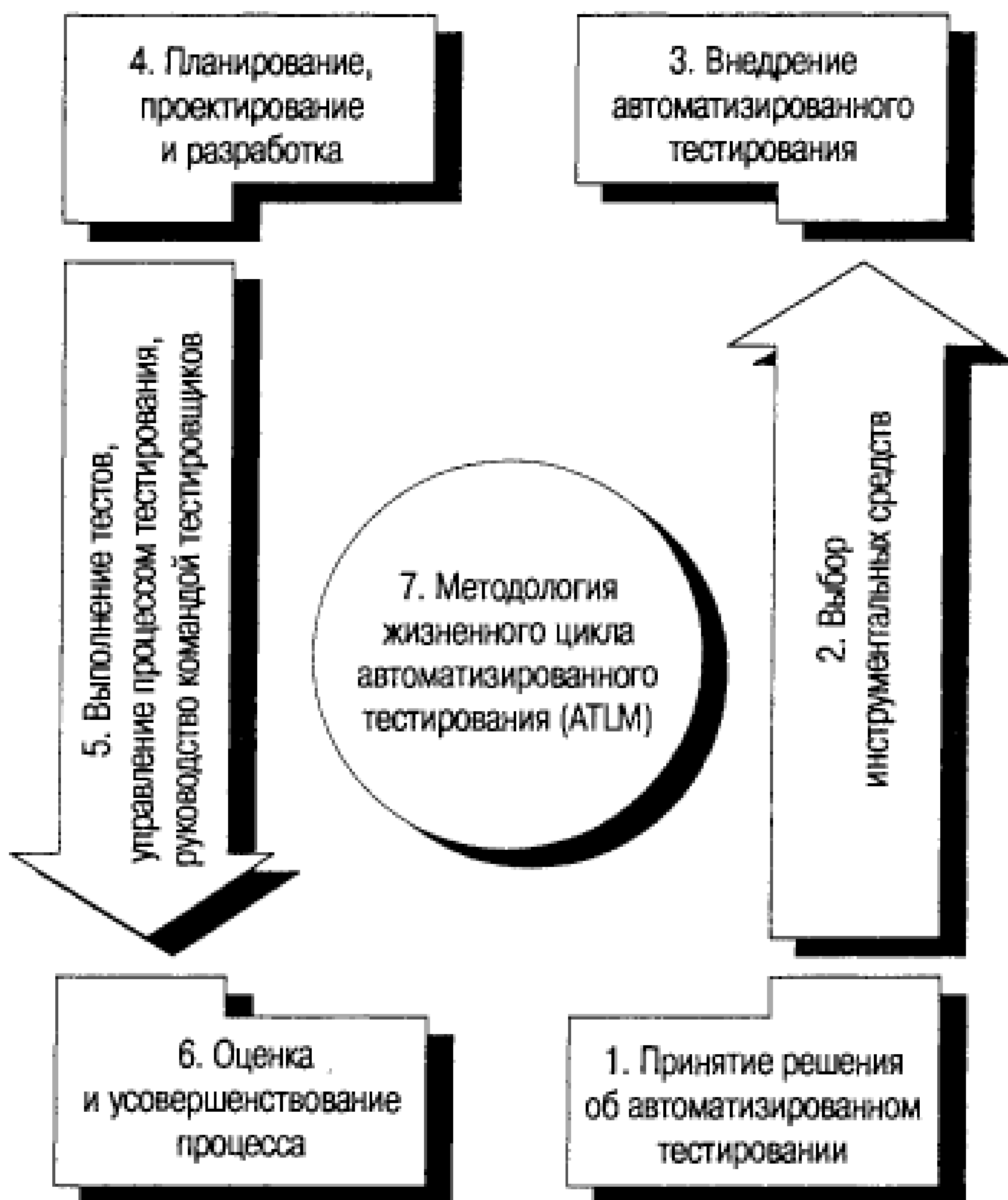

Рисунок 16: Методология ATLM [Дастин 2003]

В условиях использования гибких методологий разработки программного обеспечения возникло новое развитие методологии тестирования приложений — методология автоматизированного тестирования (ATLM).

ATLM-подход отражает преимущества современной технологии быстрой разработки приложений, когда пользователь привлекается к участию на ранних стадиях анализа, проектирования, разработки каждой версии программного обеспечения, создающейся при постоянном росте ее объемов.

Следуя ATLM, инженер по тестированию включается в жизненный цикл разработки системы на ранних стадиях проектирования и разработки каждой версии программного обеспечения, а именно во время бизнес-анализа на фазе определения требований. Это раннее включение в процесс позволяет команде инженеров по тестированию провести тщательную проверку спецификаций требований и проекта программного обеспечения, более полно понять потребности бизнеса, создать проект приемлемой тестовой среды и точный проект тестирования. Дополнительным преимуществом применения такой методологии тестирования, как ATLM, которая идет параллельно жизненному циклу разработки, является развитие близких деловых отношений между разработчиками и инженерами по тестированию, что способствует развитию сотрудничества и создает предпосылки для получения лучших результатов при проведении тестирования программных модулей, при комплексном и системном тестировании. Раннее включение тестирования важно потому, что требования или сценарии использования образуют основу или точку отсчета, относительно которой определяются требования по тестированию и измеряется успех тестирования. В частности, функциональные спецификации должны оцениваться как минимум с применением следующих критериев

- Полнота.

  Оцените ту область, для которой требование точно определено.

- Непротиворечивость.

  Убедитесь в том, что каждое требование не противоречит другим требованиям.

- Реализуемость.

  Оцените ту область, в которой требование действительно может быть реализовано с помощью имеющихся в наличии технологии, аппаратных средств, бюджета и уровня навыков занятого в проекте персонала.

- Возможность тестирования.

  Оцените ту область, в которой метод тестирования может гарантировать, что требование успешно реализовано.

Методология ATLM, предназначенная для поддержки работ по тестированию с применением средств автоматизации тестирования, включает в себя несколько этапов. Она поддерживает детализированные и взаимосвязанные работы, требуемые для принятия решения об использовании средств автоматизации тестирования. ATLM определяет процесс, необходимый для внедрения и использования средств автоматизированного тестирования, а также затрагивает разработку, проектирование, выполнение и управление тестированием. Эта методология поддерживает создание и управление тестовыми данными и средой тестирования, описывает способ разработки документации по тестированию, позволяющий отчитаться об ошибках. ATLM представляет собой структурированный подход к тестированию, который помогает команде инженеров по тестированию избежать таких распространенных ошибок, как:

- Внедрение в эксплуатацию средства автоматизированного тестирования без наличия процесса тестирования, что приводит к созданию тестовой программы, которую невозможно повторить и оценить.
- Реализация проекта тестирования без учета стандартов проектирования, что приводит к созданию тестовых скриптов, которые невозможно повторить и, следовательно, нельзя повторно использовать для версий программного обеспечения постоянно увеличивающегося объема.
- Попытка автоматизировать 100% требований к тестированию, когда применяемые средства не поддерживают автоматизацию всех необходимых тестов.
- Неправильный выбор инструментального средства.
- Запоздалое внедрение средства тестирования в жизненный цикл разработки приложения без предоставления необходимого времени на его установку и внедрение (т.е. без предоставления времени на обучение).
- Слишком позднее подключение тестировщиков к жизненному циклу разработки приложений, что приводит к плохому пониманию прикладного и системного проекта, а следовательно, к неполному тестированию.

Методология ATLM предназначена для того, чтобы гарантировать успешную реализацию автоматизированного тестирования. Методология предлагает следующие основные и подчиненные процессы:

1 Принятие решения об автоматизации .

При правильном внедрении автоматизированное тестирование должно оправдать возлагаемые на него надежды. Необходимо определить подход к разработке предложения о выборе средства тестирования с целью получения поддержки руководства. При этом производятся следующие работы и оценки:

1.1 Ожидания, связанные с автоматизированным тестированием

1.2 Польза применения автоматизированного тестирования

1.3 Получение поддержки руководства

2 Выбор инструментальных средств тестирования .

Поскольку в соответствии с критерием реализуемости средство автоматизированного тестирования должно удовлетворять большинству требований организации к тестированию, тестировщик обязан изучить среду системной разработки и другие нужды организации. Для этого должны быть реализованы следующие работы:

2.1 Исследование среды системной разработки

2.2 Изучение средств, имеющихся на рынке

2.3 Изучение и оценка инструментальных средств

2.4 Приобретение инструментальных средств

3 Внедрение автоматизированного тестирования .

3.1 Анализ процесса тестирования.

Проведение анализа процесса тестирования гарантирует, что весь процесс и стратегия тестирования определены и при необходимости могут быть модифицированы с целью успешного внедрения автоматизированного тестирования. Инженер по тестированию определяет и собирает оценки процесса тестирования, что позволит усовершенствовать процесс. Должны быть установлены цели, объекты и стратегии тестирования, а процесс тестирования должен быть документирован и доведен до сведения команды инженеров по тестированию. На этой фазе определяются виды тестирования, применимые к технической среде, а также тесты, поддерживаемые автоматизированными средствами. Оцениваются планы подключения пользователей, и анализируются навыки инженеров по тестированию на предмет соответствия требованиям и запланированным работам по тестированию.

3.2 Рассмотрение средств тестирования.

На фазе рассмотрения инструментального средства тестирования инженер по тестированию определяет, будет ли полезным для проекта включение средств автоматизированного тестирования в работы по тестированию. При этом учитываются требования к тестированию в проекте, существующая среда тестирования и людские ресурсы, пользовательская среда, платформа и функции приложения, подлежащие тестированию. Проверяется план-график проекта на наличие достаточного времени для установки средств тестирования и разработки иерархии требований. Потенциальные средства тестирования сопоставляются с требованиями к тестированию. Проверяется совместимость средств тестирования с приложением и средой. Исследуются альтернативные решения, позволяющие преодолеть проблемы, выявленные при проверке совместимости.

4 Планирование, проектирование и разработка тестирования

4.1 Документирование плана тестирования.

Фаза планирования тестирования включает в себя обзор долговременных работ по планированию тестирования. На этом этапе команда инженеров по тестированию определяет стандарты и основные направления создания процедур тестирования; аппаратные, программные и сетевые средства, необходимые для поддержки тестовой среды; требования к данным для тестирования; предварительный план-график тестирования; требования по оценке производительности; процедуру управления конфигурацией и средой тестирования; процедуру отслеживания дефектов и средства для его проведения. В план тестирования входят результаты всех предварительных фаз структурированной методологии тестирования (ATLM). Он определяет роли и обязанности, план-график тестирования в проекте, работы по планированию и проектированию тестирования, подготовку тестовой среды, риски и непредвиденные обстоятельства при тестировании, приемлемый уровень качества продукта (т.е. критерий завершения тестирования). Приложения к плану тестирования могут включать в себя процедуры тестирования, описание соглашений по именованию, стандарты формата процедур тестирования и матрицу зависимости процедуры тестирования. Установка среды тестирования — это часть планирования

тестирования. Команда инженеров по тестированию должна планировать, отслеживать и управлять работами по установке среды тестирования, что может занять немало времени. Инженеры по тестированию обязаны составить план-график и управлять работой по установке среды; установить аппаратные, программные и сетевые ресурсы среды тестирования; интегрировать и установить ресурсы среды тестирования; получить и обновить тестовые базы данных; разработать скрипты для установки среды и скрипты для тестового стенда.

4.2 Анализ требований к тестированию

4.3 Проектирование тестов.

На этой фазе определяются количество тестов, которые нужно выполнить, способы, с помощью которых можно получить доступ к тесту (например, пути или функции), и условия тестирования, которые должны соблюдаться. Необходимо определить и следовать стандартам проектирования тестирования.

4.4 Разработка тестов.

Чтобы автоматизированное тестирование можно было повторно использовать, повторять и сопровождать, необходимо определить и соблюдать стандарты разработки тестирования.

5 Выполнение и управление автоматизированным тестированием .

Команда инженеров по тестированию обязана выполнять скрипты тестирования и совершенствовать скрипты комплексного тестирования в соответствии с графиком выполнения процедуры тестирования. Кроме того, она должна оценить итоги выполнения тестирования так, чтобы избежать неправильных положительных или отрицательных оценок. Системные проблемы документируются в отчетах о системных проблемах. Необходимо добиться понимания со стороны разработчика системных проблем, проблем программного обеспечения, а также репликации данной проблемы. И, наконец, команда должна выполнить регрессионное тестирование и все прочие тесты и завершить отслеживание проблем. Для этого выполняются следующие работы:

5.1 Выполнение автоматизированного тестирования.

5.2 Основа тестового стенда

5.3 Отслеживание дефектов

5.4 Отслеживание хода тестирования

5.5 Оценка тестирования

6 Оценка и усовершенствование процесса .

Работа по критическому просмотру, или инспекции, и оценке должна проводиться на протяжении всего жизненного цикла тестирования. В течение всего жизненного цикла тестирования и последующих работ по выполнению тестирования должны оцениваться измерения и проводиться работы по окончательному критическому просмотру и оценке, что позволит усовершенствовать процесс.

6.1 Усовершенствование процесса тестирования после внедрения окончательной версии

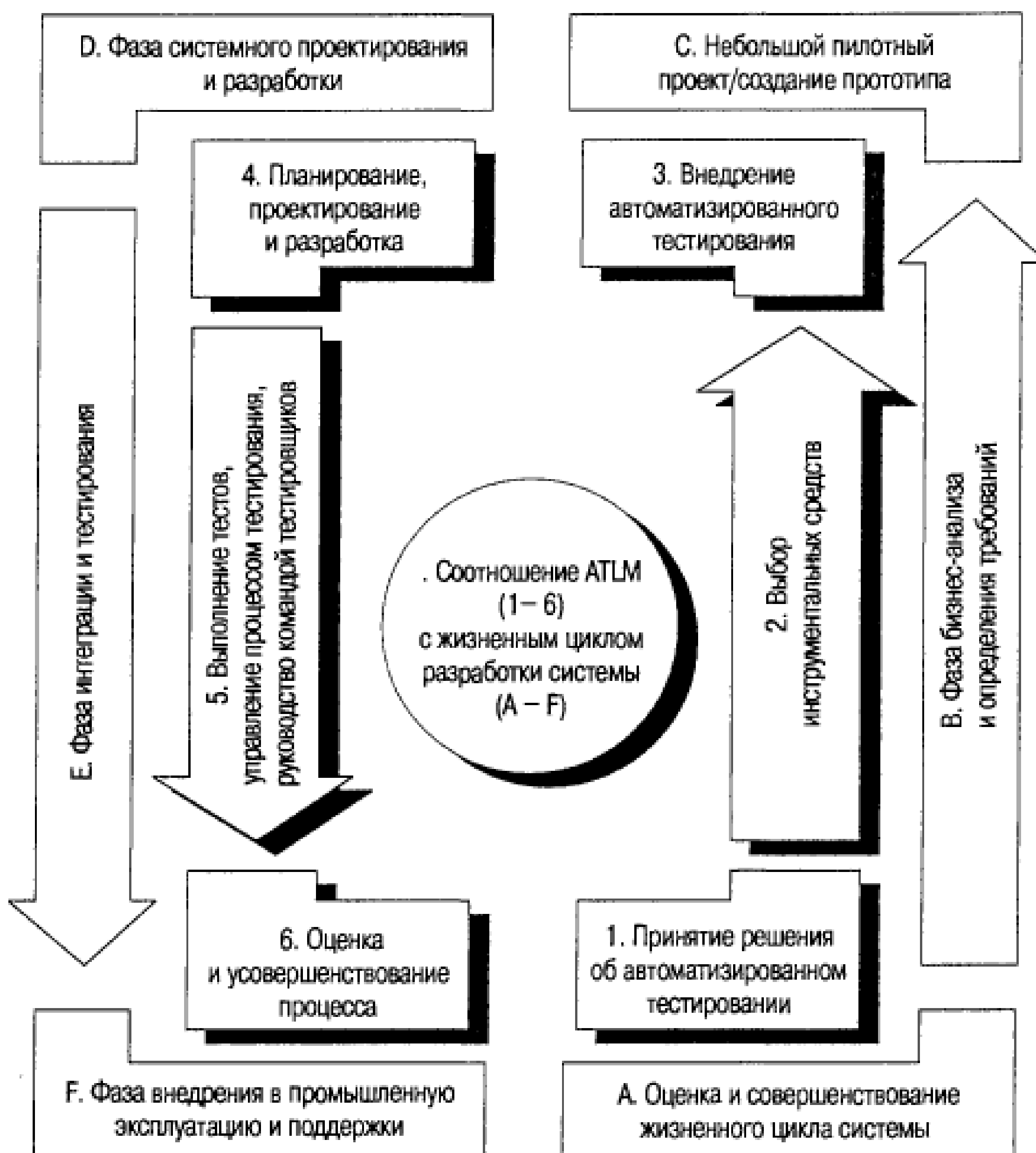

Рисунок 17: Соотношение ATLM и жизненного цикла разработки ПО [Дастин 2003]

Совершенно очевидно, методология автоматизированного тестирования должна внедряться с методологию разработки программного обеспечения.

Для получения наибольшего эффекта от выполнения программы тестирования подход ATLM должен реализовываться параллельно с жизненным циклом разработки системы. На рис. 17 показана взаимосвязь между ATLM и жизненным циклом разработки системы. Обратите внимание на то, что жизненный цикл разработки системы представлен на внешнем уровне. В правом нижнем углу рисунка показана фаза оценки процесса. Во время этой фазы возможности усовершенствования зачастую определяют то, что автоматизация тестирования действительно направлена на улучшение

жизненного цикла тестирования. Соответствующая фаза ATLM называется принятием решения об автоматизации тестирования.

На фазе бизнес-анализа и определения требований команда инженеров по тестированию проводит работы по выбору инструментальных средств тестирования (шаг 2 ATLM). Выбор инструментальных средств тестирования может осуществляться в любое время, но лучше это сделать, когда имеются требования к системе.

В идеале при внедрении автоматизированного тестирования (шаг 3 ATLM) группа разработчиков принимает в этом участие, создавая пилотный проект или небольшой прототип так, чтобы ликвидировать расхождения и извлечь определенный урок.

Работы по планированию, проектированию и разработке тестирования (шаг 4 ATLM) должны проводиться параллельно фазе проектирования и разработки системы. Некоторое планирование имело место в начале и в течение жизненного цикла разработки системы, но на этом этапе оно завершается.

Выполнение тестов и управление тестированием (шаг 5 ATLM) происходит совместно с фазой интеграции и тестирования жизненного цикла разработки системы. Системное тестирование и прочие работы по тестированию, такие как приемо-сдаточные испытания, производятся, когда создана первая версия.

Работы по критическому просмотру и оценке программы тестирования (шаг 6 ATLM) осуществляется в течение всего жизненного цикла и заканчивается на фазе внедрения в промышленную эксплуатацию и поддержки.

Системы автоматизированного тестирования

Одна из наиболее популярных систем автоматизированного тестирования — продукт компании Rational (ныне купленна IBM). В линейке продуктов есть различные компоненты, направленные на решение различных задач:

- IBM Rational Robot – универсальное средство автоматизации тестирования общего назначения для команд разработчиков, выполняющих функциональное тестирование клиент-серверных приложений. Дает возможность обнаруживать неполадки в программном обеспечении благодаря расширению сценариев тестирования средствами условной логики, позволяющей целиком охватить тестируемое приложение. Robot позволяет создавать сценарии тестирования с вызовом внешних библиотек DLL или исполняемых модулей.
- IBM Rational Performance Tester – инструмент нагрузочного и стрессового тестирования, с помощью которого можно выявлять проблемы системной производительности и их причины. Позволяет создавать тесты без написания кода и не требуя навыков программирования. Обеспечивает гибкие возможности моделирования и эмуляции различных пользовательских нагрузок. Выполняет сбор и интеграцию данных о серверных ресурсах с данными о производительности приложений, получаемыми в режиме реального времени.
- IBM Rational Functional Tester – набор средств автоматизированного тестирования, позволяющих выполнять функциональное и регрессионное тестирование, тестирование пользовательского интерфейса и тестирование, управляемое данными. Инструмент применяет технологию ScriptAssure (бесшовная проверка достоверности динамических данных) и функции поиска соответствия по шаблону, позволяющие повысить устойчивость сценариев тестирования в условиях частых изменений пользовательских интерфейсов приложений. Инженеры по тестированию могут выбрать язык сценариев для разработки и настройки тестов: Java в среде Eclipse или Microsoft Visual Basic .Net в среде Visual Studio .Net.
- IBM Rational Quality Manager – решение для реализации процессов управления тестированием и качеством, поддерживает сотрудничество участников групп по разработке программных продуктов, предоставляя им возможность обмениваться информацией, применять средства автоматизации для сокращения графиков выполнения проектов, а также

составлять отчеты по проектным показателям для принятия обоснованных решений. Rational Quality Manager может быть дополнен средством управления ресурсами тестирования Rational Test Lab, обеспечивающим учет ресурсов тестирования (серверов), их бронирование, автоматизацию развертывания тестовой среды на сервере и запуск скриптов тестирования, а также отчетность по использованию ресурсов тестирования.

Существенным недостатком этой системы автоматизированного тестирования является ее цена.

Еще одна крупная система автоматизации процесса тестирования — продукт компании Mercury (ныне куплена HP).

- HP QualityCenter – программный продукт, представляющий собой интегрированный пакет инструментов на платформе Web, предназначенных для построения и поддержки процесса тестирования приложений, а также обеспечения тесного взаимодействия команды из специалистов по тестированию. Включает в себя модули управления требованиями, релизами и циклами, тестовые примеры, а также модуль тестирования и аналитический портал отчетности.
- HP QuickTest Professional – набор средств автоматизации функционального и регрессионного тестирования программных систем, созданных с помощью основных платформ разработки. Продукт поддерживает такие среды, как Windows Presentation Foundation, Macromedia Flex, Ajax, Delphi, PowerBuilder, .Net, J2EE, обеспечивает работу с Web-сервисами, а также учитывает особенности ERP- и CRM-приложений.
- HP LoadRunner – программный продукт для автоматизации нагрузочного тестирования широкого набора программных сред и протоколов. Поддерживает SOA, работу с Web-сервисами, Ajax, RDP, SQL, продуктами Citrix, платформы Java, .Net, а также все основные ERP- и CRM-приложения от PeopleSoft, Oracle, SAP и Siebel. Пакет HP LoadRunner включает в себя более 60 мониторов сбора данных о тестируемой инфраструктуре и предоставляет детальную диагностику по работе приложений.

Другие средства автоматизированного тестирования:

- Selenium — средство автоматизации тестирования веб-приложений,

бесплатная, с открытым исходным кодом;

- Watir — созданный на Ruby набор библиотек для автоматизации тестирования веб-приложений, бесплатная, с открытым исходным кодом;
- pywinauto — библиотека для языка программирования Python для автоматизации процесса тестирования интерфейсов windows-приложений, бесплатная, с открытым исходным кодом;
- JMeter — система нагрузочного тестирования для веб-приложений, бесплатна, с открытым исходным кодом;
- LDTP — система автоматизации процесса тестирования приложений Linux и других операционных систем, бесплатна, с открытым исходным кодом;

Литература

1. Арчибальд Р. Д. Управление высоко-технологичными программами и проектами / Рассел Д. Арчибальд. — М.: ДКМ Пресс; Компаний АйТи, 2006. — 472 с.
2. Ambler S.W., One Piece at a Time. 2004
3. Ambler, S.W. (2005). The Elements of UML 2.0 Style. New York: Cambridge University Press.
4. Boehm B.W. A Spiral Model of Software Development and Enhancement, Computer, May 1988
5. Скотт Амблер. Гибкие технологии: экстремальное программирование и унифицированный процесс разработки. Пер. с англ. - Спб. ЗАО Издательский дом «Питер», 2005
6. Jacobson, I., Booch, G., Rumbaugh, J. The Unified Software Development Process. Reading, MA: Addison-Wesley, 1999.
7. Humphrey, W.S. Managing the Software Process. Reading, MA: Addison-Wesley,1989.
8. Брукс П.Ф. Мифический человеко-месяц или как создаются программные системы. - СПб.:Символ-Плюс, 2001.
9. http://ru.wikipedia.org/wiki/Гибкая_методология_разработки
10. http://gouspo.ru
11. http://swebok.sorlik.ru "Основы программной инженерии" 2004-2010 Сергей Орлик
12. SWEBOK 2004 by The Institute of Electrical and Electronics Engineers, Inc.
13. Калбертсон Роберт, Браун Крис, Кобб Гэри, Быстрое тестирование — М. ООО «ИД Вильямс», 2002
14. Криспин Лайза, Грегори Джанет, Гибкое тестирование: практическое руководство для тестироващиков ПО и гибких команд — Пер. с англ. — М. ООО «ИД Вильямс», 2010
15. Элфрид Дастин, Джефф Рэшка, Джон Пол, Автоматизированное тестирование программного обеспечения. Внедрение, управление и эксплуатация — Пер. с англ. — М.: ЛОРИ, 2003
16. Burnstein I, Practical Software Testing. A process-oriented approach, —

Springer-Verlag, New York, 2003

17. Nielsen J, Ten Usability Heuristics, http://www.useit.com/papers/heuristic/heuristic_list.html

18. http://www.protesting.ru/

19. Бек Кент. Экстремальное программирование. Разработка через тестирование, СПб.: Питер, 2003.

20. Влиссидес Дж. Применение шаблонов проектирования. Дополнительные штрихи, М.: Вильямс, 2003

21. Керниган Б., Пайк Р. Практика программирования, Пер. с англ. М.: Вильямс, 2004

22. Поль М. Дюваль, Стивен М. Матиас III, Эндрю Гловер Непрерывная интеграция: улучшение качества программного обеспечения и снижение риска. Пер. с англ. М.: Вильямс, 2008.

23. Мэри Поппендик, Том Поппендик. Бережливое производство программного обеспечения: от идеи до прибыли. Пер. с англ. М.: Вильямс, 2009.