Курс лекций по дисциплине

«Оценка качества программного обеспечения»

ТЕМА 1.Введение. Место верификации среди процессов разработки программного обеспечения (лекция 1)

Верификация – это процесс определения, выполняют ли программные средства и их компоненты требования, наложенные на них в последовательных этапах жизненного цикла разрабатываемой программной системы.

Основная цель верификации состоит в подтверждении того, что программное обеспечение соответствует требованиям. Дополнительной целью является выявление и регистрация дефектов и ошибок, которые внесены во время разработки или модификации программы.

Верификация является неотъемлемой частью работ при коллективной разработке программных систем. При этом в задачи верификации включается контроль результатов одних разработчиков при передаче их в качестве исходных данных другим разработчикам.

Для повышения эффективности использования человеческих ресурсов при разработке, верификация должна быть тесно интегрирована с процессами проектирования, разработки и сопровождения программной системы.

Заранее разграничим понятия верификации и отладки. Оба этих процесса направлены на уменьшение ошибок в конечном программном продукте, однако отладка – процесс, направленный на локализацию и устранение ошибок в системе, а верификация – процесс, направленный на демонстрацию наличия ошибок и условий их возникновения.

Кроме того, верификация, в отличие от отладки – контролируемый и управляемый процесс. Верификация включает в себя анализ причин возникновения ошибок и последствий, которые вызовет их исправление, планирование процессов поиска ошибок и их исправления, оценку полученных результатов. Все это позволяет говорить о верификации, как о процессе обеспечения заранее заданного уровня качества создаваемой программной системы.

1.1. Жизненный цикл разработки программного обеспечения

Коллективная разработка, в отличие от индивидуальной, требует четкого планирования работ и их распределения во время создания программной

системы. Один из способов организации работ состоит в разбиении процесса разработки на отдельные последовательные стадии, после полного прохождения которых получается конечный продукт или его часть. Такие стадии называют жизненным циклом разработки программной системы. Как правило, жизненный цикл начинается с формирования общего представления о разрабатываемой системе и их формализации в виде требований верхнего уровня. Завершается жизненный цикл разработки вводом системы в эксплуатацию. Однако, нужно понимать, что разработка – только один из процессов, связанных с программной системой, которая также имеет свой жизненный цикл. В отличие от жизненного цикла разработки системы, жизненный цикл самой системы заканчивается выводом ее из эксплуатации и прекращением ее использования.

Жизненный цикл программного обеспечения – совокупность итерационных процедур, связанных с последовательным изменением состояния программного обеспечения от формирования исходных требований к нему до окончания его эксплуатации конечным пользователем.

В контексте данного курса практически не будут затрагиваться такие этапы жизненного цикла, как системная интеграция и сопровождение. Для целей курса достаточно ограничиться упрощенным представлением, что после реализации кода и доказательства его соответствия требованиям разработка ПО завершается.

1.2. Модели жизненного цикла

Любой этап жизненного цикла имеет четко определенные критерии начала и окончания. Состав этапов жизненного цикла, а также критерии, в конечном итоге определяющие последовательность этапов жизненного цикла, определяется коллективом разработчиков и/или заказчиком. В настоящее время существует несколько основных моделей жизненного цикла, которые могут быть адаптированы под реальную разработку.

1.2.1. Каскадный жизненный цикл

Каскадный жизненный цикл (иногда называемый водопадным) основан на постепенном увеличении степени детализации описания всей разрабатываемой системы. Каждое повышение степени детализации определяет переход к следующему состоянию разработки .

На первом этапе составляется концептуальная структура системы, описываются общие принципы ее построения, правила взаимодействия с окружающим миром – определяются системные требования.

На втором этапе по системным требованиям составляются требования к программному обеспечению – здесь основное внимание уделяется

функциональности программной компоненты, программным интерфейсам. Естественно, все программные комплексы выполняются на какой-либо аппаратной платформе. Если в ходе проекта требуется также разработка аппаратной компоненты, параллельно с требованиями к программному обеспечению идет подготовка требований к аппаратному обеспечению.

На третьем этапе на основе требований к программному обеспечению составляется детальная спецификация архитектуры системы - описываются разбиение системы по конкретным модулям, интерфейсы между ними, заголовки отдельных функций и т.п.

На четвертом этапе пишется программный код, соответствующий детальной спецификации, на пятом этапе выполняется тестирование – проверка соответствия программного кода требованиям, определенным на предыдущих этапах.

Особенность каскадного жизненного цикла состоит в том, что переход к следующему этапу происходит только тогда, когда полностью завершены все работы предыдущего этапа. То есть сначала полностью готовятся все требования к системе, затем по ним полностью готовятся все требования к программному обеспечению, полностью разрабатывается архитектура системы и так далее до тестирования.

Естественно, что в случае достаточно больших систем такой подход себя не оправдывает. Работа на каждом этапе занимает значительное время, а внесение изменений в первичные документы либо невозможно, либо вызывает лавинообразное изменения на всех других этапах.

Как правило, используется модификация каскадной модели, допускающая возврат на любой из ранее выполненных этапов. При этом фактически возникает дополнительная процедура принятия решения.

Действительно если тесты обнаружили несоответствие реализации требованиям, то причина может крыться: (а) в неправильном тесте, (б) в ошибке кодирования (реализации), (в) в неверной архитектуре системы, (г) некорректности требований к программному обеспечению и т.д. Все эти случаи требуют анализа для принятия решения о том, на какой этап жизненного цикла надо возвратиться для устранения обнаруженного несоответствия.

1.2.2. V-образный жизненный цикл

В качестве своеобразной «работы над ошибками» классической каскадной модели стала применяться модель жизненного цикла, содержащая процессы двух видов – основные процессы разработки, аналогичные процессам

каскадной модели и процессы верификации, представляющие собой цепь обратной связи по отношению к основным процессам.

Таким образом, в конце каждого этапа жизненного цикла разработки, а зачастую и в процессе выполнения этапа, осуществляется проверка взаимной корректности требований различных уровней. Данная модель позволяет более оперативно проверять корректность разработки, однако, как и в каскадной модели предполагается, что на каждом этапе разрабатываются документы, описывающие поведение всей системы в целом.

1.2.3. Спиральный жизненный цикл

Оба рассмотренных типа жизненных циклов предполагают, что заранее известны все требования пользователей или, по крайней мере, предполагаемые пользователи системы настолько квалифицированы, что могут высказывать свои требования к будущей системе, не видя ее перед глазами.

Естественно, такая картина достаточно утопична, поэтому постепенно появилось решение, исправляющее основной недостаток V-образного жизненного цикла – предположение о том, что на каждом этапе разрабатывается очередное полное описание системы. Этим решением стала спиральная модель жизненного цикла .

В спиральной модели разработка системы происходит повторяющимися этапами – витками спирали. Каждый виток спирали – один каскадный или V-образный жизненный цикл. В конце каждого витка получается законченная версия системы, реализующая некоторый набор функций. Затем она предъявляется пользователю, на следующий виток переносится вся документация, разработанная на предыдущем витке, и процесс повторяется.

Таким образом, система разрабатывается постепенно, проходя постоянные согласования с заказчиком. На каждом витке спирали функциональность системы расширяется постепенно дорастая до полной.

1.2.4. Экстремальное программирование

Реалии последних лет показали, что для систем, требования к которым изменяются достаточно часто, необходимо еще больше уменьшить длительность витка спирального жизненного цикла. В связи с этим сейчас стали весьма популярными быстрые жизненные циклы разработки, например жизненный цикл в методологии eXtreme Programming (XP).

Основная идея жизненного цикла экстремального подхода – максимальное укорачивание длительности одного этапа жизненного цикла и тесное взаимодействие с заказчиком. По сути, на каждом этапе происходит

реализация и тестирование одной функции системы, после завершения которых, система сразу передается заказчику на проверку или эксплуатацию.

Основная проблема данного подхода – интерфейсы между модулями, реализующими эту функцию. Если во всех предыдущих типах жизненного цикла интерфейсы достаточно четко определяются в самом начале разработки, поскольку заранее известны все модули, то при экстремальном подходе интерфейсы проектируются «на лету», вместе с разрабатываемыми модулями.

1.2.5. Сравнение различных типов жизненного цикла и вспомогательные процессы

Особенности рассмотренных выше типов жизненного цикла сведены в таблицу 1. Из нее можно видеть, что различные типы жизненных циклов применяются в зависимости от планируемой частоты внесения изменений в систему, сроков разработки и ее сложности. Жизненные циклы с более короткими фазами больше подходят для разработки систем, требования к которым еще не устоялись и вырабатываются во взаимодействии с заказчиком системы во время ее разработки.

Таблица 1 Сравнение различных типов жизненного цикла

Тип жизненного цикла	Длина цикла	Верификация и внесение изменений	Интеграция отдельных компонент системы
Каскадный	Все этапы разработки системы. Длинный	В конце разработки всей системы. Редко.	Четко определенные до начала кодирования интерфейсы.
V-образный	Все этапы разработки системы. Длинный	В конце полной разработки каждого из этапов системы. Средне.	Редко изменяемые интерфейсы.
Спиральный	Разработка одной версии системы. Средний.	В конце разработки каждого из этапов версии системы. Средне.	Периодически изменяемые интерфейсы, редко меняемые в пределах версии.
XP	Разработка одной истории.	В конце разработки каждой истории. Очень часто	Часто изменяемые интерфейсы.

	Короткий.		

В приведенном выше описании различных моделей жизненного цикла по сути затрагивался только один процесс – процесс разработки системы. На самом деле в любой модели жизненного цикла можно увидеть четыре вида процессов:

1. Основной процесс разработки
2. Процесс верификации
3. Процесс управления разработкой
4. Вспомогательные (поддерживающие) процессы

Процесс верификации – процесс, направленный на проверку корректности разрабатываемой системы и определения степени ее соответствия требованиям заказчика. Подробному рассмотрению этого процесса и посвящен данный учебный курс.

Процесс управления разработкой – отдельная дисциплина, на управление очень сильно влияет тип жизненного цикла основного процесса разработки. По сути, чем короче один этап жизненного цикла, тем активнее управление, и тем больше задач стоит на менеджере проекта. При классических схемах достаточно просто построить иерархическую пирамиду подчиненности, у которой каждый нижестоящий менеджер отвечает за разработку определенной части системы. В XP-подходе нет жесткого разделения системы на части и менеджер должен охватывать все истории. При этом процесс управления активен на протяжении всего жизненного цикла основного процесса разработки.

Вспомогательные (поддерживающие) процессы обеспечивают своевременное создание всего, что может понадобиться разработчику или конечному пользователю. Сюда входит подготовка пользовательской документации, подготовка приемо-сдаточных тестов, управление конфигурациями и изменениями, взаимодействие с заказчиком и т.д. Вообще говоря, вспомогательные процессы могут существовать в течение всего жизненного цикла разработки, а могут быть своеобразными стыкующими звеньями между процессом разработки и процессом эксплуатации.

В конце данного курса особое внимание будет уделено наиболее значимым поддерживающим процессам - процессу управления конфигурациями и процессу обеспечения гарантии качества.

Основная цель процесса управления конфигурациями – обеспечение целостности всех данных, возникающих в процессе коллективной разработки. Под целостностью понимается, прежде всего, идентифицируемость, доступность этих данных в любой момент времени и

недопущение несанкционированных изменений. Важным аспектом при этом становится процесс управления изменениями данных, т.е. планирование и утверждение любых изменений в проектную документацию или программный код, а также определение области влияния этих изменений.

Процесс гарантии качества обеспечивает проведение проверок, гарантирующих, что процесс разработки удовлетворяет набору определенных требований (стандартов), необходимых для выпуска качественной продукции. Фактически он проверяет, что все предусмотренные стандартами разработки процедуры выполняются и при выполнении соблюдаются декларированные для них правила.

Нужно особо отметить, что процесс гарантии качества не гарантирует разработку качественной программной системы. Он гарантирует только лишь, что процессы разработки построены и выполняются таким образом, чтобы не снижать качество продукции.

Требования, предъявляемые к организации работы, необходимой для выпуска качественной продукции, оформлены в виде стандартов качества. Наиболее часто цитируемая и известная группа стандартов качества – серия стандартов ISO 9000. В дополнение к ним существует стандарт, содержащий требования к жизненному циклу разработки ПО – ISO 12207.

В реальной практике сейчас наиболее широко применяется стандарт ISO 12207, в отечественных госструктурах используются стандарты серии ГОСТ 34.

Стандарты комплекса ГОСТ 34 на создание и развитие АС - обобщенные, но воспринимаемые как весьма жесткие по структуре ЖЦ и проектной документации.

Международный стандарт ISO/IEC 12207 на организацию жизненного цикла продуктов программного обеспечения (ПО) – содержит общие рекомендации по организации жизненного цикла, не постулируя при этом его жесткой структуры.

1.3. Современные технологии разработки программного обеспечения:

1.3.1. Microsoft Solutions Framework

Microsoft Solutions Framework (MSF) - это методология ведения проектов и разработки решений, базирующаяся на принципах работы над продуктами самой фирмы Microsoft, и предназначенная для использования в организациях, нуждающихся в концептуальной схеме для построения современных решений [].

Microsoft Solutions Framework является схемой для принятия решений по планированию и реализации новых технологий в организациях. MSF включает обучение, информацию, рекомендации и инструменты для идентификации и структуризации информационных потоков бизнес-процессов и всей информационной инфраструктуры новых технологий.

Microsoft Solutions Framework представляет собой хорошо сбалансированный набор методик организации процесса разработки, который может быть адаптирован под потребности практически любого коллектива разработчиков. MSF содержит не только рекомендации общего характера, но и предлагает адаптируемую модель коллектива разработчиков, определяющую взаимоотношения внутри коллектива, гибкую модель проектного планирования, основанного на управлении проектными группами, а также набор методик для оценки рисков.

В момент подготовки данного учебного курса последней версией MSF была 3.1, при этом существовали документы, относящиеся к версии 4.0 beta.

MSF состоит из двух моделей:

- модель проектной группы;
- модель процессов;

и трех дисциплин:

- управление проектами;
- управление рисками;
- управление подготовкой.

В MSF нет роли «менеджер проекта» и иерархии руководства, управление разработкой распределено между руководителями отдельных проектных групп внутри коллектива, выполняющих следующие задачи:

- Управление программой
- Разработка
- Тестирование
- Управление выпуском
- Удовлетворение потребителя
- Управление продуктом

Жизненный цикл процессов в MSF сочетает водопадную и спиральную модели разработки: проект реализуется поэтапно, с наличием соответствующих контрольных точек, а сама последовательность этапов может повторяться по спирали (Рис. 4).

Рис. 4 Жизненный цикл в MSF

При таком подходе от водопадной модели берется простота планирования, от классической спиральной – легкость модификаций. При этом благодаря промежуточным контрольным точкам и обратной спирали верификации облегчается взаимодействие с заказчиком.

При управлении проектом четко ставится цель, которую необходимо достичь в результате и учитываются ограничения, накладываемые на проект. Все виды ограничений могут быть отнесены к одному из трех видов: ограничения ресурсов, ограничения времени и ограничения возможностей. Эти три вида ограничений и приоритетность задач по их преодолению образую треугольник приоритетов в MSF (Рис. 5).

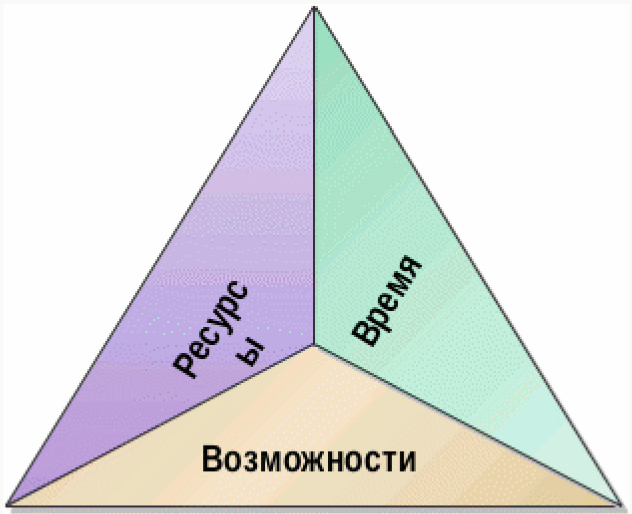


Рис. 5 Треугольник приоритетов в MSF

Треугольник приоритетов является основой для матрицы компромиссов – заранее утвержденных представлений о том, какие аспекты процесса

разработки будут четко заданы, а какие будут согласовываться или приниматься как есть.

Microsoft выпустила среду разработки, в полной мере поддерживающей основные идеи MSF – Microsoft Visual Studio 2005 Team Edition. Это первый программный комплекс, представляющий собой не среду разработки для индивидуальных членов коллектива, а комплексное средство поддержки коллективной работы.

В состав Visual Studio Team Edition входит специальная редакция для тестировщиков – Team Edition for Software Testers (Рис. 6). Материалы семинарских занятий по данному курсу ориентированы на эту среду разработки, в то время как лекционные материалы ориентированы на изложение общих принципов и методик тестирования.

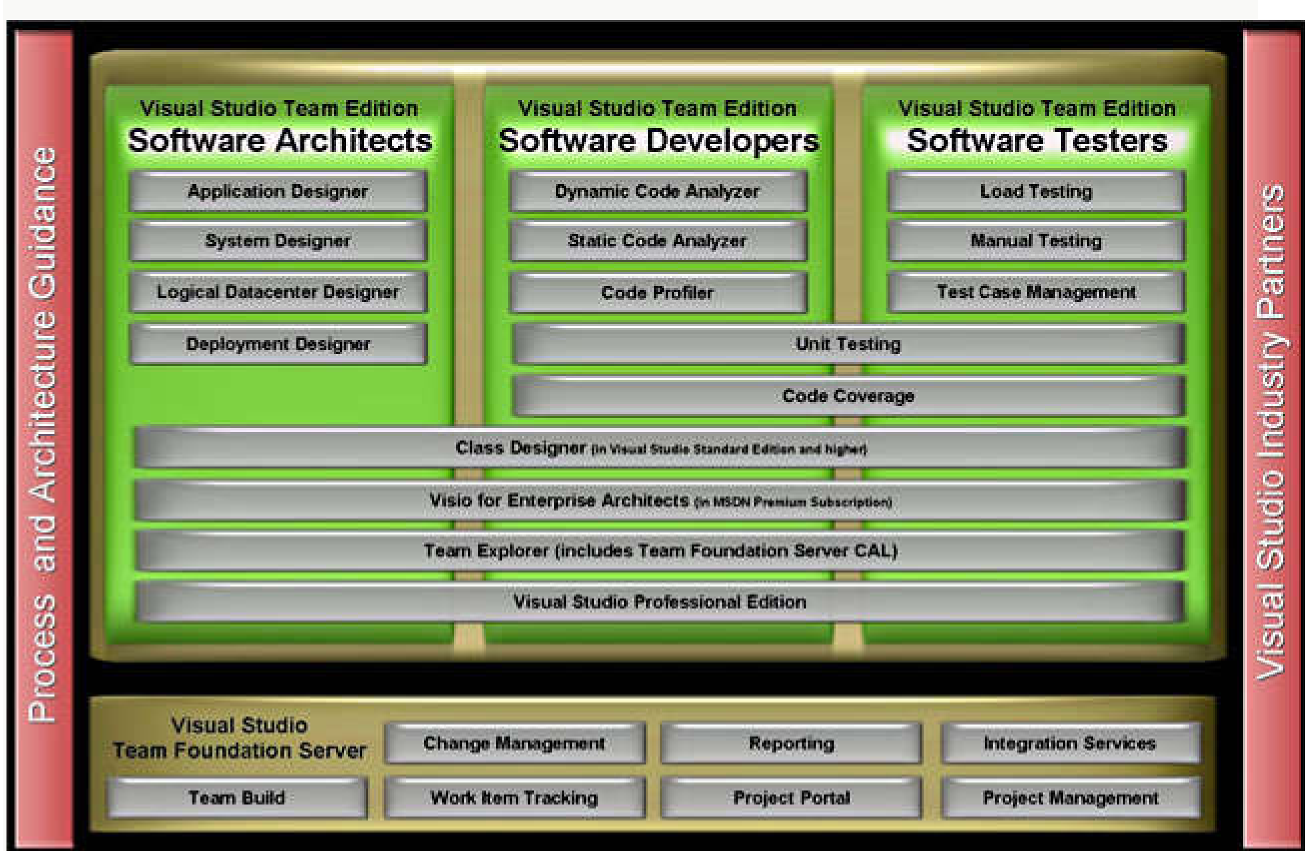


Рис. 6 Структура Microsoft Visual Studio 2005 Team System

1.3.2. Rational Unified Process

Rational Unified Process – это методология создания программного обеспечения, оформленная в виде размещаемой на Web базы знаний, которая снабжена поисковой системой.

Продукт Rational Unified Process (RUP) разработаниподдерживается Rational Software. Он регулярно обновляется с целью учета передового опыта и улучшается за счет проверенных на практике результатов.

RUP обеспечивает строгий подход к распределению задач и ответственности внутри рганизации-разработчика. Его предназначение заключается в том, чтобы гарантировать создание точно в срок и в рамках установленного бюджета качественного ПО, отвечающего нуждам конечных пользователей.

RUP способствует повышению производительности коллективной разработки и предоставляет лучшее из накопленного опыта по созданию ПО, посредством руководств, шаблонов и наставлений по пользованию инструментальными средствами для всех критически важных работ, в течение жизненного цикла создания и сопровождения ПО. Обеспечивая каждому члену группы доступ к той же самой базе знаний, вне зависимости от того, разрабатывает ли он требования, проектирует, выполняет тестирование или управляет проектом - RUP гарантирует, что все члены группы используют общий язык моделирования, процесс, имеют согласованное видение того, как создавать ПО. В качестве языка моделирования в общей базе знаний используется Unified Modeling Language (UML), являющийся международным стандартом.

Особенностью RUP является то, что в результате работы над проектом создаются и совершенствуются модели. Вместо создания громадного количества бумажных документов, RUP опирается на разработку и развитие семантически обогащенных моделей, всесторонне представляющих разрабатываемую систему. RUP – это руководство по тому, как эффективно использовать UML. Стандартный язык моделирования, используемый всеми членами группы, делает понятными для всех описания требований, проектирование и архитектуру системы.

RUP поддерживается инструментальными средствами, которые автоматизируют многие элементы процесса разработки. Они используются для создания и совершенствования различных промежуточных продуктов на различных этапах процесса создания ПО, например, при визуальном моделировании, программировании, тестировании и т.д.

RUP – это конфигурируемый процесс, поскольку, вполне понятно, что невозможно создать единого руководства на все случаи разработки ПО. RUP пригоден как для маленьких групп разработчиков, так и для больших организаций, занимающихся созданием ПО. В основе RUP лежит простая и понятная архитектура процесса, которая обеспечивает общность для целого семейства процессов. Более того, RUP может конфигурироваться для учета различных ситуаций. В его состав входит Development Kit, который

обеспечивает поддержку процесса конфигурирования под нужды конкретных организаций.

RUP описывает, как эффективно применять коммерчески обоснованные и практически опробованные подходы к разработке ПО для коллективов разработчиков, где каждый из членов получает преимущества от использования передового опыта в:

- итерационной разработке ПО,
- управлении требованиями,
- использовании компонентной архитектуры,
- визуальном моделировании,
- тестировании качества ПО,
- контроле за изменениями в ПО.

RUP организует работу над проектом в терминах последовательности действий (workflows), продуктов деятельности, исполнителей и других статических аспектов процесса с одной стороны, и в терминах циклов, фаз, итераций и временных отметок завершения определенных этапов в создании ПО (milestones), т.е. в терминах динамических аспектов процесса, с другой. []

1.3.3. eXtreme Programming

Экстремальное программирование [] – сравнительно молодая методология разработки программных систем, основанная на постепенном улучшении системы и разработки ее очень короткими итерациями. По своей сути экстремальное программирование (XP) - это одна из так называемых "гибких" методологий разработки ПО, представляющая собой небольшой набор конкретных правил, позволяющих максимально эффективно выполнять требования современной теории управления программными проектами.

XP ориентирована на:

- командную работу с тесными связями внутри команды и с заказчиком,
- разработку наиболее простых работающих решений,
- гибкое адаптивное планирование
- оперативную обратную связь (путем модульного и функционального тестирования).

Основными принципами XP является разработка небольшими итерациями на основании порции требований заказчика (т.н. пользовательских историй), написание функциональных тестов до написания программного кода, постоянное общение и постоянный рефакторинг кода.

Основными практиками XP являются

- Планирование процесса
- Частые релизы
- Метафора системы
- Простая архитектура
- Тестирование
- Рефакторинг
- Парное программирование
- Коллективное владение кодом
- Частая интеграция
- 40-часовая рабочая неделя
- Стандарты кодирования
- Тесное взаимодействие с заказчиком

1.3.4. Сравнение технологий MSF, RUP и XP

Основные особенности MSF, RUP и XP можно свести в небольшую таблицу (Таблица 2) []. По ней можно судить, что Rational Unified Process является хорошо сбалансированным решением для средних по размерам коллективов разработчиков, работающих с применением продуктов и технологий компании Rational. Сопровождение разработки системы и самой системы регламентируется самой методологией RUP, однако данная технология достаточно сильно ориентирована на внутрифирменные инструментальные средства.

Extreme Programming хорошо подходит для проектных групп малого размера и для небольших систем с часто изменяемыми требованиями. Основная проблема XP – сопровождаемость. В случае текучки кадров в коллективе разработчиков значительная часть проектной информации может быть утеряна из-за практически отсутствующей документации.

Таблица 2 Технологии MSF, RUP и XP

Технология	**Оптимальная команда**	**Соответствие стандартам**	**Допустимые технологии и инструменты**	**Удобство модификации и сопровождения**
Rational Unified Process	10 – 40 чел.	стандарты Rational	UML и продукты Rational	Удобно (RUP)
Microsoft Solutions Framework	3 – 20 чел.	адаптируема	любые	Удобно (MSF+MOF)

XP	2 – 10 чел.	стандарты отсутствуют	любые	Сложно (зависимость от конкретных участников коллектива)

Microsoft Solutions Framework является наиболее сбалансированной технологией, ориентированной на проектные группы малых и средних размеров. MSF не накладывает никаких ограничений на используемый инструментарий и содержит рекомендации весьма общего характера. Однако, эти рекомендации могут быть использованы для построения конкретного процесса, соответствующего потребностям коллектива разработчиков.

1.4. Ролевой состав коллектива разработчиков, взаимодействие между ролями в различных технологических процессах

Когда проектная команда включает более двух человек неизбежно встает вопрос о распределении ролей, прав и ответственности в команде. Конкретный набор ролей определяется многими факторами – количеством участников разработки и их личными предпочтениями, принятой методологией разработки, особенностями проекта и другими факторами. Практически в любом коллективе разработчиков можно выделить перечисленные ниже роли. Некоторые из них могут вовсе отсутствовать, при этом отдельные люди могут выполнять сразу несколько ролей, однако общий состав меняется мало.

Заказчик (заявитель). Эта роль принадлежит представителю организации, заказавшей разрабатываемую систему. Обычно заявитель ограничен в своем взаимодействии и общается только с менеджерами проекта и специалистом по сертификации или внедрению. Обычно заказчик имеет право изменять требования к продукту (только во взаимодействии с менеджерами), читать проектную и сертификационную документацию, затрагивающую нетехнические особенности разрабатываемой системы.

Менеджер проекта. Эта роль обеспечивает коммуникационный канал между заказчиком и проектной группой. Менеджер продукта управляет ожиданиями заказчика, разрабатывает и поддерживает бизнес-контекст проекта. Его работа не связана напрямую с продажей продукта, он сфокусирован на продукте, его задача - определить и обеспечить требования заказчика. Менеджер проекта имеет право изменять требования к продукту и финальную документацию на продукт.

Менеджер программы. Эта роль управляет коммуникациями и взаимоотношениями в проектной группе, является в некотором роде координатором, разрабатывает функциональные спецификации и управляет

ими, ведет график проекта и отчитывается по состоянию проекта, инициирует принятие критичных для хода проекта решений.

Менеджер программы имеет право изменять функциональные спецификации верхнего уровня, план-график проекта, распределение ресурсов по задачам. Часто на практике роль менеджера проекта и менеджера программы выполняет один человек.

Разработчик. Разработчик принимает технические решения, которые могут быть реализованы и использованы, создает продукт, удовлетворяющий спецификациям и ожиданиям заказчика, консультирует другие роли в ходе проекта. Он участвует в обзорах, реализует возможности продукта, участвует в создании функциональных спецификаций, отслеживает и исправляет ошибки за приемлемое время. В контексте конкретного проекта роль разработчика может подразумевать, например, инсталляцию программного обеспечения, настройку продукта или услуги. Разработчик имеет доступ ко всей проектной документации, включая документацию по тестированию, имеет право на изменение программного кода системы в рамках своих служебных обязанностей.

Специалист по тестированию. Специалист по тестированию определяет стратегию тестирования, тест-требования и тест-планы для каждой из фаз проекта, выполняет тестирование системы, собирает и анализирует отчеты о прохождении тестирования. Тест-требования должны покрывать системные требования, функциональные спецификации, требования к надежности и нагрузочной способности, пользовательские интерфейсы и собственно программный код. В реальности роль специалиста по тестированию часто разбивается на две – разработчика тестов и тестировщика. Тестировщик выполняет все работы по выполнению тестов и сбору информации, разработчик тестов – всю остальные работы.

Специалист по контролю качества. Эта роль принадлежит члену проектной группы, который осуществляет взаимодействие с разработчиком, менеджером программы и специалистами по безопасности и сертификации с целью отслеживания целостной картины качества продукта, его соответствия стандартам и спецификациям, предусмотренным проектной документацией. Следует различать специалиста по тестированию и специалиста по контролю качества. Последний не является членом технического персонала проекта, ответственным за детали и технику работы. Контроль качества подразумевает в первую очередь контроль самих процессов разработки и проверку их соответствия определенным в стандартах качества критериям.

Специалист по сертификации. При разработке систем, к надежности которых предъявляются повышенные требования, перед вводом системы в эксплуатацию требуется подтверждение со стороны уполномоченного органа

(обычно государственного) соответствия ее эксплуатационных характеристик заданным критериям. Такое соответствие определяется в ходе сертификации системы. Специалист по сертификации может либо быть представителем сертифицирующих органов, включенным в состав коллектива разработчиков, либо наоборот – представлять интересы разработчиков в сертифицирующем органе. Специалист по сертификации приводит документацию на программную систему в соответствие требованиям сертифицирующего органа, либо участвует в процессе создания документации с учетом этим требованиям. Также специалист по сертификации ответственен за все взаимодействие между коллективом разработчиков и сертифицирующим органом. Важной особенностью роли является независимость специалиста от проектной группы на всех этапах создания продукта. Взаимодействие специалиста с членами проектной группы ограничивается менеджерами по проекту и по программе.

Специалист по внедрению и сопровождению. Участвует в анализе особенностей площадки заказчика, на которой планируется проводить внедрение разрабатываемой системы, выполняет весь спектр работ по установке и настройке системы, проводит обучение пользователей.

Специалист по безопасности. Данный Специалист ответственен за весь спектр вопросов безопасности создаваемого продукта. Его работа начинается с участия в написании требований к продукту и заканчивается финальной стадией сертификации продукта.

Инструктор. Эта роль отвечает за снижение затрат на дальнейшее сопровождение продукта, обеспечение максимальной эффективности работы пользователя. Важно, что речь идет о производительности пользователя, а не системы. Для обеспечения оптимальной продуктивности инструктор собирает статистику по производительности пользователей и создает решения для повышения производительности, в том числе с использованием различных аудиовизуальных средств. Инструктор принимает участие во всех обсуждениях пользовательского интерфейса и архитектуры продукта.

Технический писатель. Лицо, осуществляющее эту роль, несет обязанности по подготовке документации к разработанному продукту, финального описания функциональных возможностей. Так же он участвует в написании сопроводительных документов (системы помощи, руководство пользователя).

1.5. Задачи и цели процесса верификации

Сначала рассмотрим цели верификации. Основная цель процесса – доказательство того, что результат разработки соотвествует предъявленным к нему требованиям. Обычно процесс верификации проводится сверху вниз, начиная от общих требований, заданных в техническом задании и/или

спецификации на всю информационную систему до детальных требований на программные модули и их взаимодействие. В состав задач процесса входит последовательная проверка того, что в программной системе:

- общие требования к информационной системе, предназначенные для программной реализации, корректно переработаны в спецификацию требований высокого уровня к комплексу программ, удовлетворяющих исходным системным требованиям;
- требования высокого уровня правильно переработаны в архитектуру ПО и в спецификации требований к функциональным компонентам низкого уровня, которые удовлетворяют требованиям высокого уровня;
- спецификации требований к функциональным компонентам ПО, расположенным между компонентами высокого и низкого уровня, удовлетворяют требованиям более высокого уровня;
- архитектура ПО и требования к компонентам низкого уровня корректно переработаны в удовлетворяющие им исходные тексты программных и информационных модулей;
- исходные тексты программ и соответствующий им исполняемый код не содержат ошибок.

Кроме того, верификации на соответствие спецификации требований на конкретный проект программного средства подлежат требования к технологическому обеспечению жизненного цикла ПО, а также требования к эксплуатационной и технологической документации.

Цели верификации ПО достигаются посредством последовательного выполнения комбинации из инспекций проектной документации и анализа их результатов, разработки тестовых планов тестирования и тест-требований, тестовых сценариев и процедур и последующего выполнения этих процедур. Тестовые сценарии предназначены для проверки внутренней непротиворечивости и полноты реализации требований. Выполнение тестовых процедур должно обеспечивать демонстрацию соответствия испытываемых программ исходным требованиям.

На выбор эффективных методов верификации и последовательность их применения в наибольшей степени влияют основные характеристики тестируемых объектов:

- класс комплекса программ, определяющийся глубиной связи его функционирования с реальным временем и случайными воздействиями из внешней среды, а также требования к качеству обработки информации и надежности функционирования;
- сложность или масштаб (объем, размеры) комплекса программ и его функциональных компонентов, являющихся конечными результатами разработки;

- преобладающие элементы в программах: осуществляющие вычисления сложных выражений и преобразования измеряемых величин или обрабатывающие логические и символьные данные для подготовки и отображения решений.

Определим некоторые понятия и определения, связанные с процессом тестирования, как составной части верификации. Майерс дает следующие определения основных терминов []:

Тестирование - процесс выполнения программы с целью обнаружения ошибки.

Тестовые данные – входы, которые используются для проверки системы.

Тестовая ситуация (test case) – входы для проверки системы и предполагаемые выходы в зависимости от входов, если система работает в соответствии с ее спецификацией требований.

Хорошая тестовая ситуация – та ситуация, которая обладает большой вероятностью обнаружения пока еще необнаруженной ошибки.

Удачный тест - тест, который обнаруживает пока еще необнаруженную ошибку.

Ошибка - действие программиста на этапе разработки, которое приводит к тому, что в программном обеспечении содержится внутренний дефект, который в процессе работы программы может привести к неправильному результату.

Отказ – непредсказуемое поведение системы, приводящее к неожидаемому результату, которое могло быть вызвано дефектами, содержащимся в ней.

Таким образом, в процессе тестирования программного обеспечения, как правило, проверяют следующее:

- Проверка того, что программное обеспечение соответствует требованиям на него,
- Проверка того, что в ситуациях, не отраженных в требованиях, программное обеспечение ведет себя адекватно, то есть не происходит отказ системы.
- Проверка программного обеспечения на предмет типичных ошибок, которые делают программисты.

1.6. Тестирование, верификация и валидация – различия в понятиях

Несмотря на кажущуюся схожесть, термины «тестирование», «верификация» и «валидация» означают разные уровни проверки корректности работы программной системы. Дабы избежать дальнейшей путаницы, четко определим эти понятия (Рис. 7).

Тестирование программного обеспечения – вид деятельности в процессе разработки, связанный с выполнением процедур, направленных на обнаружение (доказательство наличия) ошибок (несоответствий, неполноты, двусмысленностей и т.д.) в текущем определении разрабатываемой программной системы. Процесс тестирования относится в первую очередь к проверке корректности программной реализации системы, соответствия реализации требованиям, т.е. тестирование – это управляемое выполнение программы с целью обнаружения несоответствий ее поведения и требований.

Рис. 7 Тестирование, верификация и валидация

Верификация программного обеспечения – более общее понятие, чем тестирование. Целью верификации является достижение гарантии того, что верифицируемый объект (требования или программный код) соответствует требованиям, реализован без непредусмотренных функций и удовлетворяет проектным спецификациям и стандартам. Процесс верификации включает в себя инспекции, тестирование кода, анализ результатов тестирования, формирование и анализ отчетов о проблемах. Таким образом, принято считать, что процесс тестирования является составной частью процесса верификации, такое же допущение сделано и в данном учебном курсе.

Валидация программной системы – процесс, целью которого является доказательство того, что в результате разработки системы мы достигли тех целей, которые планировали достичь благодаря ее использованию. Иными словами, валидация – это проверка соответствия системы ожиданиям заказчика. Вопросы, связанные с валидацией выходят за рамки данного учебного курса и представляют собой отдельную интересную тему для изучения.

Если посмотреть на эти три процесса с точки зрения вопроса, на который они дают ответ, то тестирование отвечает на вопрос «Как это сделано?» или «Соответсвует ли поведение разработанной программы требованиям?», верификация – «Что сделано?» или «Соответствует ли разработанная система требованиям?», а валидация – «Сделано ли то, что нужно?» или «Соответствует ли разработанная система ожиданиям заказчика?».

1.7. Документация, создаваемая на различных этапах жизненного цикла

Синхронизация всех этапов разработки происходит при помощи документов, которые создаются на каждом из этапов. Документация при этом создается и на прямой отрезке жизненного цикла – при разработке программной системы, и на обратном – при ее верификации. Попробуем на примере V-образного жизненного цикла проследить за тем, какие типы документов создаются на каждом из отрезков, и какие взаимосвязи между ними существуют (Рис. 8).

Результатом этапа разработки требований к системе являются сформулированные требования к системе – документ, описывающие общие принципы работы системы, ее взаимодействие с «окружающей средой» - пользователями системы, а также, программными и аппаратными средствами, обеспечивающими ее работу. Обычно параллельно с требованиями к системе создается план верификации и определяется стратегия верификации. Эти документы определяют общий подход к тому, как будет выполняться тестирование, какие методики будут применяться, какие аспекты будущей системы должны быть подвергнуты тщательной проверке. Еще одна задача, решаемая при помощи определения стратегии верификации – определение места различных верификационных процессов и их связей с процессами разработки.

Верификационный процесс, работающий с системными требованиями – это процесс валидации требований, сопоставления их реальным ожиданиям заказчика. Забегая вперед, скажем, что процесс валидации отличается от приемо-сдаточных испытаний, выполняемых при передаче готовой системы заказчику, хотя может считаться частью таких испытаний. Валидация является средством доказать не только корректность реализации системы с точки зрения заказчика, но и корректность принципов, положенных в основу ее разработки.

Рис. 8 Процессы и документы при разработке программных систем

Требования к системе являются основой для процесса разработки функциональных требований и архитектуры проекта. В ходе этого процесса разрабатываются общие требования к программному обеспечению системы, к функциям которые она должна выполнять. Функциональные требования часто включают в себя определение моделей поведения системы в штатных и нештатных ситуациях, правила обработки данных и определения интерфейса с пользователем. Текст требования, как правило, включает в себя слова «должна, должен» и имеет структуру вида «В случае, если значение температуры на датчике ABC достигает 30 и выше градусов Цельсия, система должна прекращать выдачу звукового сигнала». Функциональные требования являются основой для разработки архитектуры системы – описания ее структуры в терминах подсистем и структурных единиц языка, на котором производится реализация – областей, классов, модулей, функций и т.п.

На базе функциональных требований пишутся тест-требования – документы, содержащие определение ключевых точек, которые должны быть проверены для того, чтобы убедиться в корректности реализации функциональных требований. Часто тест-требования начинаются словами «Проверить, что» и содержат ссылки на соответствующие им функциональные требования. Примером тест-требований для приведенного выше функционального требования могут служить «Проверить, что в случае падения температуры на датчике ABC ниже 30 градусов Цельсия система выдает предупреждающий

звуковой сигнал» и «Проверить, что в случае, когда значение температуры на датчике ABC выше 30 градусов Цельсия, система не выдает звуковой сигнал».

Одна из проблем, возникающих при написании тест-требований – принципиальная нетестируемость некоторых требований, например требование «Интерфейс пользователя должен быть интуитивно понятным» невозможно проверить без четкого определения того, что является интуитивно понятным интерфейсом. Такие неконкретные функциональные требования обычно впоследствии видоизменяют.

Архитектурные особенности системы также могут служить источником для создания тест-требований, учитывающих особенности программной реализации системы. Примером такого требования является, например, «Проверить, что значение температуры на датчике ABC не выходит за 255».

На основе функциональных требований и архитектуры пишется программный код системы, для его проверки на основе тест-требований готовится тест-план – описание последовательности тестовых примеров, выполняющих проверку соответствия реализации системы требованиям. Каждый тестовый пример содержит конкретное описание значений, подаваемых на вход системы, значений, которые ожидаются на выходе и описание сценария выполнения теста.

В зависимости от объекта тестирования тест-план может готовиться либо в виде программы на каком-либо языке программирования, либо в виде входного файла данных для инструментария, выполняющего тестируемую систему и передающего ей значения, указанные в тест-плане, либо в виде инструкций для пользователя системы, описывающей необходимые действия, которые нужно выполнить для проверки различных функций системы.

В результате выполнения всех тестовых примеров собирается статистика об успешности прохождения тестирования – процент тестовых примеров, для которых реальные выходные значения совпали с ожидаемыми, так называемых пройденных тестов. Не пройденные тесты являются исходными данными для анализа причин ошибок и последующего их исправления.

На этапе интеграции осуществляется сборка отдельных модулей системы в единое целое и выполнение тестовых примеров, проверяющих всю функциональность системы.

На последнем этапе осуществляется поставка готовой системы заказчику. Перед внедрением специалисты заказчика совместно с разработчиками проводят приемо-сдаточные испытания – выполняют проверку критичных для пользователя функций согласно заранее утвержденной программе

испытаний. При успешном прохождении испытаний система передается заказчику, в противном случае отправляется на доработку.

1.8. Типы процессов тестирования и верификации и их место в различных моделях жизненного цикла

1.8.1. Модульное тестирование

Модульному тестированию подвергаются небольшие модули (процедуры, классы и т.п.). При тестировании относительного небольшого модуля размером 100-1000 строк есть возможность проверить, если не все, то, по крайней мере, многие логические ветви в реализации, разные пути в графе зависимости данных, граничные значения параметров. В соответствии с этим строятся критерии тестового покрытия (покрыты все операторы, все логические ветви, все граничные точки и т.п.). []. Модульное тестирование обычно выполняется для каждого независимого программного модуля и является, пожалуй, наиболее распространенным видом тестирования, особенно для систем малых и средних размеров.

1.8.2. Интеграционное тестирование

Проверка корректности всех модулей, к сожалению, не гарантирует корректности функционирования системы модулей. В литературе иногда рассматривается «классическая» модель неправильной организации тестирования системы модулей, часто называемая методом «большого скачка». Суть метода состоит в том, чтобы сначала оттестировать каждый модуль в отдельности, потом объединить их в систему и протестировать систему целиком. Для крупных систем это нереально. При таком подходе будет потрачено очень много времени на локализацию ошибок, а качество тестирования останется невысоким. Альтернатива «большому скачку» — интеграционное тестирование, когда система строится поэтапно, группы модулей добавляются постепенно. []

1.8.3. Системное тестирование

Полностью реализованный программный продукт подвергается системному тестированию. На данном этапе тестировщика интересует не корректность реализации отдельных процедур и методов, а вся программа в целом, как ее видит конечный пользователь. Основой для тестов служат общие требования к программе, включая не только корректность реализации функций, но и производительность, время отклика, устойчивость к сбоям, атакам, ошибкам пользователя и т.д. Для системного и компонентного тестирования используются специфические виды критериев тестового покрытия (например, покрыты ли все типовые сценарии работы, все сценарии с нештатными ситуациями, попарные композиции сценариев и проч.). []

1.8.4. Нагрузочное тестирование

Нагрузочное тестирование позволяет не только получать прогнозируемые данные о производительности системы под нагрузкой, которая ориентирована на принятие архитектурных решений, но и предоставляет рабочую информацию службам технической поддержки, а также менеджерам проектов и конфигурационным менеджерам, которые отвечают за создания наиболее продуктивных конфигураций оборудования и ПО. Нагрузочное тестирование позволяет команде разработки, принимать более обоснованные решения, направленные на выработку оптимальных архитектурных композиций. Заказчик со своей стороны, получает возможность проводить приёмо-сдаточные испытания в условиях приближенных к реальным.

1.8.5. Формальные инспекции

Формальная инспекция является одним из способов верификации документов и программного кода, создаваемых в процессе разработки программного обеспечения. В ходе формальной инспекции группой специалистов осуществляется независимая проверка соответствия инспектируемых документов исходным документам. Независимость проверки обеспечивается тем, что она осуществляется инспекторами, не участвовавшими в разработке инспектируемого документа.

1.9. Верификация сертифицируемого программного обеспечения

Дадим несколько определений, определяющих общую структуру процесса сертификации программного обеспечения:

Сертификация ПО – процесс установления и официального признания того, что разработка ПО проводилась в соответствии с определенными требованиями. В процессе сертификации происходит взаимодействие *Заявителя, Сертифицирующего органа и Наблюдательного органа*

Заявитель - организация, подающая заявку в соответствующий *Сертифицирующий орган* на получения сертификата (соответствия, качества, годности и т.п.) изделия.

Сертифицирующий орган – организация, рассматривающая заявку *Заявителя* о проведении *Сертификации ПО* и либо самостоятельно, либо путем формирования специальной комиссии производящая набор процедур направленных на проведение процесса *Сертификации ПО Заявителя*.

Наблюдательный орган – комиссия специалистов, наблюдающих за процессами разработки *Заявителем* сертифицируемой информационной

системы и дающих заключение, о соответствии данного процесса определенным требованиям, которое передается на рассмотрение в *Сертифицирующий орган*.

Сертификация может быть направлена на получение сертификата соответствия, либо сертификата качества.

В первом случае результатом сертификации является признание соответствия процессов разработки определенным критериям, а функциональности системы определенным требованиям. Примером таких требований могут служить руководящие документы Федеральной службы по техническому и экспортному контролю в области безопасности программных систем [,].

Во втором случае результатом является признание соответствия процессов разработки определенным критериям, гарантирующим соответствующий уровень качества выпускаемой продукции и его пригодности для эксплуатации в определенных условиях. Примером таких стандартов может служить серия международных стандартов качества ISO 9000:2000 (ГОСТ Р ИСО 9000-2001) [] или авиационные стандарты DO-178B [], AS9100 [], AS9006 [].

Тестирование сертифицируемого программного обеспечения имеет две взаимодополняющие цели:

- Первая цель - продемонстрировать, что программное обеспечение удовлетворяет требованиям на него.
- Вторая цель - продемонстрировать с высоким уровнем доверительности, что ошибки, которые могут привести к неприемлемым отказным ситуациям, как они определены процессом, оценки отказобезопасности системы, выявлены в процессе тестирования.

Например, согласно требованиям стандарта DO-178B, для того, чтобы удовлетворить целям тестирования программного обеспечения, необходимо следующее:

- Тесты, в первую очередь, должны основываться на требованиях к программному обеспечению;
- Тесты должны разрабатываться для проверки правильности функционирования и создания условий для выявления потенциальных ошибок.
- Анализ полноты тестов, основанных на требованиях на программное обеспечение, должен определить, какие требования не протестированы.

- Анализ полноты тестов, основанных на структуре программного кода, должен определить, какие структуры не исполнялись при тестировании.

Также в этом стандарте говорится о тестировании, основанном на требованиях. Установлено, что эта стратегия наиболее эффективна при выявлении ошибок. Руководящие указания для выбора тестовых примеров, основанных на требованиях, включают следующее:

- Для достижения целей тестирования программного обеспечения должны быть проведены две категории тестов: тесты для нормальных ситуаций и тесты для ненормальных (не отраженных в требованиях, робастных) ситуаций.
- Должны быть разработаны специальные тестовые примеры для требований на программное обеспечение и источников ошибок, присущих процессу разработки программного обеспечения.

Целью тестов для нормальных ситуаций является демонстрация способности программного обеспечения давать отклик на нормальные входы и условия в соответствии с требованиями.

Целью тестов для ненормальных ситуаций является демонстрация способности программного обеспечения адекватно реагировать на ненормальные входы и условия, иными словами, это не должно вызывать отказ системы.

Категории отказных ситуаций для системы устанавливаются путем определения опасности отказной ситуации для самолета и тех, кто в нем находится. Любая ошибка в программном обеспечении может вызвать отказ, который внесет свой вклад в отказную ситуацию. Таким образом, уровень целостности программного обеспечения, необходимый для безопасной эксплуатации, связан с отказными ситуациями для системы.

Существует 5 уровней отказных ситуаций от несущественной до критически опасной. Согласно этим уровням вводится понятие уровня критичности программного обеспечения. От уровня критичности зависит состав документации, предоставляемой в сертифицирующий орган, а значит и глубина процессов разработки и верификации системы. Например, количество типов документов и объем работ по разработке системы, необходимых для сертификации по самому низкому уровню критичности DO-178B могут отличаться на один-два порядка от количества и объемов, необходимых для сертификации по самому высокому уровню. Конкретные требования определяет стандарт, по которому планируется вести сертификацию.

ТЕМА 2.Тестирование программного кода (лекции 2-5)

2.1. Задачи и цели тестирования программного кода

Тестирование программного кода – процесс выполнения программного кода, направленный на выявление существующих в нем дефектов. Под дефектом здесь понимается участок программного кода, выполнение которого при определенных условиях приводит к неожиданному поведению системы (т.е. поведению, не соответствующему требованиям). Неожиданное поведение системы может приводить к сбоям в ее работе и отказам, в этом случае говорят о существенных дефектах программного кода. Некоторые дефекты вызывают незначительные проблемы, не нарушающие процесс функционирования системы, но несколько затрудняющие работу с ней. В этом случае говорят о средних или малозначительных дефектах.

Задача тестирования при таком подходе – определение условий, при которых проявляются дефекты системы и протоколирование этих условий. В задачи тестирования обычно не входит выявление конкретных дефектных участков программного кода и никогда не входит исправление дефектов – это задача отладки, которая выполняется по результатам тестирования системы.

Цель применения процедуры тестирования программного кода – минимизация количества дефектов, в особенности существенных, в конечном продукте. Тестирование само по себе не может гарантировать полного отсутствия дефектов в программном коде системы. Однако, в сочетании с процессами верификации и валидации, направленными на устранение противоречивости и неполноты проектной документации (в частности – требований на систему), грамотно организованное тестирование дает гарантию того, что система удовлетворяет требованиям и ведет себя в соответствии с ними во всех предусмотренных ситуациях.

При разработке систем повышенной надежности, например, авиационных, гарантии надежности достигаются при помощи четкой организации процесса тестирования, определения его связи с остальными процессами жизненного цикла, введения количественных характеристик, позволяющих оценивать успешность тестирования. При этом, чем выше требования к надежности системы (ее уровень критичности), тем более жесткие требования предъявляются.

Таким образом, в первую очередь мы рассматриваем не конкретные результаты тестирования конкретной системы, а общую организацию процесса тестирования, используя подход «хорошо организованный процесс дает качественный результат». Такой подход является общим для многих международных и отраслевых стандартах качества, о которых более подробно будет рассказано в конце данного курса. Качество разрабатываемой системы при таком подходе является следствием

организованного процесса разработки и тестирования, а не самостоятельным неуправляемым результатом.

Поскольку современные программные системы имеют весьма значительные размеры, при тестировании их программного кода используется метод функциональной декомпозиции. Система разбивается на отдельные модули (классы, пространства имен и т.п.), имеющие определенную требованиями функциональность и интерфейсы. После этого по отдельности тестируется каждый модуль – выполняется модульное тестирование. Затем выполняется сборка отдельных модулей в более крупные конфигурации – выполняется интеграционное тестирование, и наконец, тестируется система в целом – выполняется системное тестирование.

С точки зрения программного кода, модульное, интеграционное и системное тестирование имеют много общего, поэтому в данной теме основное внимание будет уделено модульному тестированию, особенности интеграционного и системного тестирования будут рассмотрены позднее.

В ходе модульного тестирования каждый модуль тестируется как на соответствие требованиям, так и на отсутствие проблемных участков программного кода, могущих вызвать отказы и сбои в работе системы. Как правило, модули не работают вне системы – они принимают данные от других модулей, перерабатывают их и передают дальше. Для того, чтобы с одной стороны, изолировать модуль от системы и исключить влияние потенциальных ошибок системы, а с другой стороны – обеспечить модуль всеми необходимыми данными, используется тестовое окружение.

Задача тестового окружения – создать среду выполнения для модуля, эмулировать все внешние интерфейсы, к которым обращается модуль. Об особенностях организации тестового окружения пойдет речь в данной теме.

Типичная процедура тестирования состоит в подготовке и выполнении тестовых примеров (также называемых просто тестами). Каждый тестовый пример проверяет одну «ситуацию» в поведении модуля и состоит из списка значений, передаваемых на вход модуля, описания запуска и выполнения переработки данных – тестового сценария, и списка значений, которые ожидаются на выходе модуля в случае его корректного поведения. Тестовые сценарии составляются таким образом, чтобы исключить обращения к внутренним данным модуля, все взаимодействие должно происходить только через его внешние интерфейсы.

Выполнение тестового примера поддерживается тестовым окружением, которое включает в себя программную реализацию тестового сценария. Выполнение начинается с передачи модулю входных данных и запуска сценария. Реальные выходные данные, полученные от модуля в результате выполнения сценария сохраняются и сравниваются с ожидаемыми. В случае

их совпадения тест считается пройденным, в противном случае – не пройденным. Каждый не пройденный тест указывает либо на дефект в тестируемом модуле, либо в тестовом окружении, либо в описании теста.

Совокупность описаний тестовых примеров составляет тест-план – основной документ, определяющий процедуру тестирования программного модуля. Тест-план задает не только сами тестовые примеры, но и порядок их следования, который также может быть важен. Структура и особенности тест-планов будут рассмотрены в данной теме, проблемы, связанные с порядком следования тестовых примеров – в теме «Повторяемость тестирования».

При тестировании часто бывает необходимо учитывать не только требования к системе, но и структуру программного кода тестируемого модуля. В этом случае тесты составляются таким образом, чтобы детектировать типичные ошибки программистов, вызванные неверной интерпретацией требований. Применяются проверки граничных условий, проверки классов эквивалентности. Отсутствие в системе возможностей, не заданных требованиями, гарантируют различные оценки покрытия программного кода тестами, т.е. оценки того, какой процент тех или иных языковых конструкций выполнен в результате выполнения всех тестовых примеров. Обо всем этом пойдет речь в завершение данной темы.

2.2. Методы тестирования

2.2.1. Черный ящик

Основная идея в тестировании системы, как черного ящика состоит в том, что все материалы, которые доступны тестировщику – требования на систему, описывающие ее поведение и сама система, работать с которой он может только подавая на ее входы некоторые внешние воздействия и наблюдая на выходах некоторый результат. Все внутренние особенности реализации системы скрыты от тестировщика, таким образом, система и представляет собой «черный ящик», правильность поведения которого по отношению к требованиям и предстоит проверить.

С точки зрения программного кода черный ящик может представлять с собой набор классов (или модулей) с известными внешними интерфейсами, но недоступными исходными текстами.

Основная задача тестировщика для данного метода тестирования состоит в последовательной проверке соответствия поведения системы требованиям. Кроме того, тестировщик должен проверить работу системы в критических ситуациях – что происходит в случае подачи неверных входных значений. В идеальной ситуации все варианты критических ситуаций должны быть описаны в требованиях на систему и тестировщику остается только

придумывать конкретные проверки этих требований. Однако в реальности в результате тестирования обычно выявляется два типа проблем системы:

1. Несоответствие поведения системы требованиям
2. Неадекватное поведение системы в ситуациях, не предусмотренных требованиями.

Отчеты об обоих типах проблем документируются и передаются разработчикам. При этом проблемы первого типа обычно вызывают изменение программного кода, гораздо реже – изменение требований. Изменение требований в данном случае может потребоваться ввиду их противоречивости (несколько разных требований описывают разные модели поведения системы в одной и той же самой ситуации) или некорректности (требования не соответствуют действительности).

Проблемы второго типа однозначно требуют изменения требований ввиду их неполноты – в требованиях явно пропущена ситуация, приводящая к неадекватному поведению системы. При этом под неадекватным поведением может пониматься как полный крах системы, так и вообще любое поведение, не описанное в требованиях.

Тестирование черного ящика называют также тестированием по требованиям, т.к. это единственный источник информации для построения тест-плана.

2.2.2. Стеклянный (белый) ящик

При тестировании системы, как стеклянного ящика, тестировщик имеет доступ не только к требованиям на систему, ее входам и выходам, но и к ее внутренней структуре – видит ее программный код.

Доступность программного кода расширяет возможности тестировщика тем, что он может видеть соответствие требований участкам программного кода и видеть тем самым – на весь ли программный код существуют требования. Программный код, для которого отсутствуют требования называют кодом, непокрытым требованиями. Такой код является потенциальным источником неадекватного поведения системы. Кроме того, прозрачность системы позволяет углубить анализ ее участков, вызывающих проблемы – часто одна проблема нейтрализует другую и они никогда не возникают одновременно.

2.2.3. Тестирование моделей

Тестирование моделей находится несколько в стороне от классических методов верификации программного обеспечения. Прежде всего это связано с тем, что объект тестирования – не сама система, а ее модель,

спроектированная формальными средствами. Если оставить в стороне вопросы проверки корректности и применимости самой модели (считается, что ее корректность и соответствие исходной системе может быть доказана формальными средствами), то тестировщик получает в свое распоряжение достаточно мощный инструмент анализа общей целостности системы. Работая с моделью можно создать такие ситуации, которые невозможно создать в тестовой лаборатории для реальной системы. Работая с моделью программного кода системы можно анализировать его свойства и такие параметры системы, как оптимальность алгоритмов или ее устойчивость.

Однако тестирование моделей не получило широкого распространения именно ввиду трудностей, возникающих при разработке формального описания поведения системы. Одно из немногих исключений – системы связи, алгоритмический и математический аппарат которых достаточно хорошо проработан.

2.2.4. Анализ программного кода (инспекции)

Во многих ситуациях тестирование поведения системы в целом невозможно – отдельные участки программного кода могут никогда не выполняться, при этом они будут покрыты требованиями. Примером таких участков кода могут служить обработчики исключительных ситуаций. Если, например, два модуля передают друг другу числовые значения, и функции проверки корректности значений работают в обоих модулях, то функция проверки модуля-приемника никогда не будет активизирована, т.к. все ошибочные значения будут отсечены еще в передатчике.

В этом случае выполняется ручной анализ программного кода на корректность, называемый также просмотрами или инспекциями кода. Если в результате инспекции выявляются проблемные участки, то информация об этом передается разработчикам для исправления наравне с результатами обычных тестов.

2.3. Тестовое окружение

Основной объем тестирования практически любой сложной системы обычно выполняется в автоматическом режиме. Кроме того, тестируемая система обычно разбивается на отдельные модули, каждый из которых тестируется вначале отдельно от других, затем в комплексе.

Это означает, что для выполнения тестирования необходимо создать некоторую среду, которая обеспечит запуск и выполнение тестируемого модуля, передаст ему входные данные, соберет реальные выходные данные, полученные в результате работы системы на заданных входных данных. После этого среда должна сравнить реальные выходные данные с

ожидаемыми и на основании данного сравнения сделать вывод о соответствии поведения модуля заданному (Рис. 9).

Рис. 9 Обобщенная схема среды тестирования

Тестовое окружение также может использоваться для отчуждения отдельных модулей системы от всей системы. Разделение модулей системы на ранних этапах тестирования позволяет более точно локализовать проблемы, возникающие в их программном коде. Для поддержки работы модуля в отрыве от системы тестовое окружение должно моделировать поведение всех модулей, к функциям или данным которых обращается тестируемый модуль.

Поскольку тестовое окружение само является программой (причем, часто не на том языке программирования, на котором написана система), оно само должно быть протестировано. Целью тестирования тестового окружения является доказательство того, что тестовое окружение никаким образом не искажает выполнение тестируемого модуля и адекватно моделирует поведение системы.

2.3.1. Драйверы и заглушки

Тестовое окружение для программного кода на структурных языках программирования состоит из двух компонентов – драйвера, который обеспечивает запуск и выполнение тестируемого модуля и заглушек, которые моделируют функции, вызываемые из данного модуля. Разработка тестового драйвера представляет собой отдельную задачу тестирования, сам драйвер должен быть протестирован, дабы исключить неверное тестирование. Драйвер и заглушки могут иметь различные уровни сложности, требуемый уровень сложности выбирается в зависимости от сложности тестируемого модуля и уровня тестирования. Так, драйвер может выполнять следующие функции:

1. Вызов тестируемого модуля
2. 1 + передача в тестируемый модуль входных значений и прием результатов
3. 2 + вывод выходных значений
4. 3 + протоколирование процесса тестирования и ключевых точек программы

Заглушки могут выполнять следующие функции:

1. Не производить никаких действий (такие заглушки нужны для корректной сборки тестируемого модуля и
2. Выводить сообщения о том, что заглушка была вызвана
3. 1 + выводить сообщения со значениями параметров, переданных в функцию
4. 2 + возвращать значение, заранее заданное во входных параметрах теста
5. 3 + выводить значение, заранее заданное во входных параметрах теста
6. 3 + принимать от тестируемого ПО значения и передавать их в драйвер [].

Для тестирования программного кода, написанного на процедурном языке программирования используются драйверы, представляющие собой программу с точкой входа (например, функцией main()), функциями запуска тестируемого модуля и функциями сбора результатов. Обычно драйвер имеет как минимум одну функцию – точку входа, которой передается управление при его вызове.

Функции-заглушки могут помещаться в тот же файл исходного кода, что и основной текст драйвера. Имена и параметры заглушек должны совпадать с именами и параметрами «заглушаемых» функций реальной системы. Это требование важно не столько с точки зрения корректной сборки системы (при сборке тестового драйвера и тестируемого ПО может использоваться приведение типов), сколько для того, чтобы максимально точно моделировать поведение реальной системы по передаче данных. Так, например, если в реальной системе присутствует функция вычисления квадратного корня

```
double sqrt(double value);
```

то с точки зрения сборки системы вместо типа double может использоваться и float, но снижение точности может вызвать непредсказуемые результаты в тестируемом модуле.

В качестве примера драйвера и заглушек рассмотрим реализацию стека на языке С, причем значения, помещаемые в стек, хранятся не в оперативной памяти, а помещаются в ППЗУ при помощи отдельного модуля, содержащего две функции – записи данных в ППЗУ по адресу и чтения данных по адресу.

Форматэтихфункцийследующий:

```
void NV_Read(char *destination, long length, long offset);

void NV_Write(char *source, long length, long offset);
```

Здесь destination – адрес области памяти, в которую записывается значение, считанное из ППЗУ, source – адрес области памяти, из которой записывается значение в ППЗУ, length – длина записываемой области памяти, offset – смещение относительно начального адреса ППЗУ.

Реализация стека с использованием этих функций выглядит следующим образом:

```
long currentOffset;

void initStack()

{

currentOffset=0;

}

void push(int value)

{

NV_Write((int*)&value,sizeof(int),currentOffset);

    currentOffset+=sizeof(int);

}

int pop()

{

int value;

if (currentOffset>0)
```

```
    {
NV_Read((int*)&value,sizeof(int),currentOffset;
currentOffset-=sizeof(int);
    }
}
```

При выполнении этого кода на реальной системе происходит запись в ППЗУ, однако, если мы хотим протестировать только реализацию стека, изолировав ее от реализации модуля работы с ППЗУ, необходимо использовать заглушки вместо реальных функций. Для имитации работы ППЗУ можно выделить достаточно большой участок оперативной памяти, в которой и будет производиться запись данных, получаемых заглушкой.

Заглушки для функций могут выглядеть следующим образом:

```
char nvrom[1024];
void NV_Read(char *destination, long length, long offset)
{
printf("NV_Read called\n");
    memcpy(destination, nvrom+offset, length);
}
void NV_Write(char *source, long length, long offset);
{
printf("NV_Write called\n");
    memcpy(nvrom+offset, source, length);
}
```

Каждая из заглушек выводит трассировочное сообщение и перемещает переданное значение в память, эмулирующую ППЗУ (функция NV_Write) или возвращает по ссылке значение, хранящееся в памяти, эмулирующей ППЗУ (функция NV_Read).

Схема взаимодействия тестируемого ПО (функций работы со стеком) с реальным окружением (основной частью системы и модулем работы с ППЗУ) и тестовым окружением (драйвером и заглушками функций работы с ППЗУ) показана на Рис. 10 и Рис. 11.

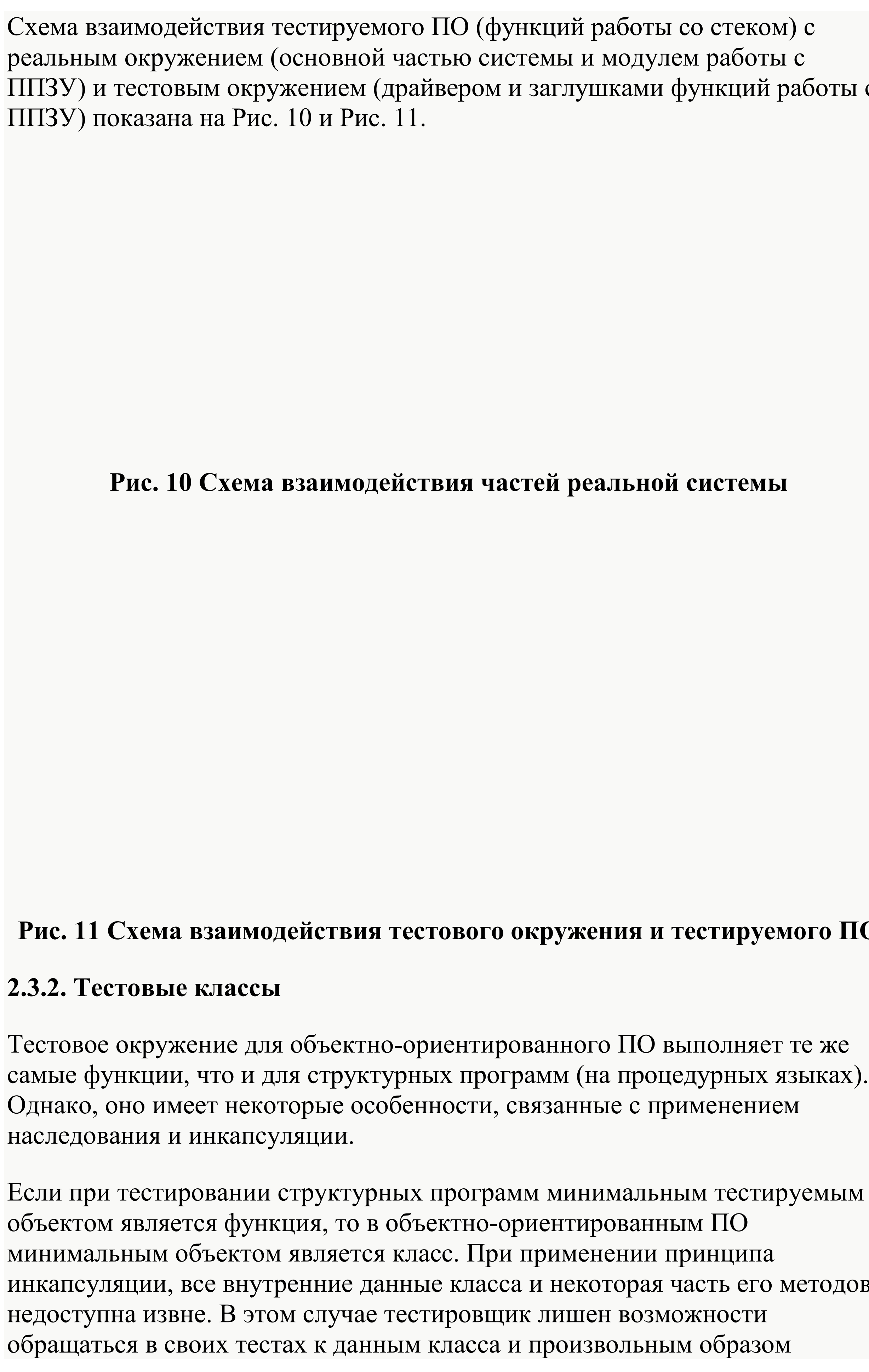


Рис. 10 Схема взаимодействия частей реальной системы

Рис. 11 Схема взаимодействия тестового окружения и тестируемого ПО

2.3.2. Тестовые классы

Тестовое окружение для объектно-ориентированного ПО выполняет те же самые функции, что и для структурных программ (на процедурных языках). Однако, оно имеет некоторые особенности, связанные с применением наследования и инкапсуляции.

Если при тестировании структурных программ минимальным тестируемым объектом является функция, то в объектно-ориентированным ПО минимальным объектом является класс. При применении принципа инкапсуляции, все внутренние данные класса и некоторая часть его методов недоступна извне. В этом случае тестировщик лишен возможности обращаться в своих тестах к данным класса и произвольным образом

вызывать методы – единственное, что ему доступно – вызывать методы внешнего интерфейса класса.

Существует несколько подходов к тестированию классов, каждый из них накладывает свои ограничения на структуру драйвера и заглушек:

1. Драйвер создает один или больше объектов тестируемого класса, все обращения к объектам происходят только с использованием их внешнего интерфейса. Текст драйвера в этом случае представляет собой т.н. тестирующий класс, который содержит по одному методу для каждого тестового примера. Процесс тестирования заключается в последовательном вызове этих методов. Вместо заглушек в состав тестового окружения входит программный код реальной системы, соответственно отсутствует изоляция тестируемого класса. Однако, именно такой подход к тестированию принят сейчас в большинстве методологий и сред разработки. Его классическое название – unit testing (тестирование модулей), более подробно он будет рассматриваться в теме 6.
2. Аналогично предыдущему подходу, но для всех классов, которые использует тестируемый класс, создаются заглушки
3. Программный код тестируемого класса модифицируется таким образом, чтобы открыть доступ ко всем его свойствам и методам. Строение тестового окружения в этом случае полностью аналогично окружению для тестирования структурных программ.
4. Используются специальные средства доступа к закрытым данным и методам класса на уровне объектного или исполняемого кода – скрипты отладчика или accessors в Visual Studio.

Основное достоинство первых двух методов – при их использовании класс работает точно таким же образом, как в реальной системе. Однако в этом случае нельзя гарантировать того, что в процессе тестирования будет выполнен весь программный код класса и не останется непротестированных методов.

Основной недостаток 3-го метода – после изменения исходных текстов тестируемого модуля нельзя дать гарантии того, что класс будет вести себя таким же образом, как и исходный. В частности это связано с тем, что изменение защиты данных класса влияет на наследование данных и методов другими классами.

Тестирование наследования – отдельная сложная задача в объектно-ориентированных системах. После того, как протестирован базовый класс, необходимо тестировать классы-потомки. Однако, для базового класса нельзя создавать заглушки, т.к. в этом случае можно пропустить возможные проблемы полиморфизма. Если класс-потомок использует методы базового

класса для обработки собственных данных, необходимо убедиться в том, что эти методы работают.

Таким образом, иерархия классов может тестироваться сверху вниз, начиная от базового класса. Тестовое окружение при этом может меняться для каждой тестируемой конфигурации классов.

2.3.3. Генераторы сигналов (событийно-управляемый код)

Значительная часть сложных программ в настоящее время использует ту или иную форму межпроцессного взаимодействия. Это обусловлено естественной эволюцией подходов к проектированию программных систем, которая последовательно прошла следующие этапы []:

1. Монолитные программы, содержащие в своем коде все необходимые для своей работы инструкции. Обмен данными внутри таких программ производится при помощи передачи параметров функций и использования глобальных переменных. При запуске таких программ образуется один процесс, который выполняет всю необходимую работу.
2. Модульные программы, которые состоят из отдельных программных модулей с четко определенными интерфейсами вызовов. Объединение модулей в программу может происходить либо на этапе сборки исполняемого файла (статическая сборка или static linking), либо на этапе выполнения программы (динамическая сборка или dynamic linking). Преимущество модульных программ заключается в достижении некоторого уровня универсальности – один модуль может быть заменен другим. Однако, модульная программа все равно представляет собой один процесс, а данные, необходимые для решения задачи, передаются внутри процесса как параметры функций.
3. Программы, использующие межпроцессное взаимодействие. Такие программы образуют программный комплекс, предназначенный для решения общей задачи. Каждая запущенная программа образует один или более процессов. Каждый из процессов использует для решения задачи либо свои собственные данные и обменивается с другими процессами только результатом своей работы, либо работает с общей областью данных, разделяемых между разными процессами. Для решения особо сложных задач процессы могут быть запущены на разных физических компьютерах и взаимодействовать через сеть. Преимущество использования межпроцессного взаимодействия заключается в еще большей универсальности – взаимодействующие процессы могут быть заменены независимо друг от друга при сохранении интерфейса взаимодействия. Другое преимущество состоит в том, что вычислительная нагрузка распределяется между процессами. Это позволяет операционной системе управлять приоритетами

выполнения отдельных частей программного комплекса, выделяя большее или меньшее количество ресурсов ресурсоемким процессам.

При выполнении многих процессов, решающих общую задачу, используются несколько типичных механизмов взаимодействия между ними, направленных на решение следующих задач:

- передача данных от одного процесса к другому;
- совместное использование одних и тех же данных несколькими процессами;
- извещения об изменении состояния процессов.

Во всех этих случаях типичная структура каждого процесса представляет собой конечный автомат с набором состояний и переходов между ними. Находясь в определенном состоянии, процесс выполняет обработку данных, при переходе между состояниями – пересылает данные другим процессам или принимает данные от них [].

Для моделирования конечных автоматов используются stateflow [] или SDL-диаграммы [], акцент в которых делается соответственно на условиях перехода между состояниями и пересылаемыми данными.

Так, на Рис. 12 показана схема процесса приема/передачи данных. Закругленными прямоугольниками указаны состояния процесса, тонкими стрелками – переходы между состояниями, большими стрелками – пересылаемые данные. Находясь в состоянии «Старт» процесс посылает во внешний мир (или процессу, с которым он обменивается данными) сообщение о своей готовности к началу сеанса передачи данных. После получения от второго процесса подтверждения о готовности начинается сеанс обмена данными. В случае поступления сообщения о конце данных происходит завершение сеанса и переход в стартовое состояние. В случае поступления неверных данных (например, неправильного формата или с неверной контрольной суммой), процесс переходит в состояние «Ошибка», выйти из которого возможно только завершением и перезапуском процесса.

Рис. 12 Пример конечного автомата процесса приема-передачи данных

Тестовое окружение для такого процесса также должно иметь структуру конечного автомата и пересылать данные в том же формате, что и тестируемый процесс. Целью тестирования в данном случае будет показать, что процесс обрабатывает принимаемые данные в соответствии с требованиями, форматы передаваемых данных корректны, а также, что процесс во время своей работы действительно проходит все состояния конечного автомата, моделирующего его поведение.

2.4. Тестовые примеры

Тестовое окружение, рассмотренное в предыдущем разделе обеспечивает процесс тестирования необходимой инфраструктурой и поддерживает ее. Непосредственно для тестирования кроме тестового окружения необходимо определить проверочные задачи, которые будет выполнять система или ее часть. Такие проверочные задачи называют тестовыми примерами.

Как уже было сказано выше, каждый тестовый пример состоит из входных значений для системы, описания сценария работы примера и ожидаемых выходных значений. Целью выполнения любого тестового примера является либо продемонстрировать наличие в системе дефекта, либо доказать его отсутствие.

2.4.1. Тест-требования как основной источник информации для создания тестовых примеров

Основным источником информации для создания тестовых примеров является различного рода документация на систему, например, функциональные требования и требования к интерфейсу.

Функциональные требования описывают поведение системы, как «черного ящика», т.е. исключительно с позиций того, что должна делать система в различных ситуациях. Иными словами, функциональные требования определяют реакцию системы на различные входные воздействия.

Например, функциональные требования на программный модуль, рассчитывающий и проверяющий контрольную сумму для записи могут выглядеть следующим образом:

Функциональные требования на модуль расчета и проверки контрольной суммы

Внешний интерфейс модуля

1. **Структура record_type**

```
struct record_type

{

bool A;

int B[20];

signed char C[5];

unsigned int CRC;

double D[1];

}
```

2. Переменная Empty

```
bool Empty;
```

3. Функция подсчета контрольной суммы записи **Set_CRC**

```
void Set_CRC(record_type record);
```

Вход:

Запись **record,** с неопределенным значением поля **CRC**.

Выход:

Запись **record,** с вычисленным по заданным правилам значение поля **CRC.**

Переменная **Empty.**

4. Функция проверки контрольной суммы записи **Check_CRC**

bool Check_CRC(record_type record);

Вход:

Запись **Rec_Mess** с определенным значением поля **CRC**.

Выход:

Возвращаемое значение **true** или **false**. Переменная **Empty.**

Функциональные требования

1. **Инициализация модуля**

При инициализации модуля переменная **Empty** должна быть установлена в значение **TRUE**.

2. **Подсчет контрольной суммы записи**
    1. **Расчет контрольной суммы**

Процедура **Set_CRC** должна производить подсчет контрольной суммы записи **Rec_Mess** по алгоритму CRC32. При подсчете контрольной суммы значение поля **CRC** не должно участвовать в суммировании. На основании произведенных расчетов должно быть вычислено и определено значение поля **CRC** таким образом, чтобы при подсчете контрольной суммы вместе с установленным значением этого поля контрольная сумма равнялась нулю.

    2. **Установка значения переменной Empty**

Если все байты полей записи (кроме возможно CRC поля) имеют нулевое значение (код 00000000B), то значение переменной Empty должно быть установлено в TRUE.

Ели хотя бы один байт записи (исключая байты поля CRC) не нулевой, то значение переменной Empty должно быть установлено в FALSE.

3. Проверка контрольной суммы записи

1. Проверка контрольной суммы

Процедура должна вычислять по заданному алгоритму CRC32 контрольную сумму записи Rec_Mess. Возвращаемое процедурой значение должно быть равно TRUE, если подсчитанное значение равно нулю. При не нулевом значении, подсчитанной контрольной суммы, должно возвращаться значение FALSE.

2. Установка значения переменной Empty

Если все байты полей записи, включая значение CRC поля, имеют нулевое значение (код 00000000B), то значение переменной Empty должно быть установлено в TRUE.

Ели хотя бы один байт записи не нулевой, то значение переменной Empty должно быть установлено в FALSE.

Начальный этап работы тестировщика заключается в формировании тест-требований, соответствующих функциональным требованиям. Основная цель тест-требований – определить, какая функциональность системы должна быть протестирована. В самом простом случае одному функциональному требованию соответствует одно тест-требование. Однако чаще всего тест-требования детализируют формулировки функциональных требований.

Тест-требования определяют, что должно быть протестировано, но не определяют, как это должно быть сделано. Например, для перечисленных выше функциональных требований можно сформулировать следующие тест-требования:

Тест-требования

1. Проверка инициализация модуля

Проверить, что начальное значение переменной **Empty** установлено **TRUE**.

2. Проверка подсчета контрольной суммы

1. Проверить, что в процедуре **Set_CRC** вычисление контрольной суммы производится по правилам алгоритма CRC32, как определено в секции 2a функциональных требований.
2. Проверить, что вычисленное значение контрольной суммы не зависит от начального значения поля **CRC**.
3. Проверить, что вычисленное значение контрольной суммы не зависит от значений байт выравнивания полей записи.
4. Проверить, что значение переменной **Empty** устанавливается при каждом вызове функции **Set_CRC** в зависимости от значений полей записи, как определено в секции 2b функциональных требований.

3. **Проверка процедуры Check_CRC**

1. Проверить, что при обращении к процедуре **Check_CRC** вычисление контрольной суммы производится по правилам алгоритма CRC32, как определено в секции 3a функциональных требований.
2. Проверить, что возвращаемое значение равно TRUE если конитрольная сумма проверяемой записи правильная и FALSE в противном случае.
3. Проверить, что проверка правильности значения контрольной суммы не зависит от значений байт выравнивания полей записи.
4. Проверить, что значение переменной **Empty** устанавливается при каждом вызове функции **Check_CRC** в зависимости от значений полей записи, как определено в секции 3b функциональных требований.

Особенности реализации тестового окружение и конкретные значения, подаваемые на вход системы и ожидаемые на ее выходе определяются тестовыми примерами. Одному тест-требованию соответствует как минимум один тестовый пример.

2.4.2. Типы тестовых примеров

Рассмотрим различные классы тестовых примеров, направленные на выявление различных дефектов в работе программной системы.

- **Допустимые данные**

Чаще всего дефекты в программных системах проявляются при обработке нестандартных данных, не предусмотренных требованиями – при вводе неверных символов, пустых строк, слишком большой скорости ввода

информации. Однако, перед поиском таких дефектов необходимо удостовериться в том, что программа корректно обрабатывает верные данные, предусмотренные спецификацией, т.е. проверить работу основных алгоритмов. Так, для функции вычисления контрольной суммы допустимыми входными данными будет произвольная запись, содержащая данные во всех полях, кроме поля контрольной суммы CRC.

```
record_type test_value1;

int i;

test_value1.A = false;

for (i=0;i<20;i++)

    test_value1.B[i] = i;

for (i=0;i<5;i++)

    test_value1.C[i] = i+5;

test_value1.D[0] = i+8;

test_value1.CRC = 0;

Set_CRC(test_value1);

printf("%d\n", test_value1.CRC);
```

Сценарием будет вызов функции Set_CRC, а ожидаемым выходным значением – корректное значение поля CRC, рассчитанное по алгоритму CRC32.

Обычно для проверки допустимых данных достаточно одного тестового примера. Но функциональные требования могут определять различные группы допустимых данных, которые могут объединяться в классы эквивалентности. В этом случае необходимо определять как минимум один тестовый пример для одного класса эквивалентности. Более подробно речь о классах эквивалентности пойдет далее.

- **Граничные данные**

Отдельный вид допустимых данных, передача которых в систему может вскрыть дефект – граничные данные, т.е. например, числа, значения которых являются предельными для их типа, строки предельной или нулевой длины и

т.п. Обычно при помощи тестирования граничных условий выявляются проблемы с арифметическим сравнением чисел или с итераторами циклов.

Для тестирования функции Set_CRC на граничных условиях можно определить два тестовых примера с минимальными и максимальными значениями полей в записи.

```
record_type test_value2;

record_type test_value3;

int i;

test_value2.A = false;

for (i=0;i<20;i++)

    test_value2.B[i] = 0;

for (i=0;i<5;i++)

    test_value2.C[i] = 0;

test_value2.D[0] = 0;

test_value2.CRC = 0;

Set_CRC(test_value2);

printf("%d\n", test_value2.CRC);

test_value3.A = true;

for (i=0;i<20;i++)

    test_value3.B[i] = pow(2,sizeof(test_value3.B[i])*8)-1;

for (i=0;i<5;i++)

    test_value3.C[i] = pow(2,sizeof(test_value3.C[i])*8)-1;

test_value3.D[0] = pow(2,sizeof(test_value3.D[0])*8)-1;

test_value3.CRC = pow(2,sizeof(test_value3.CRC)*8)-1;

Set_CRC(test_value3);

printf("%d\n", test_value3.CRC);
```

- **Отсутствие данных**

Дефекты могут проявиться и в случае, если системе не передается никаких данных или передаются данные нулевого размера. Для тестирования функции Set_CRC при отсутствии данных можно вызвать ее, передав в качестве параметра неинициализированную структуру. Однако такой тест не является точным примером отсутствия данных, скорее это пример случайных данных (возможно – неверных).

```
record_type test_value4;

Set_CRC(test_value4);

printf("%d\n", test_value4.CRC);
```

- **Повторный ввод данных**

В случае повторной передачи на вход системы тех же самых данных могут получаться различия в выходных данных, не предусмотренные в требованиях. Как правило, дефекты такого типа проявляются в результате того, что система не устанавливает внутренние переменные в исходное состояние или в результате ошибок округления.

```
record_type test_value5;

int i;

test_value5.A = false;

for (i=0;i<20;i++)

    test_value5.B[i] = i;

for (i=0;i<5;i++)

    test_value5.C[i] = i+5;

test_value5.D[0] = i+8;

test_value5.CRC = 0;

Set_CRC(test_value5);

printf("%d\n", test_value5.CRC);

Set_CRC(test_value5);
```

```
printf("%d\n", test_value5.CRC);
```

- **Неверные данные**

При проверке поведения системы необходимо не забывать проверять ее поведение при передаче ей данных, не предусмотренных требованиями – слишком длинных или слишком коротких строк, неверных символов, чисел за пределами вычислимого диапазона и т.п. Неверные данные, как и допустимые, также можно разделять на различные классы эквивалентности. Примером неверных данных для функции Set_CRC может служить запись с другой структурой, переданная в функцию через приведение типов. Если расчет контрольной суммы использует имена полей записи, то контрольная сумма может оказаться вычисленной неверно или может произойти перезапись областей памяти, не предназначенных для хранения данных.

```
struct record_type2

{

int F;

int G[45];

int H[8];

unsigned int CRC;

int K[2];

}

record_type2 test_value6;

Set_CRC((record_type)test_value6);

printf("%d\n", test_value6.CRC);
```

- **Реинициализация системы**

Механизмы повторной инициализации системы во время ее работы также могут содержать дефекты. В первую очередь эти дефекты могут проявляться в том, что не все внутренние данные системы после реинициализации придут в начальное состояние. В результате может произойти сбой в работе системы.

В качестве примера реинициализации модуля вычисления CRC может служить принудительное обнуление переменной empty.

```
record_type test_value7;

int i;

test_value7.A = false;

for (i=0;i<20;i++)

    test_value7.B[i] = i;

for (i=0;i<5;i++)

    test_value7.C[i] = i+5;

test_value7.D[0] = i+8;

test_value7.CRC = 0;

Set_CRC(test_value7);

printf("%d\n", test_value1.CRC);

empty=true;

Set_CRC(test_value7);

printf("%d\n", test_value1.CRC);
```

- **Устойчивость системы**

Под устойчивостью системы можно понимать ее способность выдерживать нештатную нагрузку, явно не предусмотренную требованиями. Например, сохранит ли система работоспособность после 10 тысяч вызовов. Для функции Set_CRC можно реализовать следующий тестовый пример:

```
record_type test_value8;

int i;

test_value8.A = false;

for (i=0;i<20;i++)

    test_value8.B[i] = i;
```

```
for (i=0;i<5;i++)

    test_value8.C[i] = i+5;

test_value8.D[0] = i+8;

test_value8.CRC = 0;

for (i=0;i<10000;i++)

    Set_CRC(test_value8);

printf("%d\n", test_value1.CRC);
```

Аналогичный анализ может быть сделан путем просмотра текста программы (если он доступен при тестировании) на основании отсутствия «истории» (хранимых данных) в реализации программы, т.е. данных, значение которых может меняться в зависимости от количества запусков программы. Таким образом, в ряде случаем тестирование может быть заменено анализом программного кода (более подробно – см. тему 5).

- **Нештатные состояния среды выполнения**

Нештатные состояния среды выполнения (например, исчерпание памяти, дискового пространства или длительная нехватка процессорного времени) могут затруднять работу системы, либо делать ее невозможной. Основная задача системы в такой ситуации – корректно завершить или приостановить свою работу.

Примером тестового примера, создающего нештатное состояние среды для функции Set_CRC может служить выделение всей свободной памяти перед вызовом функции. Если Set_CRC использует динамическую память, то в ней должны присутствовать проверки на возможность выделить память, в противном случае выполнение функции вызовет ее аварийное завершение:

```
record_type test_value9;

int i;

int *heap;

heap = malloc(_MAXMEM);

test_value9.A = false;

for (i=0;i<20;i++)
```

```
    test_value9.B[i] = i;

for (i=0;i<5;i++)

    test_value9.C[i] = i+5;

test_value9.D[0] = i+8;

test_value9.CRC = 0;

Set_CRC(test_value9);

free(heap);

printf("%d\n", test_value9.CRC);
```

2.4.2.1. Граничные условия

В тестовых примерах, прямо соответствующих тест-требованиям обычно используются входные значения, находящиеся заведомо внутри допустимого диапазона. Один из способов проверки устойчивости системы на значениях, близких к предельным – создавать для каждого входа как минимум три тестовых примера:

- Значение внутри диапазона
- Минимальное значение
- Максимальное значение

Для еще большей уверенности в работоспособности системы используют пять тестовых примеров:

- Значение внутри диапазона
- Минимальное значение
- Минимальное значение + 1
- Максимальное значение
- Максимальное значение – 1

Такой способ проверки называется проверкой на граничных значениях. Такая проверка позволяет выявлять проблемы, связанные с выходом за границы диапазона. Например, есливфункцию

```
char sum(char a, char b)

{

return a+b;
```

```
}
```

вычисляющую сумму чисел a и b будут переданы значения 255 и 255, то в случае отсутствия специальной обработки ситуации переполнения сумма будет вычислена неверно.

Другая область, при тестировании которой полезно пользоваться проверкой на граничных значениях – индексы массивов. Например, функция

```
void abs_array(char array[], char size)

{

for (int i=1;i<=size;i++)

{

array[i] = abs(array[i]);

    }

    return;

}
```

заменяющая значение на значение по модулю у каждого элемента переданного ей массива содержит ошибку в цикле for, которая может быть легко обнаружена при передаче в функцию массива единичного размера.

2.4.3. Проверка робастности (выхода за границы диапазона)

Робастность системы – это степень ее чувствительности к факторам, не учтенным на этапах ее проектирования, например, к неточности основного алгоритма, приводящего к ошибкам округления при вычислениях, сбоям во внешней среде или к данным, значения которых находятся вне допустимого диапазона. Чаще всего под робастностью программных систем понимают именно устойчивость к некорректным данным. Система должна быть способна корректно обрабатывать такие данные путем выдачи соответствующих сообщений об ошибках, сбои и отказы системы на таких данных недопустимы.

Для тестирования робастности к тестовым примерам, рассмотренным в предыдущем разделе добавляются еще два тестовых примера:

- Минимальное значение - 1
- Максимальное значение + 1,

проверяющие поведение системы за границей допустимого диапазона, а также в случае тестирования операций сравнения, дополнительно дающие гарантию того, что в них не допущена опечатка.

Таким образом, если изобразить допустимый интервал, как на Рис. 13, то можно видеть, что для тестирования интервальных значений достаточно 7 тестовых примеров – пяти допустимых и двух на робастность.

Рис. 13 Рекомендуемые проверочные значения

В литературе часто встречается утверждение, что значение внутри интервала является избыточным и его тестирование не требуется. Однако, проверка внутреннего значения является полезной как минимум с психологической точки зрения, а также в случае если интервал ограничен сложными граничными условиями. Также рекомендуется отдельно проверять значение 0 (даже если оно находится внутри интервала), т.к. зачастую это значение обрабатывается некорректно (например, в случае деления на 0).

2.4.4. Классы эквивалентности

При разработке тестовых примеров может возникнуть такая ситуация, в которой различные входные значения приводят к одним и тем же реакциям системы. Если при этом такие входные значения имеют что-то общее, то возможно объединение таких значений в классы эквивалентности, т.е. выполнение эквивалентного разбиения множества допустимых входных значений.

Разбиение на классы эквивалентности - в первую очередь, способ уменьшения необходимого числа тестовых примеров. Обычно, если в тест-требованиях специально не оговорено иное, при тестировании достаточно выполнить только один тестовый пример для каждого класса

эквивалентности. Разбиение на классы эквивалентности особенно полезно, когда на вход системы может быть подано большое количество различных значений; тестирование каждого возможного значения привело бы к слишком большому объему тестирования.

Рассмотренные выше граничные условия могут служить примером классов эквивалентности:

1. Значение из середины интервала.
2. Граничные значения.
3. Недопустимые значения за границами интервала.

Таким образом, тестирование граничных условий и робастности является частным случаем тестирования с использованием классов эквивалентности – вместо того, чтобы тестировать все недопустимые значения, выбираются только соседние с граничными.

При определении классов эквивалентности следует руководствоваться следующими правилами:

- Всегда будет, по меньшей мере, два класса: корректный и некорректный
- Если входное условие определяет диапазон значений, то, как правило бывает три класса: меньше чем диапазон, внутри диапазона и больше чем диапазон. (Значения на концах диапазона могут трактоваться как граничные значения.)
- Если элементы диапазона обрабатываются по-разному, то каждому варианту обработки будут соответствовать разные требования.

Другим примером разбиения на классы эквивалентности может служить тестирование открытия файла по его имени. В результате тестирования необходимо определить, все ли варианты имен обрабатываются системой согласно следующим тест-требованиям:

1. Проверить, что в случае присутствия в имени файла символов, не являющимися буквами латинского алфавита и цифрами, система выводит сообщение об ошибке.
2. Проверить, что в том случае, когда длина имени файла превышает 11 символов, система выдает сообщение об ошибке
3. Проверить, что система не различает регистр символов имени при открытии файла.
4. Проверить, что при открытии файлов с именами, не противоречащими требованиям 1-3, система открывает файл.

Входными значениями тестового примера являются различные имена файлов, выходными – реакция системы (ошибка или успешное открытие).

Можно выделить следующие классы эквивалентности:

По длине имени:

1. Длина имени меньше 11 символов
2. Длина имени равна 11 символам
3. Длина имени больше 11 символов

По символам:

4. Имя, состоящее из цифр и букв смешанного регистра
5. Имя, состоящее из цифр и букв нижнего регистра
6. Имя, состоящее из цифр и букв верхнего регистра
7. Имя, состоящее только из цифр
8. Имя, состоящее только из букв
9. Имя, включающее знаки препинания (не буквенно-цифровые символы)
10. Имя, включающее управляющие символы (не буквенно-цифровые символы)

Эти классы эквивалентности иллюстрируют, что проверки на границах интервалов применимы не только для тестирования арифметических операций и операций сравнения. Практически для любых данных, даже текстовых, можно определить «минимальные» и «максимальные» допустимые значения.

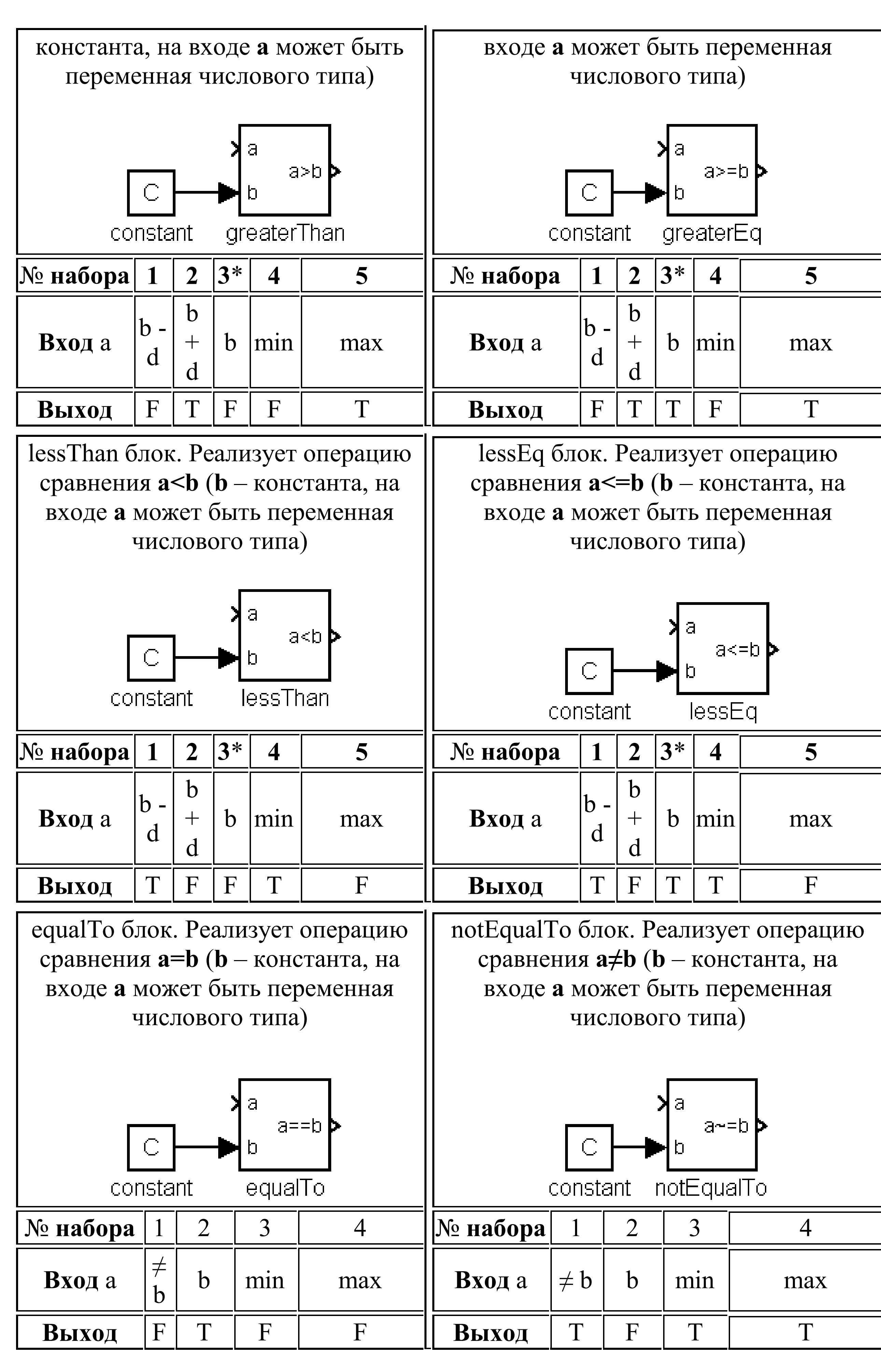
2.4.5. Тестирование операций сравнения чисел

Разбиение на классы эквивалентности широко используется при тестировании корректности реализации арифметических операций и операций сравнения. Каждую операцию можно рассматривать как блок с входами – значениям и выходом – результатом операции. Для ее тестирования выполняется разбиение диапазона изменения переменных на входах блока на классы эквивалентности и методом анализа граничных значений этих переменных.

В таблице 3 приведены тестовые наборы для блоков реализующих операции сравнения, в случае, когда на один из входов блока подаётся константа.

Таблица 3. Блоки сравнения и определённые для них тестовые наборы

greaterThan блок. Реализует операцию сравнения **a**>**b** (**b** –	greaterEq блок. Реализует операцию сравнения **a**>=**b** (**b** – константа, на



константа, на входе **а** может быть переменная числового типа)

№ набора	1	2	3*	4	5
Вход а	b - d	b + d	b	min	max
Выход	F	T	F	F	T

входе **а** может быть переменная числового типа)

№ набора	1	2	3*	4	5
Вход а	b - d	b + d	b	min	max
Выход	F	T	T	F	T

lessThan блок. Реализует операцию сравнения **a<b** (**b** – константа, на входе **а** может быть переменная числового типа)

№ набора	1	2	3*	4	5
Вход а	b - d	b + d	b	min	max
Выход	T	F	F	T	F

lessEq блок. Реализует операцию сравнения **a<=b** (**b** – константа, на входе **а** может быть переменная числового типа)

№ набора	1	2	3*	4	5
Вход а	b - d	b + d	b	min	max
Выход	T	F	T	T	F

equalTo блок. Реализует операцию сравнения **a=b** (**b** – константа, на входе **а** может быть переменная числового типа)

№ набора	1	2	3	4
Вход а	≠ b	b	min	max
Выход	F	T	F	F

notEqualTo блок. Реализует операцию сравнения **a≠b** (**b** – константа, на входе **а** может быть переменная числового типа)

№ набора	1	2	3	4
Вход а	≠ b	b	min	max
Выход	T	F	T	T

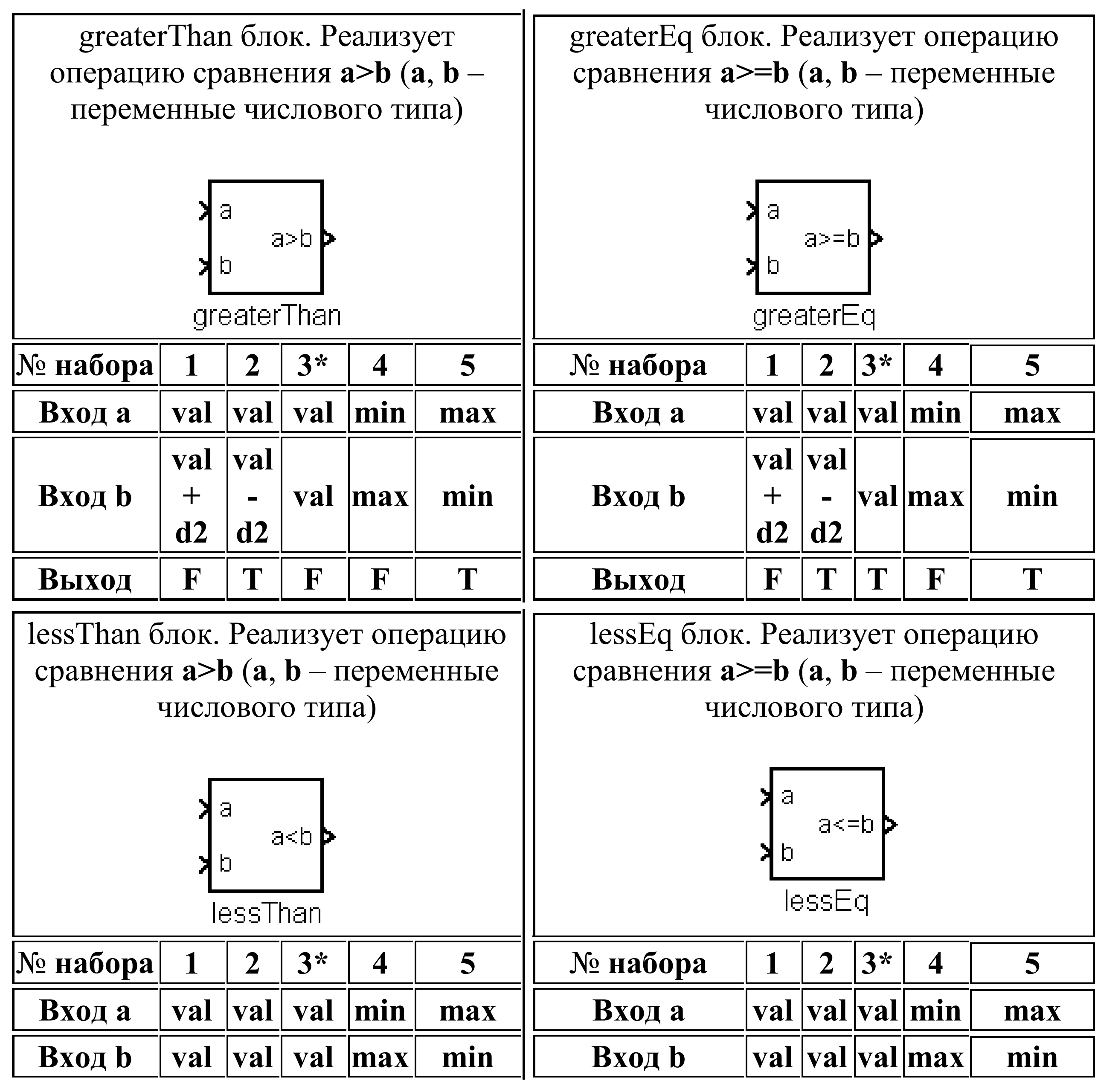
* тестовый набор реализуем только если переменная на входе a – переменная целого типа

В приведённых тестовых наборах используются следующие обозначения:

- d – шаг изменения (resolution) переменной на входе a. Если переменная на входе a – переменная целого типа, то d равно 1
- min – минимальное значение переменной на входе a
- max – максимальное значение переменной на входе a

В таблице 4 приведены тестовые наборы для блоков реализующих операции сравнения, в случае, когда на оба входа блока подаются переменные.

Таблица 4. Блоки сравнения и определённые для них тестовые наборы (продолжение)



greaterThan блок. Реализует операцию сравнения **a>b** (**a**, **b** – переменные числового типа)

№ набора	1	2	3*	4	5
Вход a	val	val	val	min	max
Вход b	val + d2	val - d2	val	max	min
Выход	F	T	F	F	T

greaterEq блок. Реализует операцию сравнения **a>=b** (**a**, **b** – переменные числового типа)

№ набора	1	2	3*	4	5
Вход a	val	val	val	min	max
Вход b	val + d2	val - d2	val	max	min
Выход	F	T	T	F	T

lessThan блок. Реализует операцию сравнения **a>b** (**a**, **b** – переменные числового типа)

№ набора	1	2	3*	4	5
Вход a	val	val	val	min	max
Вход b	val	val	val	max	min

lessEq блок. Реализует операцию сравнения **a>=b** (**a**, **b** – переменные числового типа)

№ набора	1	2	3*	4	5
Вход a	val	val	val	min	max
Вход b	val	val	val	max	min

	+ d2	- d2			
Выход	**T**	**F**	**F**	**T**	**F**

equalTo блок. Реализует операцию сравнения **a=b** (**a**, **b** – переменные любого типа)

№ набора	**1**	**2**	**3**	**4**
Вход a	**val1**	**val**	**min**	**max**
Вход b	**val2**	**val**	**max**	**min**
Выход	**F**	**T**	**F**	**F**

	+ d2	- d2			
Выход	**T**	**F**	**T**	**T**	**F**

notEqualTo блок. Реализует операцию сравнения **a≠b** (**a**, **b** – переменные любого типа)

№ набора	**1**	**2**	**3**	**4**
Вход a	**val1**	**val**	**min**	**max**
Вход b	**val2**	**val**	**max**	**min**
Выход	**T**	**F**	**T**	**T**

* тестовый набор реализуем только если переменные на входах блока - переменные целого типа

В приведённых тестовых наборах используются следующие обозначения:

- d2 – шаг изменения (resolution) переменной на входе b. Если переменная на входе b – переменная целого типа, то d2 равно 1
- val, val1, val2 – значения взятые из середины диапазона, полученного при пересечении диапазонов переменных на входах a и b
- min – минимальное значение переменной на входе блока
- max – максимальное значение переменной на входе блоке

2.5. Тест-планы

Тестовые примеры, рассматриваемые в предыдущих разделах, не существуют сами по себе – каждый тестовый пример проверяет одну ситуацию в работе системы, но вся совокупность тестовых примеров должна полностью проверять всю функциональность системы. В связи с этим описания тестовых примеров объединяют в документы, называемыми тест-планами.

Тест-план представляет собой документ, в котором перечислены все тестовые примеры, необходимые для тестирования системы, либо часть тестовых примеров, объединенных по определенному признаку.

Тест-план может быть написан на естественном или формальном языке, в последнем случае возможна передача тест-плана на вход тестового окружения для автоматического выполнения определенных в тест-плане тестовых примеров.

Существует несколько причин для объединения описаний тестовых примеров в единый документ или несколько документов:

- **Единая схема идентификации и трассировки тестовых примеров**

Поскольку тестовые примеры пишутся на основании функциональных или тест-требований, при тестировании необходимо удостовериться, что для каждого требования существует хотя бы один тестовый пример. Это достигается введением единой схемы идентификации тестовых примеров (например – сквозной нумерации) и введением ссылок на требования, на основе которых тестовый пример написан.

- **Объединение тестовых примеров в смысловые группы**

Тестовые примеры, предназначенные для проверки одних и тех же модулей системы рационально объединять в смысловые группы. Это связано с тем, что у таких примеров, как правило очень похожи входные данные и сценарии, а группировка позволяет выявлять опечатки и ошибки в тестах.

- **Внесение изменений в тестовые примеры**

При изменении тестируемой системы в ходе ее жизненного цикла неизбежно приходится изменять тестовые примеры. Общие обзоры тест-требований и тест-планов позволяют выявить, какие тесты должны быть изменены или удалены, а в каких смысловых группах необходимо создание новых тестовых примеров, проверяющих новую функциональность.

- **Определение последовательности тестирования**

Одно из важных свойств тестового примера, которое подробно будет рассматриваться в теме 3 – его независимость. Это означает, что результат выполнения тестового примера не должен изменяться в зависимости от того, какие тесты выполнялись до него. Как правило, независимость тестовых примеров достигается полной реинициализацией тестового окружения перед выполнением каждого нового тестового примера. Однако, часто возникают ситуации, в которых для экономии времени выполнения тестов, они объединяются в последовательности, в которых каждый следующий тестовый пример использует состояние тестового окружения или тестируемой системы, достигнутое во время предыдущего теста. Такие

связанные тестовые примеры должны быть отдельно помечены для того, чтобы сохранить корректный порядок их следования.

2.5.1. Типовая структура тест-плана

Рассмотрим типовую структуру тест-плана, написанного на естественном языке и содержащего тестовые примеры для проверки работы модуля расчета контрольных сумм.

Каждый тестовый пример в этом тест-плане имеет уникальный номер и ссылку на тест-требование, на основе которого он написан.

Общее описание теста помогает при сопровождении тест-планов – внесении изменений при изменении системы, инспекциях тест-планов, выявляющих несогласованность и т.п.

Также в каждом тестовом примере обязательно перечислены все входные значения и ожидаемые выходные значения, а также сценарий, описывающий последовательность действий, которые необходимо выполнить тестовому окружению для выполнения тестового примера.

Тест-план

Тестовый пример 1

Номер тест-требования: 2a, 2b

Описание теста: В данном тесте проверяется правильность вычисления значения контрольной суммы (поля CRC) при непустом значении поля CRC и нулевых значениях элементов записи

Входные данные: CRC = 12345, A=0, B=0, C=0, D=0

Ожидаемые выходные данные: CRC = 0, A=0, B=0, C=0, D=0, Empty = TRUE

Сценарий теста:

1. Установка значения поля CRC в 12345
2. Установка значений полей A-F в 0
3. Вызов функции Set_CRC
4. Проверка значений CRC на 0 и Empty на TRUE

Тестовый пример 2

Номер тест-требования: 2a

Описание теста: В данном тесте проверяется соответствие алгоритма вычисления поля CRC, заданному в спецификации требований.

Входные данные: CRC = 0, A-D заполнены байтами 01010101b

Ожидаемые выходные данные: CRC = 0111100b, Empty = FALSE

Сценарий теста:

1. Установка значения поля CRC в 0
2. Заполнение байт полей A-D байтами 01010101b
3. Вызов функции Set_CRC
4. Проверка значений CRC на 0111100b и Empty на FALSE

Тестовый пример 3

Номер тест-требования: 2a

Описание теста: В данном тесте проверяется неизменность полей A-F записи при вычислении поля CRC (подсчете контрольной суммы)

Входные данные: CRC = 0, A-D заполнены байтами 01010101b

Ожидаемые выходные данные: A-D заполнены байтами 01010101b,

Сценарий теста:

1. Установка значения поля CRC в 0
2. Заполнение байт полей A-D байтами 01010101b
3. Вызов функции Set_CRC
4. Проверка значений байт полей A-D на 01010101b

Такая структура тест-плана позволяет описывать тестовые примеры с совершенно различными наборами входных и выходных данных и сценариями, однако при большом количестве тестовых примеров эта схема слишком громоздка. В теме 4 будут рассмотрены табличные формы представления тест-планов, позволяющие записывать их более компактно.

2.6. Оценка качества тестируемого кода – статистика выполнения тестов

В результате выполнения каждого тестового примера тестовое окружение сравнивает ожидаемые и реальные выходные значения. В случае, если эти значения совпадают, тест считается пройденным, т.к. система выдала именно те выходные значения, которые ожидались, в противном случае тест считается не пройденным.

Каждый не пройденный тест потенциально указывает на потенциальный дефект в тестируемой системе, а общее их количество позволяет оценивать качество тестируемого программного кода и объем изменений, которые необходимо в него внести для устранения дефектов.

Для построения такой интегральной оценки после выполнения всех тестовых примеров тестовым окружением собирается статистика выполнения, которая как правило записывается в файл отчета о выполнении тестов. Существует несколько степеней подробности статистики выполнения тестов:

- Вывод количества пройденных и количества не пройденных тестовых примеров, а также их общего количества.

Например,

180 test cases passed

20 test cases failed

200 test cases total

- 1 + вывод идентификаторов не пройденных тестовых примеров. Позволяет локализовать тестовые примеры, потенциально выявившие дефект

Например,

Invoking test case 1 … Passed

Invoking test case 2 … Failed

Invoking test case 3 … Failed

<…>

Invoking test case 200 … Passed

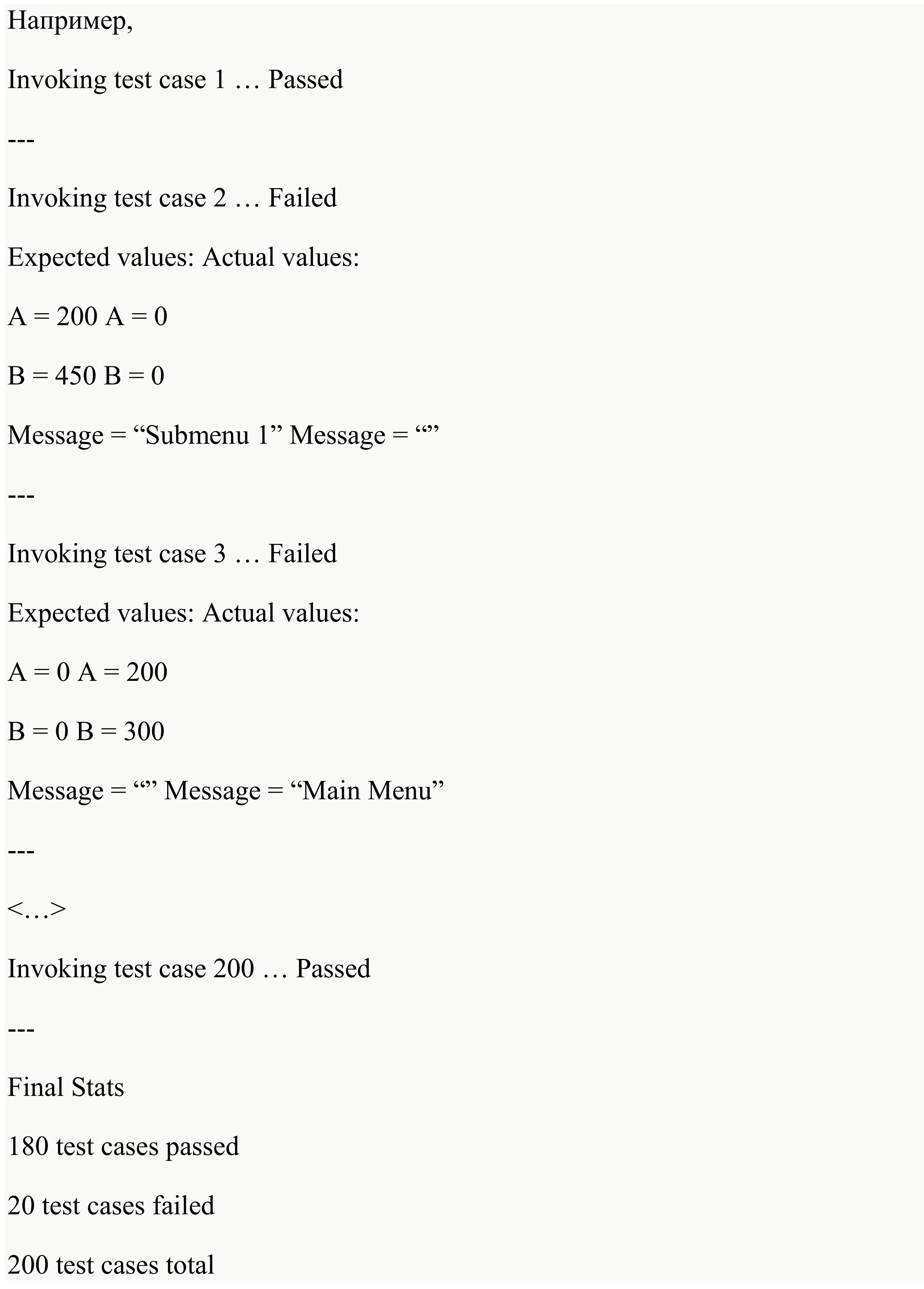
Final stats:

180 test cases passed

20 test cases failed

200 test cases total

- 2 + вывод не совпавших ожидаемых и реальных выходных данных. Позволяет проводить более глубокий анализ причин неуспешного прохождения тестового примера.



Например,

Invoking test case 1 … Passed

Invoking test case 2 … Failed

Expected values: Actual values:

A = 200 A = 0

B = 450 B = 0

Message = “Submenu 1” Message = “”

Invoking test case 3 … Failed

Expected values: Actual values:

A = 0 A = 200

B = 0 B = 300

Message = “” Message = “Main Menu”

<…>

Invoking test case 200 … Passed

Final Stats

180 test cases passed

20 test cases failed

200 test cases total

- 2 + вывод всех ожидаемых и реальных выходных данных. Вариант предыдущего пункта.

Например,

Invoking test case 1 … Passed

Invoking test case 2 … Failed

Expected values: Actual values:

A = 200 A = 0 FAIL

B = 450 B = 0 FAIL

C = 500 C = 500 P

D = 600 D = 600 P

Message = “Submenu 1” Message = “” FAIL

Invoking test case 3 … Failed

Expected values: Actual values:

A = 0 A = 200 FAIL

B = 0 B = 300 FAIL

C = 500 C = 500 P

D = 600 D = 600 P

Message = “” Message = “Main Menu” FAIL

<…>

Invoking test case 200 … Passed

Final Stats

180 test cases passed

20 test cases failed

200 test cases total

- Полный вывод ожидаемых и реальных выходных данных с отметками о совпадении и несовпадении и отметками об успешном/неуспешном завершении для каждого тестового примера.

Например,

Invoking test case 1 … Passed

A = 0 A = 0 P

B = 0 B = 0 P

C = 500 C = 500 P

D = 600 D = 600 P

Message = “” Message = “” P

Invoking test case 2 … Failed

Expected values: Actual values:

A = 200 A = 0 FAIL

B = 450 B = 0 FAIL

C = 500 C = 500 P

D = 600 D = 600 P

Message = “Submenu 1” Message = “” FAIL

Invoking test case 3 … Failed

Expected values: Actual values:

A = 0 A = 200 FAIL

B = 0 B = 300 FAIL

C = 500 C = 500 P

D = 600 D = 600 P

Message = “” Message = “Main Menu” FAIL

<…>

Invoking test case 200 … Passed

Message = “Submenu 1” Message = “Submenu 1 P

Prompt = “>” Prompt = “>” P

Final Stats

180 test cases passed

20 test cases failed

200 test cases total

Более детально различные форматы отчетов о тестировании будут рассмотрены в теме 4, а пока остановимся более подробно на важном критерии оценки качества системы тестов и степени полноты тестирования системы – уровне покрытия программного кода тестами.

2.7. Покрытие программного кода

2.7.1. Понятие покрытия

Одна из оценок качества системы тестов - это ее полнота, т.е. величина той части функциональности системы, которая проверяется тестовыми примерами. Обычно за меру полноты берут отношение объема проверенной части системы к ее объему в целом. Полная система тестов позволяет утверждать, что система реализует всю функциональность, указанную в требованиях, и, что еще более важно – не реализует никакой другой функциональности.

Один из часто используемых методов определения полноты системы тестов является определение отношения количества тест-требований, для которых существуют тестовые примеры, к общему количеству тест-требований. Т.е. в данном случае речь идет о покрытии тестовыми примерами тест-требований. В качестве единицы измерения степени покрытия здесь выступает процент тест-требований, для которых существуют тестовые примеры, называемый процентом покрытых тест-требований.

Покрытие требований позволяет оценить степень полноты системы тестов по отношению к функциональности системы, но не позволяет оценить полноту по отношению к ее программной реализации. Одна и та же функция может быть реализована при помощи совершенно различных алгоритмов, требующих разного подхода к организации тестирования.

Для более детальной оценки полноты системы тестов при тестировании стеклянного ящика анализируется *покрытие программного кода*, называемое также *структурным покрытием*.

Во время работы каждого тестового примера выполняется некоторый участок программного кода системы, при выполнении всей системы тестов выполняются все участки программного кода, которые задействует эта система тестов. В случае, если существуют участки программного кода, не выполненные при выполнении системы тестов, система тестов потенциально неполна (т.е. не проверяет всю функциональность системы), либо система содержит участки защитного кода или неиспользуемый код (например, «закладки» или задел на будущее использование системы). Таким образом, отсутствие покрытия каких-либо участков кода является сигналом к переработке тестов или кода (а иногда – и требований).

К анализу покрытия программного кода можно приступать только после полного покрытия требований. Полное покрытие программного кода не гарантирует того, что тесты проверяют все требования к системе. Одна из типичных ошибок начинающего тестировщика – начинать с покрытия кода, забывая про покрытие требований.

2.7.2. Уровни покрытия

2.7.3. По строкам программного кода (Statement Coverage)

Для обеспечения полного покрытия программного кода на данном уровне, необходимо, чтобы в результате выполнения тестов каждый оператор был выполнен хотя бы один раз.

Особенность данного уровня покрытия состоит в том, что на нем затруднен анализ покрытия некоторых управляющих структур.

Например, для полного покрытия всех строк следующего участка программного кода на языке C достаточно одного тестового примера:

Вход: condition = true; Ожидаемыйвыход: *p = 123.

```
int* p = NULL;

if (condition)

p = &variable;

*p = 123;
```

Даже если в состав тестов не будет входить тестовый пример, проверяющий работу фрагмента при значении condition = false, код будет покрыт. Однако, в случае condition = false выполнение фрагмента вызовет ошибку.

Аналогичные проблемы возникают при проверке циклов do … while – при данном уровне покрытия достаточно выполнение цикла только один раз, при этом метод совершенно нечувствителен к логическим операторам || и &&.

Другой особенностью данного метода является зависимость уровня покрытия от структуры программного кода. На практике часто не требуется 100% покрытия программного кода, вместо этого устанавливается допустимый уровень покрытия, например 75%. Проблемы могут возникнуть при покрытии следующего фрагмента программного кода:

```
if (condition)

    functionA();

else

    functionB();
```

Если functionA() содержит 99 операторов, а functionB() один оператор, то единственного тестового примера, устанавливающего condition в true, будет достаточно для достижения необходимого уровня покрытия. При этом аналогичный тестовый пример, устанавливающий значение condition в false даст слишком низкий уровень покрытия.

2.7.3.1. По веткам условных операторов (Decision Coverage)

Для обеспечения полного покрытия по данному методу каждая точка входа и выхода в программе и во всех ее функциях должна быть выполнена по крайней мере один раз и все логические выражения в программе должны

принять каждое из возможных значений хотя бы один раз, таким образом для покрытия по веткам требуется как минимум два тестовых примера.

Такжеданныйметодназывают: branch coverage, all-edges coverage, basis path coverage, DC, C2, decision-decision-path.

В отличие от предыдущего уровня покрытия данный метод учитывает покрытие условных операторов с пустыми ветками. Так, для покрытия по веткам участка программного кода

```
a = 0;

if (condition) {

a = 1;

}
```

необходимы два тестовых примера:

1. Вход: condition = true; Ожидаемый выход: a = 1;

2. Вход: condition = false; Ожидаемый выход: a = 0;

Особенность данного уровня покрытия заключается в том, что на нем не учитываются логические выражения, значения компонент которых получаются вызовом функций. Например, на следующем фрагменте программного кода

```
if ( condition1 && ( condition2 || function1() ) )

statement1;

else

statement2;
```

полное покрытие по веткам может быть достигнуто при помощи двух тестовых примеров:

1. Вход: condition1 = true, condition2 = true

2. Вход: condition1 = false, condition2 = true/false (любоезначение)

В обоих случаях не происходит вызова функции function1(), хотя покрытие данного участка кода будет полным. Для проверки вызова функции

function1() необходимо добавить еще один тестовы й пример (который, однако, не улучшает степени покрытия по веткам):

3. Вход: condition1 = true, condition2 = false.

2.7.3.2. По компонентам логических условий

Для более полного анализа компонент условий в логических операторах существует несколько методов, учитывающих структуру компонент условий и значения, которые они принимают при выполнении тестовых примеров.

2.7.3.3. Покрытие по условиям (Condition Coverage)

Для обеспечения полного покрытия по данному методу каждая компонента логического условия в результате выполнения тестовых примеров должна принимать все возможные значения, но при этом не требуется, чтобы само логическое условие принимало все возможные значения. Так, например, при тестировании следующего фрагмента:

if (condition1 | condition2)

functionA();

else

functionB();

Для покрытия по условиям потребуется два тестовых примера:

1. Вход: condition1 = true, condition2 = false
2. Вход: condition1 = false, condition1 = true.

При этом значение логического условия будет принимать значение только true, таким образом, при полном покрытии по условиям не будет достигаться покрытие по веткам.

2.7.3.4. Покрытиеповеткам/условиям (Condition/Decision Coverage)

Данный метод сочетает требования предыдущих двух методов – для обеспечения полного покрытия необходимо, чтобы как логическое условие, так и каждая его компонента приняла все возможные значения.

Для покрытия рассмотренного выше фрагмента с условием condition1 | condition2 потребуется 2 тестовых примера:

1. Вход: condition1 = true, condition2 = true

2. Вход: condition1 = false, condition1 = false.

Однако, эти два тестовых примера не позволят протестировать правильность логической функции – вместо OR в программном коде могла быть ошибочно записана операция AND.

2.7.3.5. Покрытие по всем условиям (Multiple Condition Coverage)

Для выявления неверно заданных логических функций был предложен метод покрытия по всем условиям. При данном методе покрытия должны быть проверены все возможные наборы значений компонент логических условий. Т.е. в случае n компонент потребуется 2^n тестовых примеров, каждый из которых проверяет один набор значений, Тесты, необходимые для полного покрытия по данному методу, дают полную таблицу истинности для логического выражения.

Несмотря на очевидную полноту системы тестов, обеспечивающей этот уровень покрытия, данный метод редко применяется на практике в связи с его сложностью и избыточностью.

Еще одним недостатком метода является зависимость количества тестовых примеров от структуры логического выражения. Так, для условий, содержащих одинаковое количество компонент и логических операций:

a && b && (c || (d && e))

((a || b) && (c || d)) && e

потребуется разное количество тестовых примеров. Для первого случая для полного покрытия нужно 6 тестов, для второго – 11.

2.7.4. Метод MC/DC для уменьшения количества тестовых примеров при 3-м уровне покрытия кода

Для уменьшения количества тестовых примеров при тестировании логических условий фирмой Boeing был разработан модифицированный метод покрытия по веткам/условиям (Modified Condition/Decision Coverage или MC/DC) [,]. Данный метод широко используется при верификации бортового авиационного программного обеспечения согласно процессам стандарта DO-178B [].

Для обеспечения полного покрытия по этому методу необходимо выполнение следующих условий:

- каждое логическое условие должно принимать все возможные значения
- каждая компонента логического условия должна хотя бы один раз принимать все возможные значения;
- должно быть показано независимое влияние каждой из компонент на значение логического условия, т.е. влияние при фиксированных значениях остальных компонент.

Покрытие по этой метрике требует достаточно большого количества тестов для того, чтобы проверить каждое условие, которое может повлиять на результат выражения, однако это количество значительно меньше, чем требуемое для метода покрытия по всем условиям. В таблице 5 приведены примеры тестовых наборов, необходимых для тестирования логических блоков по MC/DC. Так, например, для блока OR достаточно n+1 тестовых примеров, где n – количество входов логического блока. Первый тестовый пример показывает, что при нулевых значениях входов значение выхода также нулевое. В каждом из следующих n примеров значение каждого входа устанавливается в 1, чем показывается независимое влияние входов на значение выхода.

Таблица 5. Логические блоки и определённые для них тестовые наборы

AND блок. Реализует логическую функцию **И** для двух или более входов

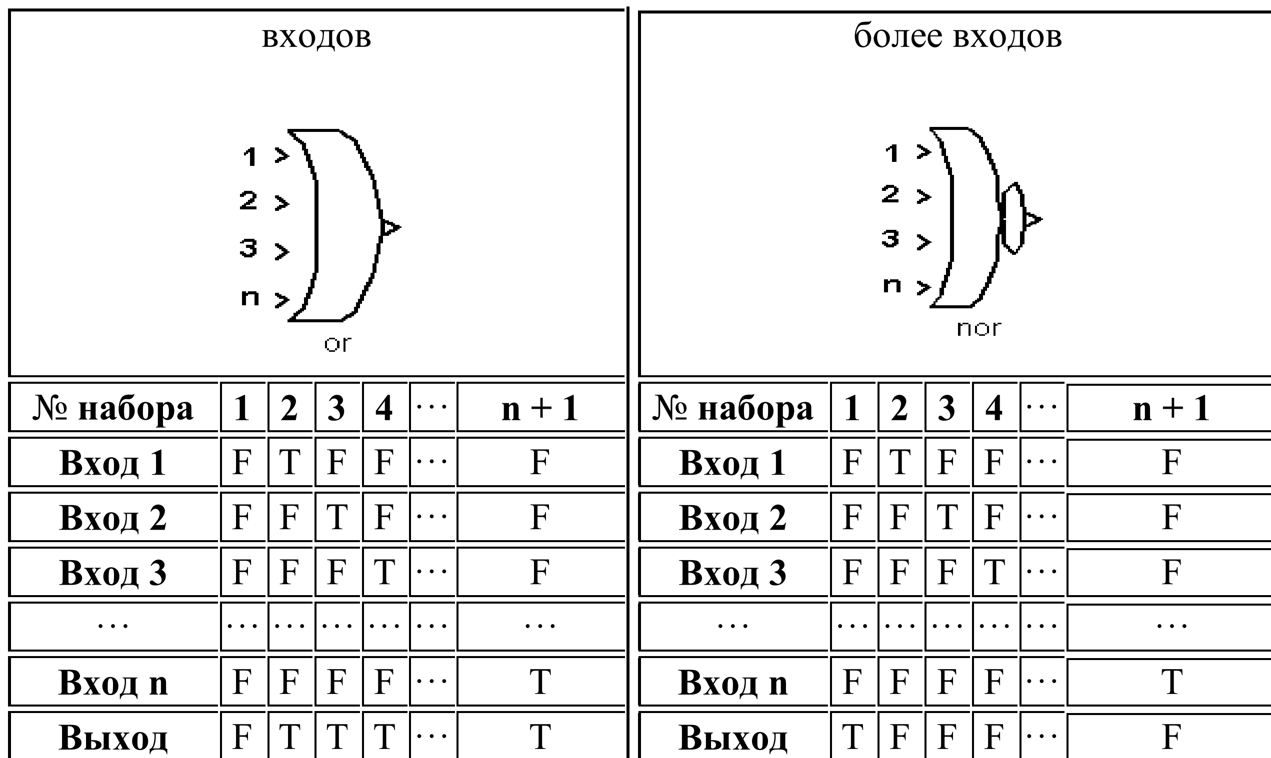
№ набора	**1**	**2**	**3**	**4**	…	**n + 1**
Вход 1	T	F	T	T	…	T
Вход 2	T	T	F	T	…	T
Вход 3	T	T	T	F	…	T
…	…	…	…	…	…	…
Вход n	T	T	T	T	…	F
Выход	T	F	F	F	…	F

NAND блок. Реализует логическую функцию **И-НЕ** для двух или более входов

№ набора	**1**	**2**	**3**	**4**	…	**n + 1**
Вход 1	T	F	T	T	…	T
Вход 2	T	T	F	T	…	T
Вход 3	T	T	T	F	…	T
…	…	…	…	…	…	…
Вход n	T	T	T	T	…	F
Выход	F	T	T	T	…	T

OR блок. Реализует логическую функцию **ИЛИ** для двух или более

NOR блок. Реализует логическую функцию **ИЛИ - НЕ** для двух или



№ набора	1	2	3	4	…	n + 1
Вход 1	F	T	F	F	…	F
Вход 2	F	F	T	F	…	F
Вход 3	F	F	F	T	…	F
…	…	…	…	…	…	…
Вход n	F	F	F	F	…	T
Выход	F	T	T	T	…	T

№ набора	1	2	3	4	…	n + 1
Вход 1	F	T	F	F	…	F
Вход 2	F	F	T	F	…	F
Вход 3	F	F	F	T	…	F
…	…	…	…	…	…	…
Вход n	F	F	F	F	…	T
Выход	T	F	F	F	…	F

2.7.5. Анализ покрытия

Целью анализа полноты покрытия кода является выявление участков кода, которые не выполняются при выполнении тестовых примеров. Тестовые примеры, основанные на требованиях, могут не обеспечивать полного выполнения всей структуры кода. Поэтому для улучшения покрытия проводится анализ полноты покрытия кода тестами и, при необходимости, проводятся дополнительные проверки, направленные на выяснение причины недостаточного покрытия и определение необходимых действий по его устранению. Обычно анализ покрытия выполняется с учетом следующих соглашений:

- анализ должен подтвердить, что полнота покрытия тестами структуры кода соответствует требуемому виду покрытия и заданному минимально допустимому проценту покрытия;
- анализ полноты покрытия тестами структуры кода может быть выполнен с использованием исходного текста, если программное обеспечение не относится к уровню А. Для уровня А необходимо проверить объектный код, сгенерированный компилятором на предмет: трассируется ли он в исходный текст или нет. Если объектный код не трассируется в исходный текст, должны быть проведены поверки объектного кода на предмет правильности генерации последовательности команд. Примером объектного кода, который напрямую не трассируется в исходный текст, но генерируется компилятором, может быть проверка выхода за заданные границы массива;

- анализ должен подтвердить правильность передачи данных и управления между компонентами кода.

Анализ полноты покрытия тестами может выявить часть исходного кода, которая не исполнялась в ходе тестирования. Для разрешения этого обстоятельства могут потребоваться дополнительные действия в процессе проверки программного обеспечения. Эта неисполняемая часть кода может быть результатом:

- недостатков в формировании тестовых примеров или тестовых процедур, основанных на требованиях: В этом случае должны быть дополнен набор тестовых примеров или изменены тестовые процедуры для обеспечения покрытия упущенной части кода. При этом может потребоваться пересмотр метода (методов), используемого для проведения анализа полноты тестов на основе требований;
- неадекватности в требованиях на программное обеспечение: В этом случае должны быть модифицированы требования на программное обеспечение, разработаны и выполнены дополнительные тестовые примеры и тестовые процедуры;
- «мертвый» код. Этот код должен быть удален и проведен анализ для оценки эффекта удаления и необходимости перепроверки;
- дезактивируемый код. Для дезактивируемого кода, который не предполагается к выполнению в каждой конфигурации, сочетание анализа и тестов должно продемонстрировать возможности средств, которыми непреднамеренное исполнение такого кода предотвращается, изолируется или устраняется. Для дезактивируемого кода, который выполняется только при определенных конфигурациях, должна быть установлена нормальная эксплуатационная конфигурация для исполнения этого кода и для нее должны быть разработаны дополнительные тестовые примеры и тестовые процедуры, удовлетворяющие целям полноты покрытия тестами структуры кода;
- избыточность условия. Логика работы твкого условия должна быть пересмотрена. Например, в условии if(A && B || !B) принципиально невозможно проверить, что часть условия A && B будет равна False в случае, когда A=True и B=False, так как вторая часть условия (!B) будет равна True и общий результат логического выражения будет True;
- защитный код. Эта часть кода используется для предотвращения исключительных ситуаций, которые могут возникнуть в процессе работы программы. Как пример, это может быть ветка default в операторе выбора switch, причем входное условие оператора switch может принимать определенные значения, которые он описывает, и как следствие, ветка default никогда не будет выполнена.

ТЕМА 3.Повторяемость тестирования (лекция 6)

3.1. Задачи и цели обеспечения повторяемости тестирования при промышленной разработке программного обеспечения

Как уже было сказано в предыдущих темах, тестирование программной системы – не разовое мероприятие, а постоянный процесс, активный в течение всего жизненного цикла разработки системы. В течение этого процесса система неизбежно изменяется – либо в результате исправления ошибок, либо в результате расширения ее функциональности. Задача тестировщика в такой ситуации – подтвердить, что новая или исправленная функциональность не вызвала новые ошибки, а если ошибки все-таки возникли – определить причины их возникновения.

Самый простой, но в то же время действенный способ такого подтверждения – полное выполнение всех тестовых примеров после каждого существенного изменения системы и сравнение результатов выполнения тестов до и после изменения.

Если результаты выполнения тестов до внесения изменений были положительными (все тесты проходили успешно), то появление неуспешно пройденных тестов может означать, что в системе появились новые дефекты, вызванные исправлением старых.

В общем случае повторное выполнение тестов может завершиться одним из трех способов:

1. Все тесты пройдены успешно. В этом случае изменения не затрагивают уже протестированные функции, но может потребоваться разработка новых тестовых примеров для новых функций системы.
2. Часть тестов, ранее выполнявшихся успешно, завершается с отрицательным результатом. Причины этого могут быть следующие:

- корректное изменение функциональности тестируемой системы, в результате которого тестовый пример перестал соответствовать требованиям;
- некорректное изменение функциональности системы, в результате которого тестовый пример выявил расхождение с требованиями;
- влияние остаточных данных от предыдущих тестовых примеров, ранее остававшееся незамеченным.

Первые две причины различимы только при помощи анализа изменений в функциональных требованиях и тест-требованиях, а также текущего состояния тест-планов и тестового окружения. По результатам этого анализа в первом случае тестировщик вносит изменения в тестовый пример (и, возможно, разрабатываются новые тестовые примеры), во втором случае тестировщик уведомляет разработчиков о наличии дефекта.

3. Выполнение тестов аварийно завершается в самом начале или при выполнении определенного тестового примера.

Данная проблема чаще всего связана с изменением внешнего окружения тестируемой части системы, которое моделирует тестовое окружение. В результате таких изменений могут меняться внешние интерфейсы, а также состав и формат входных и выходных данных. В результате тестовое окружение перестает обеспечивать необходимую для выполнения тестов инфраструктуру и возникает сбой процесса тестирования. Например, такой сбой может возникнуть в тестовом окружении при попытке обработать данные, выдаваемые системой в новом формате.

Если для выполнения тестов требуется сборка программных модулей тестового окружения и тестируемой системы в единый исполняемый код, то при изменении интерфейсов системы может возникнуть ситуация, когда невозможно не только выполнение тестов, а даже сборка окружения и системы. В этом случае также необходимо провести анализ изменений внесенных в систему и модифицировать в соответствии с ними тестовое окружение.

В некоторых случаях повторное выполнение всех тестов невозможно. Это может быть связано с большим временем выполнения всех тестов и ограниченным временем, отведенным на процесс тестирования. В этом случае часто применяется практика выборочного тестирования отдельных частей системы, затронутых изменениями. Полное тестирование при таком подходе проводится только после накопления достаточно большого количества изменений или на ключевых стадиях проекта.

Процесс, включающий в себя повторное выполнение тестов, называют *регрессионным тестированием*. Регрессионное тестирование включает в себя следующие стадии:

1. Анализ изменений в системе
2. Выбор тестовых примеров для проверки системы
3. Выполнение тестовых примеров
4. Анализ результатов выполнения
5. Модификация тестового окружения, тестовых примеров или уведомление разработчиков о дефекте системы.

Таким образом, можно определить следующие основные задачи повторяемости тестирования при внесении изменений:

- обеспечение возможности полного выполнения всех тестов, проверяющих функциональность системы или проведение

анализа, позволяющего выявить тесты, которые должны быть повторно выполнены для тестирования изменившейся функциональности;
- разработка тестовых примеров и тестового окружения с использованием методик, облегчающих модификацию при изменениях в тестируемой системе;
- разработка тестовых примеров, структура которых полностью исключает их взаимное влияние по остаточным данным.

Следствием повторяемости тестирования является постоянное обеспечение тестировщиков и разработчиков актуальной информацией о текущем состоянии системы и корректности изменений, внесенных в ходе разработки системы.

3.2. Предусловия для выполнения теста, настройка тестового окружения, оптимизация последовательностей тестовых примеров

Как уже было сказано ранее, входные данные в каждом тестовом примере явно задают начальное состояние тестируемой системы и режимы ее работы при выполнении тестового сценария.

Однако неявное влияние на выполнение теста оказывает и состояние тестового окружения. Под состоянием здесь понимается набор параметров, изменение любого из которых может повлиять либо на результат выполнения тестового примера, либо на возможность его корректной работы и завершения.

Например, для выполнения тестового примера тестируемой системе может потребоваться значительный объем дисковой или оперативной памяти. Если перед выполнением теста тестовое окружение зарезервирует эту память под свои нужды, выполнение теста окажется невозможным. Та же самая ситуация может возникнуть и в случае, если окружение не освободит память после выполнения предыдущего тестового примера.

Эта информация обычно отсутствует в тест-планах, однако требуемое для выполнения тестов состояние тестового окружения необходимо учитывать при разработке тестовых примеров.

Хорошей практикой является оформление проверок на допустимость состояния тестового окружения в виде предусловий для выполнения теста. Это позволяет диагностировать ситуации, возникающие при выборочном тестировании, приводящие к отказам тестового окружения.

Например, рассмотрим программную систему, которая может стартовать двумя различными способами – с настройками по умолчанию после включения (режим FACTORY_SETTINGS), и с последними сохраненными

настройками после перезагрузки (режим COLD_START). При этом при старте в режиме FACTORY_SETTINGS значения по умолчанию присваиваются всем настройкам системы, а после перезагрузки (режим COLD_START) все настройки остаются в значениях, установленных непосредственно перед перезагрузкой.

Для проверки следующих требований:

1. Проверить, что после включения системы настройки устанавливаются в значения по умолчанию.
2. Проверить, что после перезагрузки системы настройки устанавливаются в последнее сохраненное значение

необходимы как минимум три тестовых примера со следующими сценариями:

Тестовый пример 1

1. Включить систему в режиме FACTORY_SETTINGS
2. Проверить, что настройки имеют значения по умолчанию (в реальном тест-плане здесь должны быть проверки конкретных значений переменных)

Тестовый пример 2

1. Включить систему в режиме FACTORY_SETTINGS
2. Перезагрузить систему (вызвать ее старт в режиме COLD_START)
3. Проверить, что настройки имеют значения по умолчанию (в реальном тест-плане здесь должны быть проверки конкретных значений переменных)

Тестовый пример 3

1. Включить систему в режиме FACTORY_SETTINGS
2. Изменить значения настроек системы (в реальном тест-плане здесь должны быть установлены конкретные значения переменных)
3. Перезагрузить систему (вызвать ее старт в режиме COLD_START)
4. Проверить, что настройки имеют последние введенные значения (в реальном тест-плане здесь должны быть проверки конкретных значений переменных)

Первый пункт сценария во всех трех тестовых примерах одинаков. Если при этом первоначальный старт системы в режиме FACTORY_SETTINGS занимает значительное время, то суммарное время выполнения трех тестовых

примеров будет еще больше. Если общее количество подобных тестовых примеров достаточно велико (десятки и сотни), то при таком выполнении тестов будет нерационально расходоваться время на выполнение тестовых примеров – время на инициализацию системы в каждом тестовом примере будет превышать суммарное время выполнения «полезных» этапов сценариев тестовых примеров.

Для экономии времени можно инициализировать систему в режиме FACTORY_SETTINGS только в первом тестовом примере. Второй и третий тестовый примеры начнут свою работу из расчета, что система уже была включена в режиме FACTORY_SETTINGS, и все значения настроек уже установлены в некоторые значения. Сценарии тестовых примеров при этом будут выглядеть следующим образом:

Тестовый пример 1

1. Включить систему в режиме FACTORY_SETTINGS
2. Проверить, что настройки имеют значения по умолчанию (в реальном тест-плане здесь должны быть проверки конкретных значений переменных)

Тестовый пример 2

1. Перезагрузить систему (вызвать ее старт в режиме COLD_START)
2. Проверить, что настройки имеют значения по умолчанию (в реальном тест-плане здесь должны быть проверки конкретных значений переменных)

Тестовый пример 3

1. Изменить значения настроек системы
2. Перезагрузить систему (вызвать ее старт в режиме COLD_START)
3. Проверить, что настройки имеют последние введенные значения (в реальном тест-плане здесь должны быть проверки конкретных значений переменных)

При такой структуре тестовых примеров важна последовательность их выполнения. Первый тестовый пример инициализирует тестируемую систему и приводит ее в необходимое начальное состояние (запускает ее в режиме FACTORY_SETTINGS), второй и третий примеры, считая, что система уже инициализирована, проверяют только ее работу при перезагрузке.

В ходе разработки системы требования и программный код могут измениться таким образом, что при регрессионном тестировании может быть принято решение о выполнении тестов только для режима COLD_START.

Если при этом будут выполняться только тестовые примеры 2 и 3, то корректное выполнение сценария станет невозможным – значения настроек системы не получили значений по умолчанию при старте системы, а сама система запускается в нештатном режиме – перезагружается не включившись.

Для того, чтобы диагностировать такие ситуации, в состав предусловий тестовых примеров 2 и 3 необходимо включать проверки того, что к моменту выполнения тестового примера система находится в необходимом состоянии. Первый тестовый пример при этом может выставлять некоторый флаг (переменную в тестовом окружении), установленное значение которого будет сигнализировать о том, что система корректно стартовала

При наличии таких проверок тестовые примеры будут выглядеть следующим образом:

Первоначальные установки тестового окружения

Установить значение флага Флаг_Система_Стартовала = FALSE

Тестовый пример 1

1. Включить систему в режиме FACTORY_SETTINGS
2. Установить значение флага Флаг_Система_Стартовала = TRUE
3. Проверить, что настройки имеют значения по умолчанию (в реальном тест-плане здесь должны быть проверки конкретных значений переменных)

Тестовый пример 2

1. Проверить, что флаг Флаг_Система_Стартовала = TRUE, иначе прервать тестирование с выдачей диагностического сообщения.
2. Перезагрузить систему (вызвать ее старт в режиме COLD_START)
3. Проверить, что настройки имеют значения по умолчанию (в реальном тест-плане здесь должны быть проверки конкретных значений переменных)

Тестовый пример 3

1. Проверить, что флаг Флаг_Система_Стартовала = TRUE, иначе прервать тестирование с выдачей диагностического сообщения.

2. Изменить значения настроек системы (в реальном тест-плане здесь должны быть установлены конкретные значения переменных)
3. Перезагрузить систему (вызвать ее старт в режиме COLD_START)
4. Проверить, что настройки имеют последние введенные значения (в реальном тест-плане здесь должны быть проверки конкретных значений переменных)

При таком подходе для выполнения тестовых примеров сначала должны быть произведены первоначальные установки тестового окружения, после чего перед выполнением тестового примера 2 или 3 будет проведена проверка состояния тестируемой системы.

Пример может показаться несколько надуманным, однако, на практике часто возникает ситуация в которой друг за другом следует несколько десятков тестовых примеров, а при регрессионном тестировании требуется выполнить, например, тестовые примеры с номерами от 25 по 40. Первый тестовый пример при этом инициализирует систему, а остальные работают с уже стартовавшей системой. Если просто выполнять тестовые примеры 25-40, то их выполнение окажется невозможным – они не инициализируют систему. Разумным выходом из этой ситуации является выполнение тестовых примеров 1, 25-40.

3.3. Зависимость между тестовыми примерами, настройки по умолчанию для тестовых примеров и их групп

Для облегчения проведения регрессионного тестирования (и тестирования вообще) тестовые примеры часто разбивают на группы. Каждая группа содержит набор тестовых примеров, проверяющих отдельную локальную часть функциональности тестируемой системы. При отборе тестовых примеров для частичного регрессионного тестирования их можно отбирать сразу группами.

Тестовые примеры из предыдущего раздела можно разбить на две группы:

Тестирование старта системы: тестовый пример 1

Тестирование перезагрузки системы: тестовые примеры 2-3

Разбиение тестовых примеров на группы удобно и с точки зрения установки начального состояния тестового окружения для выполнения тестов – так, перед выполнением группы тестов можно инициализировать значения переменных или состояние системы, необходимое для выполнения всей группы. Например, если система работает в двух режимах – нормальном и сервисном, то перед выполнением группы тестов для нормального режима работы системы, устанавливать нормальный режим, а перед выполнением

тестов для сервисного режима – сервисный. Такие установки называются настройками группы тестов по умолчанию (group defaults, test group defaults).

Перед выполнением каждого тестового примера может потребоваться установка одних и тех же переменных в одни и те же значения. Для того, чтобы не дублировать эти установки в описании каждого тестового примера, в тест-плане можно определить настойки по умолчанию для каждого теста (test case defaults), например следующим образом:

Первоначальные установки тестового окружения

Установить значение флага Флаг_Система_Стартовала = FALSE

Настройки по умолчанию для группы:

Установить сервисный режим работы системы

Настройки по умолчанию для тестового примера:

Обнулить значения выходных переменных тестового окружения, в котором сохраняются настройки системы.

Группа 1: Тестирование старта системы (режим FACTORY_SETTINGS)

Тестовый пример 1

1. Включить систему в режиме FACTORY_SETTINGS
2. Установить значение флага Флаг_Система_Стартовала = TRUE
3. Проверить, что настройки имеют значения по умолчанию (в реальном тест-плане здесь должны быть проверки конкретных значений переменных)

Группа 2: Тестирование перезагрузки системы (режим COLD_START)

Тестовый пример 2

1. Проверить, что флаг Флаг_Система_Стартовала = TRUE, иначе прервать тестирование с выдачей диагностического сообщения.
2. Перезагрузить систему (вызвать ее старт в режиме COLD_START)
3. Проверить, что настройки имеют значения по умолчанию (в реальном тест-плане здесь должны быть проверки конкретных значений переменных)

Тестовый пример 3

1. Проверить, что флаг Флаг_Система_Стартовала = TRUE, иначе прервать тестирование с выдачей диагностического сообщения.
2. Изменить значения настроек системы (в реальном тест-плане здесь должны быть установлены конкретные значения переменных)
3. Перезагрузить систему (вызвать ее старт в режиме COLD_START)
4. Проверить, что настройки имеют последние введенные значения (в реальном тест-плане здесь должны быть проверки конкретных значений переменных)

Как видно из предыдущего раздела, для облегчения проведения выборочного регрессионного тестирования каждый тестовый пример должен быть полностью автономным – ход его выполнения и тем более, результат, не должны зависеть от предыдущих тестовых примеров. Тем самым при выборочном тестировании результат тестирования не зависит от выбранного набора тестовых примеров (тестового набора). Однако, на практике создание автономных тестов зачастую невозможно по различным причинам (как правило – из-за длительного времени выполнения таких тестов).

В случае, когда в наборе тестовых примеров тесты не являются автономными, говорят о *тестовой зависимости*. Тестовая зависимость бывает двух видов – предусмотренная структурой тестовых примеров и паразитная.

Пример предусмотренной тестовой зависимости был рассмотрен в предыдущем разделе – корректность выполнения тестов определялась порядком их выполнения. Такая тестовая зависимость требует документирования и сопровождения, как и сами описания тестовых примеров. Существует два вида документирования тестовых зависимостей:

- явное определение допустимого порядка выполнения тестовых примеров;
- определение допустимого порядка выполнения тестовых примеров при помощи предусловий.

Первый способ удобен при сравнительно небольшом общем количестве тестовых примеров, а в случае разбиения на группы – при небольшом размере групп тестовых примеров. При втором способе корректность порядка выполнения тестовых примеров определяется при помощи проверки того, что либо тестируемая система, либо тестовое окружение находятся в необходимом состоянии для выполнения тестового примера.

Паразитные тестовые зависимости обычно вызваны некорректным составлением тест-плана. Проявляются они, как и предусмотренные зависимости, в том, что один (или более) тестовых примеров корректно

работает только в том случае, если до него были выполнены другие тестовые примеры. Причем такая зависимость не является предусмотренной тестировщиком. Природа паразитной тестовой зависимости схожа с природой ошибок использования неинициализированных или остаточных данных в динамической памяти при программировании.

ТЕМА 4.Документация, сопровождающая процесс верификации и тестирования (лекции 6-8)

4.1. Технологические процессы верификации и роли в проекте, документация, создаваемая в ходе жизненного цикла проекта, ее назначение

В ходе работы над проектом по созданию любой сложной программной системы создается большое количество проектной документации. Основное ее назначение – координация совместных действий большого количества разработчиков в течение более или менее длительных промежутков времени – в процессе первоначальной разработки системы, в процессе выполнения работ по ее модификации, в процессе сопровождения. Структурный состав проектной документации в большинстве проектов практически одинаков – это требования к системе различного уровня (системные, функциональные и структурные), описание ее архитектуры, программный код, тесты и документы, сопровождающие процесс внедрения (руководства по установке, настройке, пользовательские руководства).

Поскольку верификация программной системы (в оптимистичном случае) выполняется в течение всего жизненного цикла разработки достаточно большим коллективом разработчиков, при тестировании создается тестовая документация. Основное ее назначение, помимо синхронизации действий тестировщиков различных уровней – обеспечение гарантий того, что тестирование выполняется в соответствии с выбранными критериями оценки качества, а также того, что все аспекты поведения системы протестированы. Кроме того, тестовая документация используется при внесении изменений в систему для проверки того, что как старая, так и новая функциональность работает корректно (Рис. 14).

Перед началом верификации менеджером тестирования (test manager) создается документ, называемый планом верификации (или планом тестирования, но это не то же самое, что тест-план). План тестирования – организационный документ, содержащий требования к тому, как должно выполняться тестирование в данном конкретном проекте. В нем определяются общие подходы к согласованию процессов разработки и верификации, определяются методики проведения верификации, состав тестовой документации и ее взаимосвязь с документацией разработчиков, сроки различных этапов верификации, различные роли и квалификация

тестировщиков, необходимые для выполнения всех работ по тестированию, требования к инструментам тестирования и тестовым стендам, оцениваются риски и приводятся пути для их преодоления.

В данном документе также определяются требования собственно к тестовой документации – тест-требованиям, тест-планам, отчетам о выполнении тестирования.

Согласно этим требованиям по системным и функциональным требованиям разработчиками тестов (test procedure developers) создаются тест-требования – документы, в которых подробно описано то, какие аспекты поведения системы должны быть протестированы. На основании описания архитектуры создаются низкоуровневые тест-требования, в которых описываются аспекты поведения конкретной программной реализации системы, которые необходимо протестировать.

Рис. 14 Документация, сопровождающая процесс верификации

На основании тест-требований разработчиками тестов (test developers) создаются тест-планы – документы, которые содержат подробное пошаговое описание того, как должны быть протестированы тест-требования.

На основании тест-требований и проектной документации разработчиков также создается тестовое окружение, необходимое для корректного выполнения тестов на тестовых стендах – драйверы, заглушки, настроечные файлы и т.п.

По результатам выполнения тестов тестировщиками (testers) создаются отчеты о выполнении тестирования (они могут создаваться либо автоматически, либо вручную), которые содержат информацию о том, какие несоответствия требованиям были выявлены в результате тестирования, а также отчеты о покрытии, содержащие информацию о том, какая доля программного кода системы была задействована в результате выполнения тестирования.

По несоответствиям создаются отчеты о проблемах – документы, которые направляются на анализ в группу разработчиков с целью определения причины возникновения несоответствия.

Изменения в систему вносятся только после всестороннего изучения этих отчетов и локализации проблем, вызвавших несоответствие требованиям. Для того, чтобы процесс изменений не вышел из под контроля и любое изменение протоколировалось (и связывалось с тестами, обнаружившими проблему), создается запрос на изменение системы. После завершения всех работ по запросу на изменение процесс тестирования повторяется до тех пор, пока не будет достигнут приемлемый уровень качества программной системы.

Форматы различных тестовых документов описаны в стандартах IEEE 1012 [] и IEEE 829 [], при дальнейшем изложении мы будем придерживаться духа этих стандартов.

Следует особо отметить, что все документы должны иметь уникальные идентификаторы и храниться в единой базе документов проекта. Это позволит сохранить управляемость процессом тестирования и поддерживать необходимое качество разрабатываемой системы. Нет ничего хуже ситуации, в которой найденная проблема не была исправлена из-за того, что отчет о ней был утерян и не попал к разработчику.

4.2. Стратегия и планы верификации

Первый документ, входящий в состав технологической документации процесса верификации – стратегия тестирования. Стратегия верификации определяет общие подходы и методики верификации, необходимые уровни

верификации проектной документации и программного кода, технологии и инструментальные средства.

Другой, не менее важный документ создаваемый перед началом процесса верификации – план верификации. Этот план содержит последовательное описание всех этапов верификации, процедур на каждом этапе и связей с этапами разработки.

Для каждого этапа определяется:

- типы входных и выходных документов;
- общая процедура верификации;
- роли и ответственности;
- форматы и соглашения по идентификации выходных документов;
- критерии оценки результативности этапа.

Иногда план верификации разделяется на отдельные документы, описывающие более подробно каждый из этапов, например:

- план верификации системных требований;
- план верификации архитектуры;
- план тестирования программного кода;
- план тестирования модулей и их интеграции;
- план системного тестирования;
- план нагрузочного тестирования;
- план полунатурных испытаний;
- план приемо-сдаточных испытаний.

Согласно разделу 4 стандарта IEEE 829 [] основная задача плана тестирования как документа – определение границ тестирования, подхода к тестированию, требуемых для тестирования ресурсов, плана-графика тестирования. План тестирования определяет тестируемые элементы и функции системы, задачи, решаемые в ходе тестирования, сотрудников, ответственных за тестирование и риски, связанные с этим планом. Такая форма плана тестирования является достаточно полной и включает в себя не только технические аспекты, связанные собственно с описанием тестовых примеров, но и организационные, связанные с общим управлением процессом тестирования. На практике объемы технических и организационных разделов планов тестирования могут достаточно сильно варьироваться. Однако, минимально необходимые элементы, которые рекомендуется включать в каждый план тестирования это:

- *идентификатор плана тестирования и номер его версии*, который позволяет однозначно находить нужный план тестирования и его последнюю актуальную версию в базе данных проекта;
- *общее описание тест-плана*;
- *трассировка на другие документы* – стандарты, планы тестирования, тест-требования, результаты выполнения тестов;
- *определение тестируемых областей системы* – указание частей проектной документации, исходных текстов, исполняемого кода, подвергаемых верификации с указанием типа верификации (автоматизированные тесты, формальные инспекции, тестирование на моделях, полунатурные испытания и т.п.)
- *определение подходов к тестированию* – определение общих методик, которых следует придерживаться при тестировании системы. Несмотря на то, что большинство тестов могут довольно сильно различаться, общие методы и подходы к их построению могут быть едиными.
- *критерий успешности/неуспешности прохождения тестов (pass/fail criteria)* – в данном разделе описывается, то, каким образом определяется успешность прохождения тестов для различных частей системы.
- *тестовые документы* – как правило, план тестирования содержит в качестве приложений все тестовые документы более низких уровней – тест-требования, тест-планы, результаты выполнения тестов, данные о сборе покрытия. В случае, если включать эти документы в состав плана тестирования представляется нецелесообразным (например, в случае их значительного объема), рекомендуется помещать ссылки на эти документы.
- *требования к среде тестирования* – данный раздел описывает требования к аппаратным и программным средствам, необходимым для проведения тестирования. В случае встроенного программного обеспечения программная система обычно работает на специальном аппаратном обеспечении, а инструментальные средства для тестирования – на обычных PC общего назначения. Для выполнения тестирования в таких условиях требуется либо использование эмуляторов, либо программно-аппаратный комплекс для сопряжения специального аппаратного обеспечения с PC. Кроме того, как правило, в состав программных средств тестирования входят кросс-средства разработки. В случае, если тестируется система общего назначения, то в данном разделе просто перечисляются требования к оборудованию, необходимому для тестирования, которые, как правило, несколько выше, чем требования к оборудованию, достаточному для работы системы.
- *Людские ресурсы и уровень их подготовки* – в данном разделе приводится состав группы тестирования, необходимый для успешного

завершения тестирования в поставленные сроки, а также приводится необходимые знания для различных ролей в группе.

- *План-график тестирования* – содержит сроки всех фаз тестирования
- *Риски* – содержит список рисков, которые могут помешать завершить тестирование в срок или с необходимым уровнем качества. Как правило, для каждого риска оценивается вероятность его возникновения, а также приводятся общие пути, при помощи которых можно избежать возникновения риска или ликвидировать его последствия

Стратегия и планы тестирования несколько отличаются от другой документации, относящейся к процессу тестирования. В первую очередь это связано с тем, что в этих документах достаточно много внимания уделяется тому, как должен быть организован процесс тестирования, а не тому, как тестировать саму систему.

4.3. Тест-требования

4.3.1. Технологические цепочки и роли участников проекта, использующих тест-требования. Связь тест-требований с другими типами проектной документации.

Тест-требования – основной документ для тестировщика, который определяет функциональность системы с точки зрения того, что должно быть проверено для того, чтобы удостовериться в ее корректном функционировании, а также – на основании какого внешнего эффекта можно убедиться, что проверяемая функция реализована правильно.

Существует два подхода к написанию тест-требований – функциональный и структурный. Тест-требования, написанные в рамках функционального подхода основываются на системных требованиях и требованиях к программному обеспечению системы.

Тест-требования, написанные в рамках структурного подхода пишутся на основании описания архитектуры системы и принимают в расчет строение исходных текстов системы. Из-за такого различия функциональный и структурный подходы часто называют подходами черного и белого ящиков. Структурные тест-требования важны в том случае, когда к надежности системы предъявляются повышенные требования. Т.е. когда важно проверить не только насколько корректно система в целом отрабатывает сценарии своей работы (корректные и некорректные с точки зрения пользователя), но и как в различных нестандартных ситуациях будут вести себя отдельные ее компоненты.

Рис. 15 Место тест-требований среди проектной документации

На практике почти всегда применяются оба подхода к разработке тест-требований, в результате в состав документации проекта включаются тест-требования верхнего уровня и тест-требования нижнего уровня, по которым составляются тест-планы (Рис. 15).

4.3.2. Свойства тест-требований

Как уже говорилось выше, тест-требования содержат описание требованиий по проверке всех основных функций системы. Тест-требования должны быть достаточными построения тест-плана проверки реализации задачи без знакомства с ее программными текстами, т.е. тест-требования должны обладать свойством *изоляции от внутренней структуры системы*.

Как правило, структура тест-требований следует структуре раздела функциональных требований на систему. Задача каждого требования - определение того, **что** надо проверить. Техника исполнения каждой такой проверки - задача тест-плана. Обычный формат описания отдельного требования следующий:

Проверить, что при <описание внешнего воздействия> [происходит] <описание реакции программы >.

Тест-требования, написанные в рамках функционального подхода обычно разделяют на следующие группы:

- функции контроля входных данных,
- функции обработки ошибок (ввода, вычислений),
- функции получения основного результата,

- функции обработки особых ситуаций,
- функции оформления и вывода результатов.

Конкретизация программной реализации может потребовать уточнений или расширений реакций на различные ситуации, возникающие при решении задачи. В этом случае рекомендуется оформить дополнительные тест-требования низкого уровня для структурной проверки системы.

Совокупность тест-требований должна обладать некоторыми важными свойствами: полнота, верифицируемость и непротиворечивость.

Как правило, одному системному или функциональному требованию соответствует минимум одно тест-требование. Если совокупность проверок, задаваемых тест-требованиями, покрывает всю функциональность системы, определенную в системных требования и требованиях к программному обеспечению, то говорят о *полноте* тест-требований. При изменениях требований к системе для поддержания полноты должны меняться и тест-требования.

Как системные требования и требования к ПО, так и тест-требования должны обладать свойством *верифицируемости*. Т.е. для каждого требования должна существовать возможность определить четкий критерий проверки – выполняется это требование в реализованной системе или нет.

Примером не верифицируемого требования может служить следующее «требование»:

Система должна иметь интуитивно понятный пользовательский интерфейс.

Очевидное «тест-требование» будет выглядеть как

Проверить, что система имеет интуитивно понятный пользовательский интерфейс

Без четкого определения критериев интуитивной понятности, проверить такое требование при помощи написания тестовых примеров не представляется возможным. Однако, если сопроводить такое требование количественными или качественными характеристиками интуитивно понятного интерфейса – написание тестовых примеров по требованиям становится возможным. Так, среди критериев интуитивной понятность могут быть следующие: глубина вложенности меню не более трех, наличие всплывающих подсказок на каждом элементе управления каждой экранной формы и т.п.

При большом количестве тест-требований и частых их изменениях может возникнуть ситуация, в которой различные требования перестают быть согласованными. В этом случае такие требования имеют взаимоисключающие друг друга критерии проверки. Т.е., например, в простом случае, одно тест-требование на пользовательский интерфейс может декларировать необходимость проверки того, что введенный пользователем пароль имеет длину не более 16 символов, а тест-требование к базе данных системы – что допустимый размер пароля, сохраняемого в БД – от 4 до 12 символов. В этом случае эти два требования являются противоречивыми. Для того, чтобы устранить это противоречие нужно проводить анализ системных и функциональных требований с последующей модификацией тест-требований. Тест-требования по которым составляются тест-планы для тестирования системы обычно обладают свойством *непротиворечивости*, поскольку противоречия обычно устраняются на уровне верификации проектной документации. Однако противоречия могут быть выявлены и позже, в результате попытки создать адекватные тестовые примеры.

4.4. Тест-планы

4.4.1. Технологические цепочки и роли участников проекта, использующих тест-планы. Связь тест-планов с другими типами проектной документации.

На основании тест-требований составляются тест-планы - программы испытаний (проверки, тестирования) программной реализации системы. В отличие от тест-требований в тест-плане описываются конкретные способы проверки функциональности системы, т.е. то, **как** должна проверяться функциональность. Как правило, тест-план состоит из отдельных тестовых примеров, каждый из которых проверяет некоторую функцию или набор функций системы. Для каждого тестового примера однозначно определяется критерий успешного прохождения (pass/fail criteria), при помощи которого можно судить о том – соответствует ли поведение системы заданному в требованиях или нет (Рис. 16).

Рис. 16 Место тест-планов среди проектной документации

Критерием качества тест-плана является покрытие (выполнение) всех требований к проверке правильности функционирования программной реализации. Желательной характеристикой тест-плана является проверка исполнения всех веток схемы программной реализации.

Структура тест-плана может соответствовать структуре тест-требований или следовать логике внешнего поведения системы. Каждый пункт тест-плана описывает, **как** производится проверка правильности функционирования программной реализации, и содержит:

- ссылку на требование(я), которое проверяется этим пунктом;
- конкретное входное воздействие на программу (значения входных данных);
- ожидаемую реакцию программы (тексты сообщений, значения результатов)
- описание последовательности действий, необходимых для выполнения пунктов тест-плана.

В состав тест-плана рекомендуется дополнительно включать пункты, служащие для проверки ветвей программы, не выполнявшихся при проверке удовлетворения функциональных требований. Такие пункты тест-плана могут иметь указание “Для полноты покрытия” в поле ссылки.

Тест-план может готовиться в формализованной форме и служить входным документом для тестовой оснастки, по которому тесты будут выполняться в автоматическом режиме с автоматической фиксацией результатов. В случае, если тест-план готовится в виде текстового документа, возможно только ручное тестирование системы по данному тест-плану.

4.4.2. Возможные формы подготовки тест-планов

Форма представления тест-плана в первую очередь зависит от того, каким образом тест-план будет использоваться в процессе тестирования. При ручном тестировании удобно представление тест-планов в виде текстовых документов, в которых отдельные разделы представляют собой описания тестовых примеров. Каждый тестовый пример в таком случае включает в себя перечисление последовательности действий, которые необходимо выполнить тестировщику для проведения тестирования – *сценария теста*, а также ожидаемые отклики системы на эти действия. Такая форма представления тест-плана неудобна для автоматизации тестирования, поскольку описания на естественном языке практически не поддаются формализации.

Для автоматизированного тестирования сценарий теста может записываться на каком-либо формальном языке, в этом случае возможно непосредственное использование тест-планов как входных данных для среды тестирования.

Другой формой представления тест-планов является таблица. Эта форма наиболее часто используется при четко и формально определенных входных потоках данных системы. Например, каждый столбец таблицы может

представлять собой тестовый пример, каждая строка – описание входного потока данных, а в ячейке таблицы записывается передаваемое в данном тестовом примере в данный поток значение. Ожидаемые значения для данного теста записываются в аналогичной таблице, в которой в строках перечисляются выходные потоки данных.

И, наконец, третьей формой представления тестовых примеров является определение примеров в виде конечного автомата. Такая форма представления используется при тестировании протоколов связи или при тестировании программных модулей, взаимодействие которых с внешним миром производится при помощи обмена сообщениями по заранее заданному интерфейсу. Модуль при этом может быть представлен как конечный автомат с набором состояний, а тест-план будет состоять из двух частей – описания переходов между состояниями и их параметров и тестовых примеров, в которых задается маршрут перехода между состояниями, параметры переходов и ожидаемые значения. Такое представление тест-плана может быть пригодно как для ручного, так и для автоматизированного тестирования.

4.4.3. Сценарии

Представление сценариев, удобное для ручного тестирования – тест-план в виде текстового документа, в котором каждый тестовый пример представляет один раздел. Для каждого тестового примера в этот документ записывается следующая информация:

- идентификатор;
- описание теста и его цель;
- ссылки на тестируемую часть системы;
- ссылки на используемую проектную документацию, в частности тест-требования;
- перечисление действий сценария;
- ожидаемая реакция системы на каждый пункт сценария.

Подразумевается, что действия сценария должны быть описаны таким образом, чтобы их мог воспроизвести человек с практически любым уровнем подготовки. Описание ожидаемой реакции системы должно также быть записано таким образом, чтобы можно было однозначно судить о том – соответствует реакция ожидаемой или нет.

Так, неудачной ожидаемой реакцией при ручном тестировании была бы запись

Сообщение «Загрузка» пропадает через приемлемое время

Степень приемлемости здесь будет зависеть от терпеливости тестировщика, и обеспечить повторяемость тестирования будет затруднительно. Более удачной формой описания той же самой ожидаемой реакции будет

Сообщение «Загрузка» исчезает с экрана не более, чем через 10 секунд после появления.

Ниже приведен пример описания тестового примера в виде сценария, предназначенного для ручного тестирования:

Группа тестов: Работа с учетными записями

Тестовый пример: 1289-15

Назначение: Проверка того, что учетная запись пользователя проверяется перед началом передачи данных и в случае ввода записи по умолчанию при максимальной защите системы передачи не происходит

Тест-требования: 8.5.8.1, 8.5.8.2

Предусловия для теста: Система должна быть приведена в состояние «Максимальная защита» и сброшена в настройки по умолчанию

Критерий прохождения теста: Все ожидаемые значения совпадают с реальными

Сценарий тестирования:

№	**Шаг сценария**	**Ожидаемый результат**
1	Запустить терминальный клиент и соединиться с системой по адресу 127.0.0.1	Должно появиться приглашение терминала TRANSFER>
2	Запустить процесс передачи данных при помощи ввода команды SEND DATA	Должно появиться приглашение DATA TRANSFER INITIATED и следующими двумя строками Enter your credentials… Login:
3	Ввести имя учетной записи default	Должна появиться строка Password:
4	Ввести пароль default	Должно появиться сообщение Default user blocked – system set

№	Шаг сценария	Ожидаемый результат
		to High security и соединение с терминалом должно быть прервано

Как можно видеть, такая форма представления действительно неудобна для автоматизации тестирования и предназначена исключительно для ручного тестирования. Иногда такие тест-планы совмещают с отчетами о проведении тестирования, добавляя в таблицу описания сценария третью и четвертые колонки – «Реальный результат» и «Соответствует», в который заносятся реальная реакция системы и указание на совпадение/несовпадение результатов соответственно. В конце описания каждого тестового примера добавляется графа «Пройден/не пройден», в которую заносится информация о том, пройден ли тестовый пример в целом. В конце всего тест-плана совмещенного с отчетом помещается графа «Тестовых примеров пройдено/всего», в которую заносится число пройденных тестовых примеров и общее их число.

Сценарии тестирования для автоматического тестирования часто описывают на том или ином языке программирования. Например, методы в тестирующих классах Microsoft Visual Studio Team Edition представляют собой именно пошаговые описания действий, которые необходимо выполнить тестовому окружению для проведения тестирования. Возможна и более близкая к естественному языку форма подготовки тестовых примеров. Например, при тестировании логической функции с уровнем покрытия MC/DC и описании тестовых примеров на одном из диалектов Visual Basic Script возможно записать сценарий тест-плана в такой форме:

```
'------------------------------------------------------------

' TEST CASES

'------------------------------------------------------------

' 8 testcases

' 1 2 3 4 5 6 7 8

' ---------------------------------------------

' computed - - 0 0 0 - - -

' good1 0 1 0 0 0 0 0 0

' computed2 - - - - 0 - - -
```

```
' good2 1 1 1 0 0 1 1 1

' delay - - - - - 0 - -

' pack1 1 1 1 1 1 1 0 0

' pack2 0 0 0 0 0 0 0 1

' ----------------------------------------------

' output_message 1 0 0 1 0 0 0 1

'------------------------------------------------------------------

' Testcase #1:

Call Test_Message_Call (-, 0, -, 1, -, 1, 0, 1)

'------------------------------------------------------------------

' Testcase #2:

Call Test_Message_Call (-, 1, -, 1, -, 1, 0, 0)

'------------------------------------------------------------------

' Testcase #2:

Call Test_Message_Call (0, 0, -, 1, -, 1, 0, 0)

'------------------------------------------------------------------' Testcase #4:

Call Test_Message_Call (0, 0, -, 0, -, 1, 0, 1)

'------------------------------------------------------------------' Testcase #5:

Call Test_Message_Call (0, 0, 0, 0, -, 1, 0, 0)

'------------------------------------------------------------------' Testcase #6:

Call Test_Message_Call (-, 0, -, 1, 0, 1, 0, 0)

'------------------------------------------------------------------' Testcase #7:

Call Test_Message_Call (-, 0, -, 1, -, 0, 0, 0)

'------------------------------------------------------------------' Testcase #8:
```

Call Test_Message_Call (-, 0, -, 1, -, 0, 1, 1)

При такой форме представления сценарий каждого тестового примера состоит из последовательности вызовов функций (в данном случае функция всего одна), которые передают данные в среду тестирования.

4.4.4. Таблицы

Как уже говорилось выше, табличное представление тестов удобно при четко формализованных входных и выходных потоках данных системы. Например, в предыдущем фрагменте тест-плана в комментариях приведена таблица, в которой по вертикали указаны имена входных потоков данных системы, по горизонтали приведены номера тестовых примеров, а в ячейках на их пересечении приведены значения. Выходные значения приводятся в том же формате ниже:

' 1 2 3 4 5 6 7 8

' ---

' computed - - 0 0 0 - - -

' good1 0 1 0 0 0 0 0 0

' computed2 - - - - 0 - - -

' good2 1 1 1 0 0 1 1 1

' delay - - - - - 0 - -

' pack1 1 1 1 1 1 1 0 0

' pack2 0 0 0 0 0 0 0 1

' ---

' output_message 1 0 0 1 0 0 0 1

Табличное представление, как правило, используется для упрощения работы по подготовке и сопровождению большого количества однотипных тестов. Среда тестирования, использующая табличное описание тестовых примеров в качестве входных данных включает в себя интерпретатор таблиц, преобразующих это описание в последовательность команд, выполняемых средой для проведения тестирования, т.е. своего рода сценарий.

В случае, когда однотипными являются не только входные и выходные данные, но и их значения, может использоваться альтернативная форма

представления табличных данных. Тестовые примеры в ней также нумеруются по горизонтали, а входные потоки данных – по вертикали. Однако, под каждым из потоков данных перечисляются возможные входные значения, а факт того, что это входное значение должно быть передано в данном тестовом примере, отмечается помещением специальной метки (например, символа X) на пересечении значения и тестового примера в таблице:

+-------------+

INPUTS: | a b c d e f |

---------------------------+-------------+

Power_On_Mode |

COLD | X X X

WARM | X X X

Configuration_Store_Id |

0xFFFD | X X X X X X

IR_Access_Mode |

1 | X X X X

0 | X

0xFFFF | X

Reset_Mode |

0 | X X X X X X

Reset_Source |

0 | X X

1 | X

2 | X X X

При интерпретации каждого такого тестового примера он преобразуется в последовательность команд, которые выполняются средой тестирования, например для тестового примера a:

Power_On_Mode = COLD

Configuration_Store_Id = 0xFFFD

IR_Access_Mode = 1

Reset_Mode = 0

Reset_Source = 1

Run_Test()

Последняя команда здесь запускает тест на выполнение с установленными входными данными.

4.4.5. Конечные автоматы

Форма подготововки тест-планов в виде описания конечных автоматов удобна при тестировании программных модулей или систем, поведение которых также может быть описано в виде конечного автомата. В этом случае процесс тестирования представляет собой обмен сообщениями между двумя конечными автоматами, изменяющими свое состояние в процессе обмена. Критерием полноты такого тестирования будет достижимость всех состояний тестируемой системы всеми возможными способами.

Описание тест-планов в виде конечного автомата обычно состоит из двух частей – определения самого тестирующего конечного автомата и определения сценариев перехода между состояниями – тестовых примеров.

Рассмотрим такой тест-план на следующем примере. Пусть тестируемый модуль представляет собой простой конечный автомат с тремя состояниями - «Начальное», «Прием данных» и «Ошибка». Автомат начинает свою работу в начальном состоянии, из которого может быть переведен в состояние «Прием данных» по получению сообщения «Начало данных». Он может переходить из этого состояния в него же по получению каждого следующего правильного блока данных, в состояние «Ошибка» по получению неверного блока данных или в начальное состояние по получению сообщения «Конец данных». При переходе в состояние «Ошибка» он передает сообщение «Возникла ошибка». Из состояния «Ошибка» он может переходить в начальное состояние по получению сообщения «Ошибка обработана». Структурная схема такого автомата показана на Рис. 17.

Рис. 17 Структурная схема тестируемого конечного автомата

Тестирующий конечный автомат должен уметь посылать все воспринимаемые тестируемым автоматом сообщения и воспринимать все посылаемые им сообщения. При этом целью тестирования будет проведение тестируемого автомата по всем состояниям всеми возможными способами. Один из возможных вариантов построения тестирующего автомата заключается в построении автомата с эквивалентными состояниями. Управление таким автоматом в-основном будет проводиться при помощи описаний тестовых примеров, а не при помощи сообщений извне.

Так, такой тестирующий автомат будет иметь три состояния – «Начальное», «Передача данных» и «Обработка ошибки». При переходе из начального состояния в состояние «Передача данных» он передает сообщение «Начало данных», в состоянии «Передача данных» он будет передавать блоки данных, описанные в тестовом примере, в т.ч. возможно, ошибочные. При получении сообщения «Возникла ошибка» автомат перейдет в состояние «Обработка ошибки» из которого перейдет в начальное состояние передав сообщение «Ошибка обработана». В начальное состояние тестирующий автомат может перейти и в случае завершения последовательности блоков данных, описанных в тестовом примере, в этом случае при переходе он пошлет сообщение «Конец данных». Структурная схема такого автомата показана на Рис. 18.

Рис. 18 Структурная схема тестирующего конечного автомата

Далее приведен пример определения этого тестирующего автомата в тест-плане. Оно будет выглядеть следующим образом:

STATES DEFINITION:

State1=Начальное

State2=Передача данных

State3=Обработка ошибки

PASS DEFINITION

Pass1=State1->State2 with function call BeginData(Param1)

Pass2=State2->State2 with function call SendData(Param1)

….

Pass5=State2->State3 external with function call ErrorReceived(Message)

В разделе STATES DEFINITON определены все состояния тестирующего автомата, в разделе PASS DEFINITION – переходы между состояниями. Переход из состояния M в состояние N определяется выражением StateN->StateM. При переходе вызывается функция тестового драйвера, имя которой записывается после строки with function call. Если в функцию должны быть переданы параметры, их имена указываются в скобках. Если какой-либо переход должен происходить при получении внешнего сообщения, это обозначается ключевым словом external. При этом вызывается функция, обрабатывающая полученное сообщение

Тестовые примеры для тестирования конечного автомата будут выглядеть следующим образом:

TESTCASE 1

Data:

begBlock=\027

sndBlock[0]='H'

sndBlock[1]='i'

errBlock=0

Scenario:

Pass1(begBlock)

Pass2(sndBlock[0])

Pass2(sndBlock[1])

Pass2(errBlock)

Pass5(message)

В этом примере в секции Data определяются данные для сообщений, передаваемых автоматом, а в секции Scenario – последовательность переходов по состояниям с передаваемыми данными.

При тестировании конечных автоматов при помощи обычных тестирующих классов можно использовать аналогичный подход.

4.4.6. Генераторы тестов

В некоторых случаях для упрощения процедуры тестирования используются специальные инструментальные средства, автоматически генерирующие тестовые примеры. Эти системы различаются по используемым методам генерации тестовых примеров, а получаемые тестовые примеры различаются по областям применимости.

Различают следующие способы генерации тестовых примеров:

- по формализованным требованиям;
- случайным образом;
- по программному коду.

Первый способ генерации тестовых примеров приемлем для тестирования системы как «черного ящика», но требует чтобы тест-требования (или системные/функциональные требования) были подготовлены на специальном формальном языке оформления требований, например RDL (Requirements Definition Language). Затем по требованиям строятся тестовые примеры, которые проверяют функциональность системы с точки зрения требований, т.е. в этом случае достигается основная цель верификации – проверить, ведет ли себя система в соответствии с требованиями.

К сожалению, этот путь достаточно трудоемок и экономия времени от автоматической генерации тестов зачастую сводится на нет необходимостью в выделении дополнительного времени на перевод всех требований в формальную форму. В связи с этим рекомендуется применять данный метод только для тестирования систем, требования на которые могут быть сравнительно легко формализованы с использованием того или иного языка, например, системы поддержки коммуникационных протоколов.

Второй метод генерации тестовых примеров – на основе случайных данных. В этом случае не может идти и речи о систематизированном тестировании и гарантиях качества системы. Такой подход может применяться только в случае необходимости проверить поведение системы в случае передачи в нее большого количества неверных данных или определить количественные параметры поведения системы под большой нагрузкой.

Третий метод тестирования основан на анализе исходных текстов системы и построения тестов, которые выполняют каждое логическое условие и каждый оператор системы. В результате достигается очень высокий уровень покрытия программного кода. Однако, в этом случае тесты проверяют не то, что система должна делать в соответствии с требованиями, а то, как она делает то, что уже запрограммировано. Перед тестировщиком в этом случае стоит задача анализа программного кода системы на соответствие требованиям, что зачастую представляет собой задачу не менее сложную, чем ручное написание тестов для проверки требований. Обычно рекомендуется вначале написать все тесты по требованиям, а затем, в случае необходимости, воспользоваться генератором тестов по программному коду. При этом целью использования генератора будет не достижение максимально возможного покрытия любой ценой, а анализ причин непокрытия при выполнении тестов требований, и, в случае необходимости, коррекции требований.

4.5. Отчеты о прохождении тестов

4.5.1. Технологические цепочки и роли участников проекта, использующих отчеты о прохождении тестов. Связь отчетов о прохождении тестов с другими типами проектной документации.

Отчеты о прохождении тестов – основной (а иногда единственный) источник для заключения о соответствии протестированной системы требованиям. После выполнения всех тестов, описанных в тест-планах, среда тестирования создает отчет о том, насколько успешно система выполнила эти тесты. Такой отчет как минимум содержит информацию о каждом выполненном тестовом примере (его идентификатор) и результат выполнения его выполнения – успех или неудачу.

По результатам анализа отчетов о прохождении тестов могут быть выявлены либо дефекты в самой системе, либо некорректно составленные или противоречивые требования. В обоих случаях результаты анализа служат основой для создания запросов на изменение требований и/или кода системы. После корректного исправления дефектов при регрессионном тестировании неуспешно выполненные тестовые примеры должны выполниться успешно (Рис. 19).

Рис. 19 Генерация отчета о прохождении тестов и изменения по результатам его анализа

Отчеты о прохождении тестов могут служить основой для отслеживания состояния проекта – если с течением времени количество обнаруживаемых дефектов (неуспешно выполненных тестовых примеров) падает при условии сохранения качества тестирования – это свидетельствует о повышении качества разрабатываемой системы. С другой стороны, при внесении значительных изменений в систему, количество дефектов неизбежно возрастает. Таким образом, идеальный график зависимости количества дефектов от времени похож на синусоиду с уменьшающейся амплитудой на каждом полупериоде.

4.5.2. Возможные формы представления отчетов о прохождении тестов

В разделе 2.6 уже приводилось несколько примеров отчетов о выполнении тестовых примеров, однако в этом разделе основной уклон делался в сторону общей статистики выполнения тестов.

В стандарте IEEE 829 отчет о прохождении тестов разделен на три различных документа и описан в разделах 9 (Test log), 10 (Test incident report) и 11 (Test summary report). В эти разделы включены соответственно общий отчет о прохождении тестов, отчет о проблемах, выявленных в результате выполнения тестов и общую статистику прохождения тестов. В данном курсе отчет о прохождении тестов считается единым документом, разделенным на три части:

- общая (заголовочная информация);
- результаты выполнения тестовых примеров (положительные и отрицательные);
- итоговая информация о выполнении тестовых примеров (общая статистика по выполненным тестам).

Заголовочная часть отчета о прохождении тестов служит для идентификации отчета и протоколирования того, какая часть разрабатываемой системы подвергалась тестированию, какая ее версия, какая конфигурация тестового стенда использовалась для выполнения тестов.

В заголовочную часть отчета о выполнении тестов обычно включается следующая информация:

1. Название проекта или тестируемой системы
2. Общий идентификатор группы тестовых примеров, включенных в отчет
3. Идентификатор тестируемого модуля или группы модулей и номера их версий
4. Ссылку на разделы и версии тест-требований или функциональных требований, по которым написаны тесты, для которых сгенерирован отчет
5. Время начала выполнения теста и его продолжительность
6. Конфигурацию тестового стенда, на которой выполнялся тест
7. Имена и фамилии автора тестов и/или лица, выполнявшего тесты.

Ниже показаны два примера таких заголовочных частей отчета, создаваемых различными инструментальными средствами. Красными цифрами в скобках обозначены соответствующие пункты приведенного выше списка.

```
**************************************************

** Document Test Environment
```

** User's Computer: COMPUTER_185 **(6)**

** Testing Host Application: FacilityTest **(6)**

** Testing Host Version: 5.12 **(6)**

**

************* Server Related Data *************

** Server Computer: SERVER_105 **(6)**

** Server Version: 6.24.0 (Build 16) **(6)**

** Configuration: Control remote bench **(6)**

** Mode: Realtime **(6)**

** Test executed on: 7/29/06; at 10:09:40 AM **(5)**

** Tester Name is [Sidorov A.] **(7)**

** Software Version is: CNTRL 115 01 5 **(1)**

** Test Station being used is: COMPUTER_185 **(6)**

****

===
========= REMOTE CONTROL FUNCTION SOFTWARE TEST REPORT

===
=========Project Name : Facility Remote Control **(1)**

Function Name : Infrared Transmitter Signal Handler **(3)**

Test Name : IRDA_C05A_1091K **(2)**

Document Name : SSRD for the Remote Control Function **(4)**

Paragraph Name : Button Signals **(4)**

Primary Paragraph Tag : [PTAG::SSRD IR BTN SIGNALS] **(4)**

Template Class : Test

Shall Tag(s) : SSRD IR BTN SIGNALS 10 **(4)**

```
Shall(s) template : Test

------------------------------------------------------------------- MODIFICATION HISTORY:

-------------------------------------------------------------------

Ver Date Author Change Description CR No.

--- --------- ---------- ----------------------- --------

01 19 Jul 06 Ivanov K. (7)Initial Development. CR_10
```

```
=====================================================================
```

```
; SIMULATION RESULTS FILE

; Matrix Compiler CORE VERSION 3.00

; TEST PLAN

; ELEMENT: IRDA_IA.TMC (2)

; TITLE: Test Plan for Inrfrared source files test (1)

; TEST DATE/TIME Wed 02.11.2005 23:12:53 (5)

; SYS section: 2.3.5.6 Version: 24 (4)

; SRD section: 6.3 Version: 12 (4)

; SDD section: 12.3 Version: 33 (4)

; SOURCE FILE(S): IRDA.C Version: 18 (4)

; IRDA.H Version: 2 (4)

;

; SIMULATOR SETUP: (6)

; MODE HIGH (6)

; INC C (6)

Следующая часть отчета о прохождении тестов должна содержать информацию о результате выполнения каждого тестового примера –
```

завершился ли он успешно или в результате его выполнения были выявлены какие-либо несоответствия с ожидаемым результатом. В некоторых проектах эта часть отчета может быть представлена в одной из двух форм – полной или краткой. Полная форма содержит всю информацию о тестовом примере, краткая – только информацию об обнаруженных в результате выполнения тестового примера несоответствий ожидаемых и реальных выходных значений.

Обычно каждая запись о результате прохождения каждого тестового примера в полной форме содержит следующую информацию:

1. Идентификатор тестового примера
2. Краткое описание тестового примера
3. Перечисление всех входных значений тестового примера
4. Перечисление всех ожидаемых и реальных выходных значений тестового примера
5. Для каждой пары «ожидаемое-реальное выходное значение» - информацию о совпадении/несовпадении этих значений
6. Сообщение о том, пройден или не пройден тестовый пример.

В краткой форме каждая запись обычно содержит следующую информацию:

1. Идентификатор тестового примера
2. Перечисление не совпавших ожидаемых и реальных выходных значений тестового примера
3. Для каждой пары «ожидаемое-реальное выходное значение» - информацию о совпадении/несовпадении этих значений
4. Сообщение о том, пройден или не пройден тестовый пример.

Ниже приведено два примера информации о прохождении тестового примера в краткой и полной формах соответственно. Красными цифрами в скобках отмечены соответствующие пункты приведенных выше списка для краткой и полной форм соответственно.

[Testcase 163] **(1)** :True: <EQ> :True: **(4)** ** Passed Number 163 **

[Testcase 164] :True: <EQ> :True: ** Passed Number 164 **

[Testcase 165] :True: <EQ> :True: ** Passed Number 165 **

[Testcase 166] :False: <EQ> :True: **(4)** ** Fail Number 1 **

*** Inputs for Testcase 166

DisplayTextLine2.ItemChecked = 2 (2 expected)

```
     DisplayTextLine2.ItemChecked = 2 (2 expected)

     *** Outputs for Testcase 166

     DisplayTextLine2.ItemChecked = 2 (2 expected) (2)

 (3) --- DisplayTextLine2.ItemChecked = 2 (1 expected)

     DisplayTextLine9.ItemChecked = 2 (2 expected)

; 1) Test group 1, case a. (1)

; Test case verifies that infrared watchdog is activated by

; startup pulse sequence (2)

; Test requirements section 6.4.3.1.2

CASE DEFAULTS : (3)

T_FL_Sys_Fail_Called = 0

T_Update_Time = 1828ACh

T_CMT_Menu_Last_Update = 18639Ch

T_Level_1_Status = 180004h

T_Level_2_Status = 180304h

T_Stop_Method = 0

T_Fault_Report = 1

     INPUTS : (3)

num_iterations = 1

entry_procedure = 1

T_NV_Power_On_Count = 1

T_Reset_Value = 0

T_Time_Since_Power_On = 1

OUTPUTS: EXPECTED (4) ACTUAL (4) RESULT (5)
```

T_NV_Power_On_Count = 1 1 PASS

T_NV_Power_On_Count_Check = 65533 65533 PASS

T_BBRAM_Power_On_Count = 1 1 PASS

T_Time_Since_Power_On = 100 100 PASS

T_FH_Queue_Msg_Count = 2 2 PASS

T_Pulse[0].Data[0] = 0 0 PASS

T_Pulse[0].Data[1] = 0 10 FAIL

Test case FAILED **(6)**

Завершающая часть отчета о прохождении тестов должна содержать краткую итоговую информацию о выполнении всех тестовых примеров, по которым составлялся отчет.

Обычно эта часть отчета содержит следующую информацию:

1. Общее количество выполненных тестовых примеров
2. Количество успешно пройденных тестовых примеров
3. Количество неуспешно пройденных тестовых примеров
4. Общее количество проверенных выходных значений
5. Количество выходных значений, у которых ожидаемое значение не совпало с реальным.

Ниже приведен пример этой части отчета.

TEST RESULTS:

No. of Test Cases Failed : 0 **(3)**

No. of Test Cases Passed : 45 **(2)**

Total No. of Tests Included : 46 **(1)**

Total No. of Outputs Checked : 2783 **(4)**

No. of failed Outputs Checks : 128 **(5)**

Часто в отчет о выполнении тестов кроме количественной статистики помещают раздел с подробным объяснением причин неуспешно пройденных тестовых примеров. Каждый пункт такого объяснения обычно содержит следующую информацию:

1. Идентификаторы тестовых примеров, благодаря неуспешному выполнению которых выявлена проблема
2. Ссылка на разделы требований, по которым написаны тестовые примеры
3. Ссылка на участки программного кода в котором выявлена проблема
4. Описание сути проблемы и (опционально) возможные пути ее решения с точки зрения тестировщика.

Данный раздел может служить основой для создания отчетов о проблемах, либо частично заменять их.

Пример такого раздела приведен ниже:

TEST CASES WITH FAILURES SUMMARY

===================================

testcases | failures total | explanation in section

------------------+----------------+-----------------------

(1) 11b-l,n-p; 12b-d | 67 | 1

------------------+----------------+-----------------------

total | 67 |

1.

LOCATION:

PR_IR_DATA.C, lines 1323, 1347; **(3)**

Software requirements section 7.4.5.5; **(2)**

Test requirements section 7.4.8; **(2)**

PROBLEM:

Test requirements are not changed, but Software requirements are updated to reflect new system functionality. **(4)**

DEMONSTRATION:

Test cases: 11 b-l, n-p; 12 b-d **(1)**

PROPOSED SOLUTION:

Update test requirements section 7.4.8 to meet software requirements section 7.4.5.5. **(4)**

4.5.3. Автоматическое и ручное тестирование

Некоторые тестовые примеры не могут быть выполнены в автоматическом режиме и поэтому требуют ручной работы тестировщика по их выполнению. Результаты выполнения ручных тестовых примеров могут заноситься в тот же самый документ, что и результаты выполнения автоматических тестовых примеров. Особенно часто это делается в случае, если и автоматические и ручные тесты проверяют одну и ту же функциональную часть тестируемой системы. В этом случае при генерации отчета о прохождении тестов для ручных тестов генерируется форма, в которую тестировщик заносит данные о результатах проведенного им ручного тестирования. Само ручное тестирование может заключаться либо в выполнении тестового сценария, заданного в тест-плане, либо в экспертном анализе участков программного кода системы, которые не могут быть выполнены при автоматическом тестировании на тестовом стенде. Форма для ручного тестирования обычно содержит следующую информацию:

1. идентификатор ручного тестового примера
2. описание сценария ручного тестового примера или задачи экспертного анализа
3. имя лица, проводившего ручное тестирование
4. версии требований, на основании которых проводилось ручное тестирование
5. Ссылки на участки программного кода, для которого проводится ручное тестирование
6. Информацию о соответствии программного кода требованиям (результат ручного тестирования) – соответствует/не соответствует
7. Информацию о потенциально возможных проблемах внутри допустимого диапазона значений и за его пределами
8. Информацию о возможности покрытия тестируемого вручную программного кода при достижении условий, указанных в требованиях
9. Информацию об итоговом результате ручного тестового примера – успешно/неуспешно.

Ниже приведен пример заполненной формы для ручного тестирования. Красными цифрами в скобках выделены соответствующие пункты приведенного выше списка, зеленым выделен текст, вводимый в форму тестировщиком:

****

Manual Analysis of Testcase 1 **(1)**: verify that the IR scan goes at a 10Hz rate. **(2)**

****

1. Activity Description: Independent Code Analysis.

Tester Name: Petrov P. **(3)**

Pass Criteria: Tester name is not the same as programmer name

2. Activity Description: Name and CM Version of file under review

Module and/or Function Name: IRDA.C CM Version: 56 **(4,5)**

Document chapter: 7.8.5 CM Version: 12 **(4)**

Pass criteria: All documents exists in respective versions

3. Activity Description: Requirements Implemented.

Identify which lines of code in the module under review incorporate requirements

PassCriteria: Code implements the requirements as defined in the requirement document

Result (Pass/Fail): Pass **(6)**

4. Activity Description: Code Robustness

Identify line numbers and give a brief description on the software design to handle the following cases

Analysis for at, Inside boundary : The variable refreshRate is set to 10 at line 235 of IRDA.C and later used in IR_Init() function to set NVRAM values on line 590. **(5,7)**

Analysis for out-of-boundary (robustness) : N/A **(7)**

5. Activity Description: Structural Coverage Analysis

PassCriteria: Software and software structures (when applicable) are accessible under conditions specified by requirements

Y/N: YES **(8)**

6. Activity Description: Overall manual test result

Result (Pass/Fail): Pass **(9)**

4.6. Отчеты о покрытии программного кода

4.6.1. Технологические цепочки и роли участников проекта, использующих отчеты о покрытии. Связь отчетов о покрытии с другими типами проектной документации.

Степень покрытия программного кода тестами – важный количественный показатель, позволяющий оценить качество системы тестов, а в некоторых случаях – и качество тестируемой программной системы. Данные о степени покрытия помещаются в отчеты о покрытии, генерируемые при выполнении тестов инструментальными средствами, поддерживающими процесс тестирования, т.е. по сути, генерируются средой тестирования (Рис. 20). Формат отчетов о покрытии обычно един внутри проекта или нескольких проектов и часто зависит от особенностей инструментальных средств тестирования.

В отчете о покрытии в стандартизированной форме указываются участки программного кода тестируемой системы (или ее части), которые не были выполнены во время выполнения тестовых примеров, т.е. не были покрыты тестами. Причины непокрытия анализируются тестировщиками, по результатам анализа составляются отчеты о проблемах и запросы на изменение – документы, в которых описывается объекты разработки, которые необходимо изменить и причины этих изменений.

Рис. 20 Генерация отчета о покрытии и изменения по результатам его анализа

Недостаточное покрытие может свидетельствовать о неполноте системы тестов или тест-требований, в этом случае в запросе об изменении указывается на необходимость расширения системы тестов или тест-требований. Другой причиной недостаточного покрытия могут быть участки защитного кода, которые никогда не выполнятся даже в случае нештатной

работы системы. В этом случае в запросе на изменении указывается на необходимость модификации исходных текстов, либо отмечается, что для этого участка программной системы не требуется покрытие. В качестве третьей причины недостаточного покрытия может выступать рассогласование требований и программного кода системы, в результате которого в коде могут остаться неиспользуемые более участки, либо наоборот, появиться участки, рассчитанные на будущее (и реализующие функциональность, не описанную в требованиях). В этом случае в запросе на изменение указывается на необходимость модификации требований и/или кода системы для приведения их в согласованное состояние.

4.6.2. Возможные формы отчетов о покрытии

Типичный отчет о покрытии представляет собой список структурных элементов покрываемого программного кода (функций или методов), содержащий для каждого структурного элемента следующую информацию:

- Название функции или метода
- Тип покрытия (по строкам, по ветвям, MC/DC или иной)
- Количество покрываемых элементов в функции или методе (строк, ветвей, логических условий)
- Степень покрытия функции или метода (в процентах или в абсолютном выражении)
- Список непокрытых элементов (в виде участков непокрытого программного кода с номерами строк)

Кроме того, отчет о покрытии содержит заголовочную информацию, позволяющую идентифицировать отчет и общий итог – общую степень покрытия всех функций, для которых собирается информация о покрытии.

Пример такого отчета о покрытии приведен ниже

Coverage Report

Generated 10/07/2006 for file Testing_Facilities.cpp

----

1) function main_Menu()

Coverage: Instructions

Elements: 25 structured lines of code (SLOCs)

Covered: 22 lines (88%)

Not covered:

291 default:

292 return -1;

293 break;

Coverage: Branches

Elements: 5 branches

Covered: 4 branches (80%)

Not covered (starting and ending lines only):

default:

break;

----

2) function item_Help()

Coverage: Instructions

Elements: 180 structured lines of code (SLOCs)

Covered: 180 lines (100%)

Coverage: Branches

Elements: 2 branches

Covered: 2 branches (80%)

Total functions: 2

Total instructions coverage: 98.5%

Total branches coverage: 86%

Отчет о покрытии может создаваться либо для всех функций программного модуля или всего проекта, либо выборочно для определенных функций.

В случае, если размер функций, для которых генерируется выборочный отчет, невелик, может применяться другая форма отчета о покрытии, в котором покрытый и непокрытый программный код выделяется различными цветами. Такая форма неприменима для покрытия ветвей и логических условий, но может применяться для покрытия по строкам.

Пример такого отчета приведен ниже:

Coverage report for BaseCalculator.AnalaizerClass.Format method.

Generated on 25/07/2006

```
public static string Format()

{

string formstr = "";

string prev = "";

if (expression.Length <= 65536) {

for (int i = 0; i < expression.Length; i++) {

switch (expression[i]) {

case '0': {

if (prev == "число" || prev == "") {

formstr += expression[i].ToString();

} else {

formstr += " " + expression[i].ToString();

}

prev = "число";

break;

}

}

}
```

```
} else {

MessageBox.Show("Слишком длинное выражение.");

Program.res = 7;

return "&Error 07";

}

}
```

Зеленым цветом отмечены выполненные в результате тестирования участки метода, красным – не выполненные.

Конкретная форма отчета о покрытии определяется инструментарием и технологическими процессами проекта.

Рассмотрим, например, отчет о структурном покрытии, генерируемый полученного при помощи средства для анализа программного кода CodeTEST компании Metrowerks.

Сбор покрытия производится по определенному уровню покрытия, информация об этом входит в состав заголовка отчета о покрытии. В данном случае програмный код покрывался по MC/DC.

CodeTEST Advanced Coverage Tools

Modified Condition Decision Coverage Report (MCDC), DO-178B Level A

CodeTEST Report Generator 2.2.04

Report generated on Sun Aug 21 14:28:16 2005

IDB File:

<Path_to_IDB_file>

42963330-1C0 Thu May 26 21:36:00 2005

Далее выводится информация о проценте покрытия по всем модулям в совокупности (в данном случае, это 33 модуля):

Overall Summary (MCDC): 33 Files

1405 true-false decisions

674 48.0% covered

494 35.2% partially covered

237 16.9% not covered

1928 conditions in the decisions

1068 55.4% covered

571 29.6% partially covered

289 15.0% not covered

278 case branches

154 55.4% covered

124 44.6% not covered

581 coverage events

504 86.7% covered

77 13.3% not covered

Рассмотрим более подробно каждую из характеристик:

true-false decisions – количество логических выражений, в данном случае их 1405. Из этого числа на оба возможных значения (True и False) покрыто лишь 674, еще 494 покрыто на какое-то одно из значений, а 237 не покрыто вообще.

conditions in the decisions – количество логических условий внутри логических выражений. Как можно догадаться, их количество не может быть меньше количества логических выражений. Непокрытие логических условий явным образом влечет за собой непокрытие логических выражений. Факт, что если все условия будут покрыты, то все выражения также будут покрыты.

case branches – количество веток ветвления операторов выбора (**select**). Ветка считается покрытой, если выражение, стоящее в условии оператора select, принимает значение, соответствующее данному варианту выбора (**case**), в результате чего ветка выполняется.

coverage events – в данном случае, это количество точек входа / выхода в функциях. Точка входа в функцию считается покрытой, если эта функция

была вызвана хотя бы один раз. Точками выхода являются точка окончания функции (}), когда управление передается в место вызова этой функции, а также операторы возврата (**return**), после выполнения которых управление также передается в место вызова функции.

Затем выводится информация для первого модуля, а именно: имя модуля, путь к файлу на диске, дата последнего изменения, контрольная сумма. После этого выводится суммарная информация о покрытии только для этого модуля (для всех его функций, количество которых также указывается). Структура этой информации аналогична структуре, описанной выше:

1. File <Module_1_name.cpp> in <Path_to_module_1>

Last Modified: Fri Oct 15 18:34:14 2004

Checksum: CEDBAB67

File Summary (Extended MCDC): 6 Functions

24 true-false decisions

10 41.7% covered

11 45.8% partially covered

3 12.5% not covered

39 conditions in the decisions

20 51.3% covered

15 38.5% partially covered

4 10.3% not covered

0 case branches

12 coverage events

12 100.0% covered

0 0.0% not covered

После этого происходит поочередный вывод суммарной информации для каждой из функций в отдельности, а также всех ключевых участков функции, влияющих на покрытие, а именно: точки входа / выхода, логические выражения, логические условия, операторы выбора. При этом для каждого из

этих ключевых участков выводится информация о степени покрытия, где возможными значениями могут быть:

COVERED – участок полностью покрыт в понятиях типа, к которому он приписан;

PARTIALLY covered – участок частично покрыт. Применимо только к логическим выражениям и условиям и указывает на то, что выражение (условие) покрыто на какое-то одно из двух возможных значений;

NOT covered – участок не покрыт. Это означает, что данный участок не выполнялся в процессе работы скомпилированного программного кода.

Ниже представлен типичный результат покрытия для функции:

1.1 Function <Function_1>

void Function_1(void)

Function Summary (MCDC):

3 true-false decisions

0 0.0% covered

2 66.7% partially covered

1 33.3% not covered

5 conditions in the decisions

1 20.0% covered

3 60.0% partially covered

1 20.0% not covered

0 case branches

2 coverage events

2 100.0% covered

0 0.0% not covered

coverage line 266: function entry

COVERED

decision line 267: if statement

if ((A == B) && (...

T F

1: *ttt *fxx ((A == ...)&& COVERED

2: *ttt tfx ((C...)&& PARTIALLY covered

3: *ttt ttf ((D == 0) PARTIALLY covered

decision line 271: if statement

if (E <= 0)

T F

1: t *f (E <= 0) PARTIALLY covered

decision line 276: if statement

if (D == 0)

T F

1: t f (D...) NOT covered

coverage line 291: function end

}

COVERED

Следует обратить внимание на то, каким образом происходит разбор логических выражений. Предположим, что у нас имеется выражение вида:

if ((A==B) && (C > 0) && (D == 0))

и пусть результат покрытия для него такой, как приведен выше (decision line 267: if statement). Для этого выражения строится таблица, где в каждой строке помещается информация для каждого из логических условий, входящих в это логическое выражение. Нумерация логических условий начинается с 1, в нашем случае их 3.

В первом столбце указывается комбинация значений логических условий (с участием рассматриваемого условия), которую необходимо выставить, чтобы логическое выражение приняло значение True. В данном случае, так как все условия соединены логическим оператором «И» (&&), то такой комбинацией может быть только TTT. Если указанная комбинация была сформирована в процессе тестирования программы, то она помечается символом (*).

Во втором столбце указывается комбинация значений логических условий (с участием рассматриваемого условия), которую необходимо выставить, чтобы логическое выражение приняло значение False. В данном случае достаточно, чтобы хотя бы одно условие приняло значение False, при этом значения всех последующих условий не учитываются (это отображается символом X).

Третий столбец – это вырезка текста логического условия, позволяющая легче ориентироваться в коде при анализе покрытия.

Из получившейся таблицы видно, что второе и третье логическое условие в процессе выполнения не принимали значения False. Определение причины этого и является задачей анализа структурного покрытия.

После того, как в рамках данного модуля будут рассмотрены все функции, рассматривается следующий модуль, и так далее. Как видно, структура отчетного файла, получаемого в результате работы CodeTEST, достаточно проста и имеет вложенную структуру.

Отчеты о покрытиях, создаваемые Microsoft Visual Studio Team Edition будут рассмотрены на семинарских занятиях.

4.6.3. Покрытие на уровне исходных текстов и на уровне машинных кодов

В некоторых случаях инструментальные средства сбора покрытия анализируют покрытие программного кода тестами не на уровне исходных текстов системы, а на уровне машинных инструкций. В этом случае степень покрытия зависит и от того, какой исполняемый код генерируется компилятором. Отчеты о покрытии в этом случае выглядят следующим образом (отчет о покрытии сгененирован при помощи отладчика Microtec XRAY для Motorola 680x0):

******* INSTRUCTION EXECUTION COVERAGE - Analyze Unit: Menu_Display

Function: DATA_PROCESSING\Menu_Display. Instruction(s) not executed:

5668 switch (Key)

```
0004A0F6 0352 BCHG D1,(A2)

0004A102 0466 0466 SUBI.W #$466,-(A6)

0004A106 0466 0466 SUBI.W #$466,-(A6)

0004A10A 0466 0466 SUBI.W #$466,-(A6)

0004A10E 0466 0466 SUBI.W #$466,-(A6)

0004A112 0466 0466 SUBI.W #$466,-(A6)

0004A116 0466 0474 SUBI.W #$474,-(A6)

5751 i = 1;

0004A4A6 7401 MOVEQ #$1,D2

Executed Total Percent Analyze-unit

467 475 98.32 DATA_PROCESSING\Menu_Display

0 unit(s) excluded.

******* BRANCH EXECUTION COVERAGE - Analyze Unit: Menu_Display

Function: DATA_PROCESSING\Menu_Display. Branch(es) not executed:

5612 if ( First_Call == TRUE ) { /* Only execute this block during th

00049E3A 6600 0722 BNE.W $4A55E Branch not taken.

5613 if ( MP_Rd_Config_Straps ( RAW_STRAPS, &Strap_Config ) ==

00049E52 664A BNE.B $49E9E Branch not taken.

5750 if (i == 17)

0004A4A4 6602 BNE.B $4A4A8 Fall thru not taken.

Executed Total Percent Analyze-unit

74 77 96.10 DATA_PROCESSING\Menu_Display

0 unit(s) excluded.
```

Поскольку степень покрытия может меняться в зависимости от оптимизации при генерации кода, в некоторых случаях даже при полном выполнении всех

операторов языка высокого уровня на котором написана программная система, не удается достичь полного покрытия на уровне исполняемого кода.

Например, в случае покрытия следующей конструкции на языке C:

```
typedef enum

{

CC1 = 250,

CC2 = 251,

CC3 = 252

} E_CC;

…

E_CC key;

    …

switch (key) {

case CC1: printf("CC1"); break;

case CC2: printf("CC2"); break;

case CC3: printf("CC3"); break;

}
```

компилятор Microtec C для Motorola 680x0 создаст таблицу возможных значений переменной key, в которой будут присутствовать все элементы от 0 до 255. Реально в программной системе возможно передать только значения констант CC1 – CC3, в результате покрытыми окажутся только три ветки из 255 возможных в исполняемом коде. Никакими стандартными средствами увеличить степень такого покрытия нельзя. В этом случае отчет о покрытии сопровождается дополнительной информацией о причинах невозможности обеспечить полное покрытие.

Сбор информации о покрытии на уровне исполняемого кода наиболее часто применяется в высококритичных программных системах, в которых не допускается наличия «мертвого» исполняемого кода, который потенциально может привести к сбою или отказу во время работы системы. К таким

системам в первую очередь можно отнести авиационные бортовые системы, медицинские системы и системы обеспечения безопасности информации.

4.7. Отчеты о проблемах

4.7.1. Технологические цепочки и роли участников проекта, использующих отчеты о проблемах. Связь отчетов о проблемах с другими типами проектной документации.

Каждое несоответствие с требованиями, найденное тестировщиком, должно быть задокументировано в виде отчета о проблеме. Вероятность обнаружения и исправления ошибки, вызвавшей это несоответствие, зависит от того, насколько качественно она задокументирована. Отчеты о проблемах могут поступать не только от тестировщиков, но и от специалистов технической поддержки или пользователей, однако их общая цель – указать на наличие проблемы в системе, которая должна быть устранена. Если отчет составлен некорректно, разработчик не сможет устранить проблему, поэтому можно считать этот отчет одним из самых важных документов в цепочке тестовой документации.

Главное, что должно быть включено в отчет об ошибке, это:

- Способ воспроизведения проблемы. Для того, чтобы разработчик смог устранить проблему, он должен разобраться в ее причинах, самостоятельно воспроизведя ее (и, возможно, не один раз). Один из самых тяжелых случаев в процессе разработки возникает при невоспроизводимой проблеме, т.е. проблеме, у которой точно неизвестен способ ее вызывать. Нахождение такого способа – одна из самых нетривиальных задач в работе тестировщика.
- Анализ проблемы с кратким ее описанием. Лучше всего приводить описание в тех же терминах, в которых составлены требования на часть системы, в которой обнаружена проблема. В этом случае минимизируется вероятность недопонимания сути проблемы.

Любой отчет о проблеме должен быть составлен немедленно после ее обнаружения. Если отчет будет составлен спустя значительное время, повышается вероятность того, что в него не попадет какая-либо важная информация, которая поможет устранить причину проблемы в кратчайшие сроки.

4.7.2. Структура отчетов о проблемах, их трассировка на программный код и документацию

Структура отчёта о проблеме в целом мало различается в различных проектах, изменения обычно касаются только порядка и имен следования

полей. Некоторые поля могут отражать специфику данного конкретного проекта, однако обычно эти поля следующие:

- Объект, в котором найдена проблема. Здесь помещается максимально полная информация – для документации это название документа, раздел, автор, версия. Для исходных текстов это имя модуля, имя функции/метода или номера строк, версия.
- Выпуск и версия системы. Определяет место, откуда был взят объект с обнаруженной проблемой. Обычно требуется отдельная идентификация версии системы (а не только версии исходных текстов), поскольку может возникнуть путаница с повторно выявленными проблемами. В этом случае, если проблема уже была когда-то выявлена разработчиком и потом вновь появилась из-за того, что в систему попала не самая последняя версия программного модуля, разработчик может решить, что ему пришел старый отчет и проблемы на самом деле не существует.
- Тип отчёта:
    - Ошибка кодирования – код не соответствует требованиям.
    - Ошибка проектирования – тестировщик не согласен с проектной документацией.
    - Предложение – у тестировщика возникла идея, как можно усовершенствовать код.
    - Расхождение с документацией – поведение ПО не соответствует руководству пользователя или проектной документации, или вообще нигде не описано. При этом у тестировщика нет оснований объявлять, где именно находится ошибка.
    - Взаимодействие с аппаратурой – неверная диагностика плохого состояния устройства, ошибка в интерфейсе с устройством.
    - Вопрос – тестировщик не уверен, что это проблема и ему требуется дополнительная информация.
- Степень важности. Строгого критерия определения степени важности не существует и обычно это поле кодируют от 1 (незначительно) до 10 (фатально). Однако способов обоснованной оценки нет – очень сложно определить, насколько фатальной может оказаться, например, опечатка в один символ в руководстве пользователя.
- Суть проблемы. Краткое (не более 2 строчек) определение проблемы. Даже если две проблемы очень похожи, их описания должны различаться.
- Можно ли воспроизвести проблемную ситуацию? Ответ = Да, Нет, Не всегда. Последнее – если проблема носит нерегулярный характер.

Нужно описывать, когда она проявляется, а когда – нет (например, не вовремя нажать не ту клавишу).

- Подробное описание проблемы и способ её воспроизведения. При этом нужны подробности – и в описании условий воспроизведения, и в описании причины объявления получившейся реакции ошибкой.

Обычный вид отчета о проблеме, соответствующего данной структуре, следующий:

Отчёт о проблеме

Порядковый номер отчёта:

Автор отчёта: ___________________ Дата создания отчёта: __ __ __

Документы/разделы, связанные с проблемой:

……………………………………..

Идентификация объекта/процесса, где проявляется проблема:

……………………………………………………………………………………….

Определение проблемы:

……………………………………………………………………………………….

Автор решения: ___________________ Дата формирования решения __ __ __

Принятое решение: (возможно, ссылки на изменяемые компоненты/запросы на изменения)

……………………………………………………………………………………….

Результаты анализа, определяющие, на содержание каких компонент влияет решение:

……………………………………………………………………………………….

План проверок, восстанавливающих текущее состояние документов разработки

…………………………………………………………………………………….

Оценка принятого решения автором отчёта о проблеме: _

…………………………………………………………………………………….

(0 = полностью согласен

1 = не все аспекты проблемы учтены/разрешены

2 = основная часть проблемы осталась неразрешённой

3 = решение не адекватно проблеме – не устраняет её)

4.8. Трассировочные таблицы

4.8.1. Технологические цепочки и роли участников проекта, использующих трассировочные таблицы. Связь трассировочных таблиц с другими типами проектной документации.

На каждом этапе жизненного цикла разработки программной системы создается различного рода проектная документация. Как правило, документация каждого последующего этапа создается на базе документации предыдущего этапа. Для целей упрощения навигации по различным документам и в т.ч. упрощения верификации документации и самой системы часто используются перекрестные ссылки между разделами документов.

Так, например, часто применяемая практика заключается в том, что каждое требование в документах помечается уникальным идентификатором – якорем (anchor). Якоря в требованиях могут иметь, например, следующий формат:

[ANCHOR: код]

Где код записывается в виде AAA_RR_NNNNNN, где AAA – тип документа (SYS, ORD и т.п.), RR – номер раздела верхнего уровня, в котором содержится якорь и NNNNNN – номер ссылки с ведущими нулями.

Требование в таком виде будет выглядеть следующим образом:

Для каждого вычислимого атрибута ***должна*** быть определена роль, от которой производится вычисления. [ANCHOR: SYS_02_000084]

Если возникает необходимость сослаться на требование из того же самого документа или из любого другого, то в ссылке указывается код якоря для

соответствующего требования. Так, например, если ссылка записывается в тексте и имеет следующий формат:

[REF: код]

Где код имеет тот же формат, что и для якоря, ссылка на требования будет иметь следующий вид:

Для доступа к названию роли в формуле расчета значения вычислимого атрибута ***должна*** использоваться мнемоника [RoleName]. [ANCHOR: SRD_02_000058] [REF: SYS_02_000084].

Система якорей и ссылок использует те же самые идеи, что и обычный гипертекст, однако, часто возникает необходимость не только в указании самого факта связи между требованиями или разделами документов, а дополнительно указать тип связи. Например, можно выделить следующие типы связей:

- обычная гиперссылка;
- ссылка требований нижнего уровня на требования верхнего уровня;
- ссылка на различные варианты одного и того же требования, предназначенного для разных вариантов системы (например, для разных платформ).

В этом случае можно указывать тип ссылки рядом с кодом якоря, на который она ссылается. Однако, часто используется и другой метод организации типизированных ссылок между документами – трассировочные таблицы.

В общем случае в строках и столбцах трассировочной таблицы указаны идентификаторы якорей, на которые и из которых идет ссылка, а в ячейке на пересечении строк и столбцов отмечается либо факт наличия ссылки, либо ее тип.

Трассировочные таблицы могут использоваться для машинного анализа ссылочной целостности проектной документации или для быстрой навигации в больших объемах документов.

4.8.2. Возможные формы трассировочных таблиц

Как уже было сказано в предыдущем разделе, одна из форм трассировочных таблиц подразумевает указание идентификаторов якорей в строках и столбцах и типа связи в ячейках на их пересечении. Такая таблица будет выглядеть следующим образом:

	SYS_01_0001	SYS_01_0002	SYS_01_0003	SYS_01_0003

SRD_01_0001	cross-ref			variant
SRD_01_0002	based on		variant	
SRD_01_0003	based on			variant
SRD_01_0004		cross-ref	variant	

В первом столбце перечислены якоря требований к программному обеспечению, в первой строке – якоря системных требований. На пересечении указаны типы ссылок – cross-ref – обычная информационная перекрестная ссылка, based on – требование к ПО основано на данном системном требовании, variant – может использоваться либо одно, либо другое требование к ПО в зависимости от варианта системы.

Также часто используется более простая форма трассировочных таблиц. В первом столбце перечисляются якоря или номера разделов документа, который ссылается на другие. В строке – названия документов, на которые идет ссылка, а в ячейках – якоря или номера разделов, на которые идет ссылка. Трассировочная таблица в этом случае будет выглядеть следующим образом:

РД СВТ	**Protection Profile**	**Protection Profile глава 6**	**Protection Profile главы 1-4**	**Комментарии**
2.2.2 КСЗ должен требовать от пользователей идентифицировать себя при запросах на доступ.	5.2.2.4 FIA_UID.1.2	286, 288	82 ER_AUTHENTICATION, ER_IDENTIFICATION, 57 Identification&Authentication, 80 P.I_AND_A, 58 Non-bypassability	
2.2.2 КСЗ должен подвергать проверке подлинность идентификации - осуществлять аутентификацию.	5.2.2.3 FAI_UAU.1.2	277, 280		В названии трассировочного тега опечатка. Правильно - FIA_UAU.1.2
2.2.2 КСЗ должен препятствовать доступу к	5.1.3.1	286, 287		5.1.3.1 - необходимые данные для

защищаемым ресурсам неидентифицированных пользователей и пользователей, подлинность идентификации которых при аутентификации не подтвердилась.				аутентификации - атрибуты учетной записи пользователя

Здесь РД СВТ – документ Гостехкомиссии РФ, включающий в себя требования по защищенности средств вычислительной техники [], Protection Profile – один из аналогичных документов, используемых в США. Таблица представляет собой трассировку требований РД СВТ на требования различных глав Protection Profile. В первой строке указаны главы, а в ячейчах – конкретные разделы.

ТЕМА 5.Формальные инспекции (лекции 9-10)

5.1. Задачи и цели проведения формальных инспекций

Не всегда возможна разработка автоматических или хотя бы четко формализованных ручных тестов для проверки функциональности программной системы. В некоторых случаях выполнение тестируемого программного кода невозможно в условиях, создаваемых тестовым окружением. Такая ситуация возможна во встроенных системах, если программный код предназначен для обработки исключительных ситуаций, создаваемых только на реальном оборудование.

В тех случаях, когда верифицируется не программный код, а проектная документация на систему, которую нельзя «выполнить» или создать для нее отдельные тестовые примеры также обычно прибегают к методу экспертных исследований программного кода или документации на корректность или непротиворечивость.

Такие экспертные исследования обычно называют инспекциями или просмотрами. Существует два типа инспекций – неформальные и формальные.

При неформальной инспекции автор некоторого документа или части программной системы передает его эксперту, а тот, ознакомившись с

документом, передает автору список замечаний, которые тот исправляет. Сам факт проведения инспекции и замечания, как правило, нигде отдельно не сохраняются, состояние исправлений по замечаниям также нигде не отслеживается.

Формальная инспекция, напротив, является четко управляемым процессом, структура которого обычно четко определяется соответствующим стандартом проекта. Таким образом, все формальные инспекции имеют одинаковую структуру и одинаковые выходные документы, которые затем используются при разработке.

Факт начала формальной инспекции четко фиксируется в общей базе данных проекта. Также фиксируются документы, подвергаемые инспекции, списки замечаний, отслеживаются внесенные по замечаниям изменения. Этим формальная инспекция похожа на автоматизированное тестирование – списки замечаний имеют много общего с отчетами о выполнении тестовых примеров.

В ходе формальной инспекции группой специалистов осуществляется независимая проверка соответствия инспектируемых документов исходным документам. Независимость проверки обеспечивается тем, что она осуществляется инспекторами, не участвовавшими в разработке инспектируемого документа. Входами процесса формальной инспекции являются инспектируемые документы и исходные документы, а выходами – материалы инспекции, включающие список обнаруженных несоответствий и решение об изменении статуса инспектируемых документов. Рис. 21 иллюстрирует место формальной инспекции в процессе разработки программных систем.

Рис. 21 Место формальной инспекции в проекте по разработке программной системы

5.2. Этапы формальной инспекции и роли ее участников

Как правило, процесс формальной инспекции состоит из пяти фаз: инициализация, планирование, подготовка (экспертиза), обсуждение, завершение. В некоторых случаях подготовку и обсуждение целесообразно рассматривать не как последовательные этапы, а как параллельные подпроцессы. В частности, такая ситуация может иметь место при использовании автоматизированной системы поддержки проведения формальных инспекций. Процедура формальной инспекции проекта должна точно описывать порядок проведения формальных инспекций в данном проекте.

После устранения обнаруженных в ходе формальной инспекции несоответствий процесс формальной инспекции повторяется, возможно, в другой форме и с другим составом участников. Процедура формальной инспекции должна регламентировать возможные формы проведения повторной инспекции в зависимости от объема и характера изменений, внесенных в объект инспекции. Как правило, допускается упрощение процесса повторной инспекции (проведение инспекции одним инспектором,

отсутствие фазы обсуждения) при внесении в объект инспекции незначительных изменений относительно ранее инспектировавшейся версии.

5.2.1. Инициализация

Руководитель проекта или его заместитель запрашивает из базы, хранящей все данные проекта (например, из системы конфигурационного управления, см. тему 11) список объектов, готовых к инспекции, выбирает объект инспекции, затем назначает участников формальной инспекции: автора, ведущего и одного или нескольких инспекторов. Ведущий также может выполнять роль инспектора; остальные участники выполняют только одну роль. На роль ведущего или инспектора не допускается назначать сотрудников, участвовавших в разработке объекта инспекции.

Обычно в роли автора выступает один из разработчиков объекта инспекции, но возможны ситуации, когда разработчик недоступен – например, переведен в другой проект или находится в отпуске. Тогда на роль автора назначается сотрудник, который будет исправлять обнаруженные несоответствия в инспектируемых документах. При инспектировании документов, разработанных заказчиком, автор может не назначаться.

Рекомендуется назначать не менее, чем двух инспекторов. Количество инспекторов может быть увеличено, если инспектируются документы, отличающиеся особой сложностью или новизной понятий, а также, если в качестве инспекторов привлекаются сотрудники с недостаточным опытом. Однако рекомендуемое общее число участников инспекции не должно превышать пяти.

В обоснованных случаях процедура формальной инспекции проекта может допускать проведение инспекции единственным инспектором, например, когда объект инспекции отличается особой простотой, и оцениваемые характеристики такого объекта инспекции тривиальны. Примером такого объекта инспекции может служить пакет результатов сбора структурного покрытия, получаемый после выполнения ранее проинспектированных тестов, для которого проверяется только состав пакета и согласованность версий.

В случае, если проводится повторная инспекция по сокращенной форме, ведущий самостоятельно инициирует процесс повторной инспекции без участия руководителя проекта. Процедура формальной инспекции проекта может разрешать ведущему самостоятельно инициировать процесс повторной инспекции (в том же составе участников), даже когда он проводится в полной форме, если это диктуется спецификой проекта.

5.2.2. Планирование

Как только процесс формальной инспекции был инициирован, ведущий проверяет, что инспектируемые документы размещены в базе данных проекта, а их статус соответствует готовности к формальной инспекции. Если это не так, инспекция откладывается.

Затем он должен изменить статус инспектируемых документов так, чтобы отметить факт начала инспекции и ограничить доступ к инспектируемой документации. Во время инспекции изменения документов невозможно, а соответствующий статус сохраняется до конца инспекции. Далее будем называть этот статус Review.

После этого ведущий должен скопировать из базы данных проекта бланк инспекции и занести в него идентификаторы инспектируемых и исходных документов и номера их версий, список участников с указанием их ролей и дату фактического начала процесса инспекции, т.е. того момента, когда инспектируемые документы были переведены в состояние Review.

Ведущий должен оценить время, необходимое инспекторам для подготовки, и продолжительность обсуждения. Время, отводимое на этап подготовки, не может быть менее одного часа. Также ведущий должен определить дату, время и место обсуждения, если оно будет проходить в форме собрания. При этом может потребоваться согласование с другими участниками инспекции. Если оценка продолжительности обсуждения в форме собрания превышает 2 часа, то необходимо запланировать несколько собраний, каждое из которых будет длиться не более двух часов.

Процедура формальной инспекции проекта может допускать проведение повторной инспекции без собрания, если итогом предыдущей инспекции было решение о проведении повторной инспекции в сокращенной форме. Так же допускается не проводить собрание, если результаты формальной инспекции ведутся и хранятся в электронном виде. В этом случае, процедура формальной инспекции проекта должна регламентировать взаимодействия участников формальной инспекции между собой. Кроме того, процедура формальной инспекции проекта должна определять механизм подготовки, проведения обсуждения и принятия решения.

Подготовив бланк инспекции и определив время и место собрания, ведущий должен известить участников инспекции о времени и месте проведения собрания и разослать им подготовленный бланк инспекции.

Процедура формальной инспекции проекта может предусматривать использование бланка, заполненного в ходе предыдущей инспекции, если проводится повторная инспекция в сокращенной форме и при этом ведущий является единственным инспектором.

5.2.3. Подготовка

Получив письмо или назначение с прикрепленным к нему бланком инспекции, инспекторы должны извлечь из базы данных проекта исходные и инспектируемые документы, используя указанные в бланке идентификаторы и номера версий. При этом инспекторы должны убедиться, что все документы находятся в соответствующем состоянии.

В ходе подготовки инспекторы детально изучают инспектируемые документы, руководствуясь списком контрольных вопросов. Обнаруженные несоответствия должны быть точно локализованы, сформулированы и записаны.

При проведении повторной инспекции в сокращенной форме допускается ограничиться анализом изменений по отношению к той версии объекта инспекции, которая инспектировалась на предыдущей инспекции. Если при этом обнаруживается, что имеются изменения, не связанные с зафиксированными замечаниями, то процесс инспекции прерывается и назначается новая инспекция в полной форме. Исключением из этого правила может быть случай, когда такие изменения заключаются в исправлении тривиальных ошибок, не затрагивающих сущности инспектируемых документов, таких, например, как опечатки в комментариях, не влияющие на смысл фразы.

Если при проведении повторной инспекции в сокращенной форме единственным инспектором, он считает, что объем изменений слишком велик, или изменения слишком сложны, он имеет право прервать процесс инспекции, известив руководителя проекта, с тем, чтобы была назначена новая инспекция в полной форме.

Автор, если он не является разработчиком объекта инспекции, должен в процессе подготовки детально с ним ознакомиться, чтобы быть готовым отвечать на вопросы инспекторов в ходе обсуждения, а после завершения инспекции устранить найденные несоответствия.

5.2.4. Обсуждение

Обсуждение проводится в форме одного или нескольких собраний, каждое из которых продолжается не более двух часов. В один день рекомендуется проводить не более одного собрания. Если обсуждение не укладывается в запланированное число собраний, то назначаются дополнительные собрания. Для проведения собрания необходимо присутствие ведущего, хотя бы одного из инспекторов и, как правило, автора. Однако, ведущий может по своему усмотрению провести собрание в отсутствие автора, если тот болен или по какой-либо иной причине не может присутствовать на собрании, при условии, что ни один из инспекторов не обнаружил несоответствий, или их замечания очевидны и не требуют разъяснений со стороны автора, или с автором установлена телефонная связь. Если собрание было начато в

отсутствие автора, а в дальнейшем возникла необходимость его присутствия, ведущий должен прервать и отложить собрание.

Собрание откладывается, если ни один из инспекторов не подготовился к обсуждению. Ведущий также может по своему усмотрению отложить собрание, если не подготовился или отсутствует хотя бы один из инспекторов.

В ходе обсуждения ведущий синхронизирует работу участников, зачитывая инспектируемый документ либо последовательно называя разделы или абзацы текста или элементы диаграмм, либо каким-то иным способом обеспечивает синхронный просмотр документа всеми участниками. По мере продвижения по документу инспекторы прерывают ведущего в тех местах, к которым у них имеются замечания. В случае отсутствия разногласий ведущий фиксирует несоответствие и продолжает продвижение по документу. При инспектировании документов небольшого объема ведущий, по своему усмотрению, может не синхронизировать просмотр документа всеми участниками, а просто опрашивать участников о наличии замечаний; такой опрос может быть совмещен с заполнением списка контрольных вопросов (см. ниже).

Если мнения участников по высказанному замечанию расходятся, то ведущий управляет дискуссией, последовательно предоставляя слово всем желающим высказаться, причем автор пользуется правом внеочередного предоставления слова. Если в результате дискуссии изменилась формулировка замечания, то ведущий записывает эту новую формулировку, затем зачитывает ее, и, если все участники с ней согласны, продолжает продвижение по документу.

Результатом дискуссии может также быть признание отсутствия проблемы. В этом случае ведущий убеждается в том, что все с этим согласны, и продолжает продвижение по документу.

Участники должны стремиться обозначить проблемы, но не искать их решения. Достижение консенсуса по спорным вопросам также не является целью дискуссии. Если имеется расхождение во мнениях, то должны быть зафиксированы все альтернативные мнения. Ведущий должен прервать дискуссию, если оценивает ее как непродуктивную.

Все участники обязаны уважительно относиться к оппонентам, не перебивать говорящего и высказываться тогда, когда ведущий предоставит им слово. Не допускаются параллельные обсуждения узким составом – каждый участник обязан адресовать свои высказывания всему собранию, а не соседу.

Необходимо также избегать критики и оценки квалификации коллег. Целью инспекции является повышение качества инспектируемых документов, а не

оценка квалификации автора или других участников инспекции. Ведущий формальной инспекции не обладает преимуществом перед другими участниками обсуждения, он лишь организует этот процесс и фиксирует его результаты в бланке инспекции.

В ходе обсуждения необходимо в бланке инспекции проставить ответы на контрольные вопросы и зафиксировать замечания. Для этого ведущий последовательно зачитывает контрольные вопросы. При отсутствии у всех инспекторов замечаний, нарушающих сформулированное в вопросе свойство, против вопроса ставится отметка (галочка или крестик) в графе “Yes” или «Да»; в противном случае отметка ставится в графе “No” или «Нет», а в графе “Замечания” (или аналогичной) перечисляются номера соответствующих замечаний, записанных в таблице для замечаний, которая помещена в конце бланка инспекции.

Отметка в графе “N/A” (“Неприменимо”) ставится только в том случае, когда сформулированное в соответствующем вопросе свойство не может быть оценено для данного объекта инспекции; в этом случае в графе “Замечания” записывается обоснование невозможности оценить данное свойство.

Если при проведении повторной инспекции используется бланк от предыдущей инспекции, в котором уже проставлены ответы на контрольные вопросы, то эти ответы не исправляются, а в таблице для замечаний против зафиксированных ранее несоответствий делаются отметки об их устранении. В случае обнаружения новых несоответствий замечания записываются в таблицу после записанных ранее, а следующая повторная инспекция обязательно назначается в полной форме.

В конце обсуждения участники принимают решение о возможности принятия объекта инспекции в имеющейся версии, либо о необходимости внесения исправлений и проведения повторной инспекции в полной или сокращенной форме. Объект инспекции может быть принят в имеющейся версии только при отсутствии несоответствий. Решение о проведении повторной инспекции в сокращенной форме принимается только в том случае, если все участники с этим согласны. Если хотя бы один из участников настаивает на полной форме повторной инспекции, то повторная инспекция должна проводиться в полной форме. Мнение ведущего учитывается наравне с мнениями других участников. Принятое решение фиксируется ведущим на бланке инспекции и заверяется подписями всех участников, включая представителя службы качества, если он присутствовал на собрании.

Теоретически возможна ситуация, когда автор не согласен ни с одним из зафиксированных замечаний, а инспекторы настаивают, что несоответствия имеют место. В таком случае невозможно принять решение об изменении

статуса инспектируемых документов, поэтому инспекция должна быть отложена, а решение проблемы вынесено за рамки процесса формальной инспекции.

На бланке инспекции также фиксируется продолжительность собрания и время, затраченное каждым из участников на подготовку.

Процедура формальной инспекции проекта может допускать отмену обсуждения и собрания, если ни у одного из инспекторов нет замечаний. Это возможно либо при проведении инспекции одним инспектором, одновременно являющимся ведущим, либо при использовании автоматизированной системы поддержки проведения формальных инспекций, которая обеспечит ведущего информацией о несоответствиях, обнаруженных каждым из инспекторов, и о том, что каждый из них подтвердил завершение изучения объекта инспекции. В этом случае заполнение бланка инспекции производится ведущим самостоятельно.

5.2.5. Завершение

В конце собрания, по окончании обсуждения, инспекторы сдают ведущему свои рабочие материалы, которые включают в себя распечатки инспектируемых документов с пометками и бланки инспекции. Ведущий складывает эти материалы в прозрачную папку вместе с экземпляром бланка инспекции, заполненным в ходе обсуждения, причем титульный лист бланка инспекции должен лежать сверху, чтобы можно было по нему идентифицировать папки.

После собрания ведущий изменяет статус инспектируемых документов в базе данных проекта в соответствии с принятым решением – либо им присваивается статус «Принят», либо «Переработать».

В последнем случае необходима повторная инспекция, вид которой уточняется кратким комментарием.

5.3. Документирование процесса формальной инспекции

Обычно, если предприятие ведет несколько проектов по разработке программных систем, процедура формальной инспекции регламентируется в виде стандарта предприятия. Это позволяет сотрудникам, участвующим в формальных инспекциях, легко адаптироваться при переходе из проекта в проект.

Однако каждый проект может иметь свою специфику. В силу этого рекомендуется разрабатывать для каждого проекта свою процедуру формальной инспекции. Процедура формальной инспекции проекта должна уточнять и дополнять настоящий стандарт с учетом специфики данного

проекта. Процедура формальной инспекции проекта не должна противоречить требованиям настоящего стандарта.

Процедура формальной инспекции проекта должна точно описывать порядок проведения формальных инспекций в данном проекте.

В процедуре формальной инспекции проекта не рекомендуется дублировать общие положения настоящего стандарта, за исключением отдельных, особо важных моментов, таких, например, как изменение статуса инспектируемых документов.

В процедуре формальной инспекции проекта должны быть названы все идентификаторы состояний инспектируемых документов в базе данных проекта, с которыми приходится иметь дело участникам формальной инспекции. Как минимум, должны быть названы идентификаторы состояний, обозначающих:

- готовность документа к проведению инспекции;
- прохождение фаз планирования, подготовки и обсуждения;
- необходимость переработки документа;
- подтверждение соответствия исходным документам.

Процедура формальной инспекции проекта должна регламентировать минимальное количество инспекторов для каждого типа объектов инспекции, если в проекте инспектируются документы различного уровня сложности, требующие участия различного количества инспекторов.

Процедура формальной инспекции проекта должна регламентировать возможные формы проведения повторной инспекции в зависимости от объема и характера изменений, внесенных в объект инспекции. Как правило, допускается упрощение процесса повторной инспекции (выполнение инспекции одним инспектором, отсутствие фазы обсуждения) при внесении в объект инспекции незначительных изменений. Процедура формальной инспекции проекта может предусматривать использование бланка от предыдущей инспекции, если проводится повторная инспекция в сокращенной форме. Процедура формальной инспекции проекта может разрешать ведущему самостоятельно инициировать процесс повторной инспекции (в том же составе участников), даже когда он проводится в полной форме, если это диктуется спецификой проекта.

5.3.1. Бланк инспекции

Бланк инспекции – основной документ, который заполняется в ходе проведения инспекций. Обычно он разрабатывается в ходе разработки

стандартов проекта. Для каждого типа объектов инспекции в проекте должен быть разработан свой бланк инспекции.

Бланк инспекции состоит из трех основных частей:

- титульный лист;
- список контрольных вопросов;
- список несоответствий.

Кроме того, рекомендуется на всех страницах бланка, кроме первой, помещать колонтитул, включающий в себя как минимум номер бланка инспекции.

5.3.1.1. Титульный лист

Титульный лист предназначен для идентификации формальной инспекции и записи решения и обычно включает, как минимум, следующие элементы:

- слова «формальная инспекция»;
- идентификатор проекта;
- идентификатор типа объекта инспекции, например, «Тест», «Стандарт проекта» и т.п.;
- идентификатор версии бланка инспекции;
- идентификатор конфигурационной базы данных;
- место для записи идентификаторов каждого из инспектируемых документов;
- место для записи идентификаторов версий каждого из инспектируемых документов;
- место для записи идентификаторов каждого из исходных документов;
- место для записи идентификаторов версий каждого из исходных документов;
- место для записи даты начала инспекции;
- место для записи фактических даты и времени начала собрания;
- место (таблица) для записи фамилий участников инспекции с указанием их ролей и местами для подписи и записи времени, затраченного на подготовку;
- место для записи продолжительности собрания;
- место для фиксации принятого решения.

Идентификатор документа состоит из имени файла в базе данных проекта и полного пути к нему. Общие для разных документов элементы идентификации, такие, как путь или имя базы, могут быть вынесены в отдельные поля бланка.

Если процедурой формальной инспекции проекта предусмотрена возможность проведения повторной инспекции с использованием бланка от предыдущей инспекции, то титульный лист также должен включать следующие поля:

- место для записи даты проведения повторной инспекции;
- место для записи идентификаторов версий каждого из повторно инспектируемых документов;
- место для записи фамилии ведущего повторной инспекции;
- место для записи времени, затраченного ведущим на проведение повторной инспекции;
- место для фиксации принятого решения;
- место для подписи ведущего.

Все перечисленные элементы должны располагаться на одной странице.

5.3.1.2. Список контрольных вопросов

Список контрольных вопросов должен быть оформлен в виде таблицы, состоящей из следующих колонок:

- порядковый номер;
- текст вопроса;
- место для положительного ответа («Yes» или «Да»);
- место для отрицательного ответа («No» или «Нет»);
- место для ответа «N/A» или «Неприменимо»;
- место для ссылки на несоответствие.

Контрольные вопросы должны быть сформулированы таким образом, чтобы положительный ответ означал отсутствие несоответствий. Формулировки должны быть понятными, четкими и однозначными.

5.3.1.3. Список несоответствий

Список несоответствий должен быть оформлен в виде незаполненной таблицы с тремя колонками:

- для порядкового номера;
- для описания несоответствия;
- для отметки об исправлении.

5.3.1.4. Колонтитул

Колонтитул должен включать:

- идентификатор проекта;
- идентификатор версии бланка инспекции;
- место для записи идентификаторов хотя бы одного из инспектируемых документов;
- место для записи идентификаторов версий хотя бы одного из инспектируемых документов.

5.3.2. Жизненный цикл инспектируемого документа

В процессе формальной инспекции существует 2 типа документов:

- документы проекта (целевой документ, исходный документ, поддерживающий документ);
- вспомогательные документы (отчет о проведенной инспекции, список контрольных вопросов, список обнаруженных проблем).

Вспомогательные документы возникают в процессе инспекции и могут изменяться в течение процесса инспекции. Титульный лист создается на стадии инициализации. Список обнаруженных проблем создается на стадии подготовки. Список контрольных вопросов заполняется на стадии обсуждения. После завершения процесса инспекции вспомогательные документы помещаются в архив и более не подлежат изменению. Вспомогательные документы хранятся в соответствии с установленными для них сроками.

В процессе формальной инспекции инспектируемый документ последовательно сменяет несколько состояний. В процессе разработки (до начала формальной инспекции) документ имеет состояние Active (Активный). В этом состоянии автор может обращаться к документу как по чтению, так и по записи. После того, как автор посчитал, что закончил работу над документом, он переводит документ в состояние Ready (Готов). Это означает, что документ готов к формальной инспекции. В состоянии Ready автор уже не может изменять документ. Следующим состоянием документа является Review (формальная инспекция). В это состояние документ помещается на стадии инициализации формальной инспекции. Перевод документа в состояние Review осуществляет ведущий. В состоянии Review доступ к документу возможен только для чтения для всех участников формальной инспекции. Если документ прошел формальную инспекцию (не было обнаружено проблем), то он переходит в состояние Approved (Утвержден). Перевод документа в состояние Approved осуществляет ведущий. В этом состоянии документ доступен только для чтения для всех участников формальной инспекции, а также для остальных участников проекта. Если же после формальной инспекции в целевом документе требуются исправления, документ переводится в состояние Update

(Переработка). В этом состоянии автор имеет доступ к документу как по чтению, так и по записи. После переработки документа автор присваивает документу состояние Ready и процесс перехода по состояниям повторяется до тех пор, пока документ не будет переведен в состояние Approved. Если в инспектируемый документ не требуется вносить значительных изменений то, после того как ведущий убедится в том, что необходимые исправления были сделаны, целевой документ может быть переведен в состояние Approved.

Инспектируемый документ подлежит исправлению после завершения процесса инспекции. После исправления целевой документ может пройти повторную инспекцию. Таким образом, целевой документ может пройти несколько последовательных инспекций (совершить несколько витков жизненного цикла документов в процессе формальной инспекции). Общий жизненный цикл инспектируемого документа может изображен на Рис. 22.

Рис. 22 Жизненный цикл инспектируемого документа в процессе формальной инспекции

5.4. Формальные инспекции программного кода

Процесс формальной инспекции программного кода подчиняется всем правилам, определенным для абстрактной формальной инспекции, однако, имеет некоторые особенности, связанные, в первую очередь со структурой инспектируемого программного кода, а также с тем, что обычно инспектируются участки кода, которые невозможно проверить при помощи автоматизированного тестирования, основанного на тестовых примерах.

5.4.1. Особенности этапа просмотра инспектируемого кода

Выделение ключевых точек и построение или использование таблиц трассировки. Перед началом просмотра исходного кода рекомендуется отметить пункты требований, на соответствие которым проверяется исходный код, а также записать обоснования того, почему эти требования не могут быть проверены в автоматическом режиме. После этого можно переходить к просмотру собственно исходного кода. Все пометки, которые

придется вносить в ходе инспектирования в исходный код необходимо делать не в файле, хранящемся в базе данных проекта, а в его копии, которая потом будет подшита к материалам инспекции. Копия может быть в том же формате, что и исходный файл, либо распечатана на бумаге или выведена в формат DOC, PDF или аналогичный, допускающий комментирование.

При помощи трассировочных таблиц в исходном коде определяются инспектируемые функции или методы, соответствующие необходимым требованиям. Участки кода выделяются и отмечаются меткой или номером соответствующего требования. Если участок кода соответствует требованиям, то необходимо отметить этот факт либо цветом выделения, либо соответствующим текстовым примечанием. Если участок кода имеет проблемы, этот факт должен быть отражен либо цветом выделения, либо ссылкой на соответствующий пункт списка замечаний в бланке инспекции.

В случае отсутствия трассировочных таблиц требований на исходный код рекомендуется делать пометки, поясняющие, почему именно данный участок кода реализует указанные требования. Такие пометки помогут на этапе обсуждения документа.

Проверка стиля кодирования. Отдельным объектом проверки при формальной инспекции программного кода является стиль кодирования. В большинстве проектов существуют стандарты, описывающие правила оформления исходных текстов программ и файлов данных. Неверный стиль кодирования не влияет на работоспособность программы в целом, но значительно затрудняет сопровождение и поддержку изменений в ходе дальнейшего развития системы. Поэтому отклонения от стиля кодирования в инспектируемых участках кода также должны отмечаться в тексте и в списке замечаний. В некоторых случаях проводят инспекции, целиком направленные на проверку стиля кодирования.

Проверка надежности кода. В некоторых случаях рекомендуется проверять наличие участков, гарантирующих робастность, даже если требования прямо не определяют необходимости обработки недопустимых значений. В случае, если потенциально возможна некорректная работа программы из-за отсутствия обработчиков неверных значений, рекомендуется отметить это в списке замечаний.

5.4.2. Особенности этапа проведения собрания

Распределение ролей. В составе инспекторов желательно иметь хотя бы одного специалиста, представляющего себе особенности выполнения инспектируемого кода в реальной установке системы. Это особенно важно при тестировании встроенных систем, тестирование которых проводится на эмуляторах. Во время собрания такой специалист может помогать ведущему

определять последовательность рассмотрения замечаний в случае их большого количества.

Управление собранием. При проведении собрания нецелесообразно зачитывать текст инспектируемого файла, как это обычно рекомендуется. Вместо этого ведущему лучше ограничиться перечислением имен функций или методов, либо, в случае, если в ходе инспекции проверяется соответствие исходного кода требованиям – перечислением номеров или идентификаторов требований. Инспектора при наличии замечаний по функции или требованию поднимают руку и зачитывают замечания.

5.4.3. Особенности этапа завершения и повторной инспекции

Документирование собрания. Для облегчения труда автора инспектируемого документа по исправлению замечаний, каждое замечание, признанное на собрании существенным, рекомендуется точно трассировать на строки исходного кода и требований.

Контроль за внесением изменений. При повторной инспекции исходных текстов рекомендуется использовать специализированные инструментальные средства для сравнения файлов. Изменения по итогам инспекции должны вноситься только в те участки, к которым были высказаны замечания. В случае наличия других изменений ведущий вправе назначить новую инспекцию в полной форме.

5.5. Формальные инспекции проектной документации

Процесс формальной инспекции проектной документации подчиняется всем правилам, определенным для абстрактной формальной инспекции, однако, имеет некоторые особенности, связанные, в первую очередь, с тем, что у проектной документации проверяется ее непротиворечивость и полнота. Точное определение этих терминов в рассматриваемом контексте затруднительно, поэтому под непротиворечивостью будем понимать отсутствие в проектной документации требований, с противоположным смыслом, согласно которым возможно несколько совершенно различных вариантов реализации программной системы. Под полнотой будем понимать достаточность требований для однозначной реализации поведения системы.

5.5.1. Особенности этапа просмотра документации

При инспектировании требований к системе, как правило, рассматривается как «внешняя», так и «внутренняя» информация, касающаяся данного документа. Под внешней информацией понимается в первую очередь суть технических решений, принятых при разработке системы, те принципы, которые отличают ее от других систем. При этом проверяется согласованность требований с этими принципами. Под внутренней

информацией понимается прежде всего внутренняя целостность и непротиворечивость документа – свойства, которые позволяют разрабатывать программный код, лишенный двусмысленностей и нестабильных участков. Главный вопрос, на который должен ответить инспектор при проверке внутренних свойств документа: "Определены ли требования так, чтобы коллектив разработчиков смог работать с ним?" или иначе "Эти требования недвусмысленны, полны и авторитетно выражены?". Процесс инспекции может помочь ответить на эти вопросы, а список контрольных вопросов, представленный ниже, может быть использован при проведении инспекций:

1. Является ли каждое требование совершенно недвусмысленным? (Если требование прочесть подряд несколько раз, делая ударение сначала на первом слове, потом - на втором, затем - на третьем и т.д., будет ли при этом меняться смысл этого требования?)
2. Существует ли для каждого из установленных требований некоторый компетентный специалист, который сможет сказать после завершения разработки, выполнено ли данное требование или нет? Определен ли метод решения этой проблемы в документации по требованиям?
3. Существует ли какие-либо не установленные или отсутствующие требования?
4. Существуют ли среди заданных такие требования, которые не являются необходимыми?
5. Если существуют какие-либо конфликтующие требования, известно ли понятное решающее правило, в каких ситуациях какому требованию следует отдавать предпочтение?

5.5.2. Особенности этапа завершения

Влияние несогласованности документации на процесс разработки. Трассировка изменений на программный код. Первичная инспекция проектной документации, как правило, проводится тогда, когда сама программная система еще не написана. Однако при проведении инспекции изменений в требованиях к уже работающей системе (например, при обновлении ее версии), может потребоваться комплексная одновременная инспекция документации и созданного на ее основе программного кода. При этом может возникнуть ситуация, когда изменения, которые требуется внести в документацию по результатам инспекции, требуют соответствующего изменения в программном коде. Решение этой проблемы достигается использованием трассировочных таблиц.

Несколько иная ситуация возникает в случае, когда комплексная инспекция проводится не после изменения требований, а после завершения всей цепочки изменений – после изменения функциональных требований, архитектуры, низкоуровневых требований и программного кода. В этом

случае при обнаружении противоречивых требований необходимо выявить все части программной системы, которые реализуют эти требования. В случае, если разработка этих частей выполнялась разными людьми, могла различаться и трактовка противоречивых требований. В этом случае ликвидация противоречивости может повлечь за собой «волну изменений» в проектной документации и исходных текстах системы. Для того, чтобы избежать «волн изменений» по завершению инспекций рекомендуется проводить ее поэтапно до начала следующего этапа жизненного цикла или до разработки документов следующего уровня детализации.

ТЕМА 6.Модульное тестирование (лекция 11)

Как уже было сказано ранее, процесс верификации активен в течение практически всего жизненного цикла системы и работает параллельно с процессом разработки. Разработка системы, как правило, идет на различных уровнях – вначале разрабатывается концепция системы, системные требования, затем архитектура системы, ее разбиение на модули, затем разрабатываются отдельные модули. Последовательность этих уровней зависит от типа жизненного цикла, но их состав практически всегда одинаков. Процесс верификации также разбивается на отдельные уровни:

- *системное тестирование*, в ходе которого тестируется система в целом;
- *интеграционное тестирование*, в ходе которого тестируются группы взаимодействующих модулей и компонент системы;
- *модульное тестирование*, в ходе которого тестируются отдельные компоненты;

Технические аспекты методов разработки и проведения тестирования на каждом из трех уровней были рассмотрены в предыдущих темах – на каждом из уровней разрабатываются тестовое окружение, автоматизированные тесты, проводятся формальные инспекции. Однако, каждый из этих трех уровней имеет свои организационные особенности, рассмотрению которых будут посвящены следующие три темы, начиная с данной.

6.1. Задачи и цели модульного тестирования

Каждая сложная программная система состоит из отдельных частей – модулей, выполняющих ту или иную функцию в составе системы. Для того, чтобы удостовериться в корректной работе системы в целом, необходимо вначале протестировать каждый модуль системы по отдельности. В случае возникновения проблем при тестировании системы в целом это позволяет проще выявить модули, вызвавшие проблему и устранить соответствующие дефекты в них. Такое тестирование модулей по отдельности получило называние *модульного тестирования* (*unit testing*).

Для каждого модуля, подвергаемого тестированию, разрабатывается тестовое окружение, включающее в себя драйвер и заглушки, готовятся тест-требования и тест-планы, описывающие конкретные тестовые примеры.

Основная цель модульного тестирования – удостовериться в соответствии требованиям каждого отдельного модуля системы перед тем, как будет произведена его интеграция в состав системы.

При этом в ходе модульного тестирования решаются следующие основные задачи []:

1. Поиск и документирование несоответствий требованиям
2. Поддержка разработки и рефакторинга низкоуровневой архитектуры системы и межмодульного взаимодействия
3. Поддержка рефакторинга модулей
4. Поддержка устранения дефектов и отладки

Первая задача – классическая задача тестирования, включающая в себя не только разработку тестового окружения и тестовых примеров, но и выполнение тестов, протоколирование результатов выполнения, составление отчетов о проблемах.

Вторая задача больше свойственна «легким» методологиям типа XP, в которых применяется принцип тестирования перед разработкой (Test-driven development), при котором основным источником требований для программного модуля является тест, написанный до реализации самого модуля. Однако, даже при классической схеме тестирования, модульные тесты могут выявить проблемы в дизайне системы и нелогичные или запутанные механизмы работы с модулем.

Третья задача связана с поддержкой процесса изменения системы. Достаточно часто в ходе разработки требуется проводить рефакторинг модулей или их групп – оптимизацию или полную переделку программного кода с целью повышения его сопровождаемости, скорости работы или надежности. Модульные тесты при этом являются мощным инструментом для проверки того, что новый вариант программного кода выполняет те же функции, что и старый.

Последняя, четвертая, задача сопряжена с обратной связью, которую получают разработчики от тестировщиков в виде отчетов о проблемах. Подробные отчеты о проблемах, составленные на этапе модульного тестирования, позволяют локализовать и устранить многие дефекты в программной системе на ранних стадиях ее разработки или разработки ее новой функциональности.

В силу того, что модули, подвергаемые тестированию, обычно невелики по размеру, модульное тестирование считается наиболее простым (хотя и достаточно трудоемким) этапом тестирования системы. Однако, несмотря на внешнюю простоту, с модульным тестированием связано две проблемы.

Первая из них связана с тем, что не существует единых принципов определения того, что в точности является отдельным модулем.

Вторая заключается в различиях в трактовке самого понятия модульного тестирования – понимается ли под ним обособленное тестирование модуля, работа которого поддерживается только тестовым окружением или речь идет о проверке корректности работы модуля в составе уже разработанной системы. В последнее время термин «модульное тестирование» чаще используется во втором смысле, хотя в этом случае речь скорее идет об интеграционном тестировании.

Эти две проблемы рассмотрены в двух следующих разделах.

6.2. Понятие модуля и его границ. Тестирование классов.

Традиционное определение модуля с точки зрения его тестирования – «модуль – это компонент минимального размера, который может быть независимо протестирован в ходе верификации программной системы». В реальности часто возникают проблемы с тем, что считать модулем. Существует несколько подходов к данному вопросу:

- модуль – это часть программного кода, выполняющая одну функцию с точки зрения функциональных требований;
- модуль – это программный модуль, т.е. минимальный компилируемый элемент программной системы;
- модуль – это задача в списке задач проекта (с точки зрения его менеджера);
- модуль – это участок кода, который может уместиться на одном экране или одном листе бумаги;
- модуль – это один класс или их множество с единым интерфейсом.

Обычно за тестируемый модуль принимается либо программный модуль (единица компиляции) в случае, если система разрабатывается на процедурном языке программирования или класс, если система разрабатывается на объектно-ориентированном языке.

В случае систем, написанных на процедурных языках, процесс тестирования модуля происходит так, как это было рассмотрено в темах 2-4 – для каждого модуля разрабатывается тестовый драйвер, вызывающий функции модуля и собирающий результаты их работы и набор заглушек, которые имитируют

поведение функций, содержащихся в других модулях, не попадающих под тестирование данного модуля. При тестировании объектно-ориентированных систем существует ряд особенностей, прежде всего вызванных инкапсуляцией данных и методов в классах.

В случае объектно-ориентированных систем более мелкое деление классов и использование отдельных методов в качестве тестируемых модулей нецелесообразно в связи с тем, что для тестирования каждого метода потребуется разработка тестового окружения, сравнимого по сложности с уже написанным программным кодом класса. Кроме того, декомпозиция класса нарушает принцип инкапсуляции, согласно которому объекты каждого класса должны вести себя как единое целое с точки зрения других объектов.

Процесс тестирования классов как модулей иногда называют компонентным тестированием. В ходе такого тестирование проверяется взаимодействие методов внутри класса и правильность доступа методов к внутренним данным класса. При таком тестировании возможно обнаружение не только стандартных дефектов, связанных с выходами за границы диапазона или неверно реализованными требованиями, а также обнаружение специфических дефектов объектно-ориентированного программного обеспечения:

- дефектов инкапсуляции, в результате которых, например, сокрытые данные класса оказывается недоступными при помощи соответствующих публичных методов;
- дефектов наследования, при наличии которых схема наследования блокирует важные данные или методы от классов-потомков;
- дефектов полиморфизма, при которых полиморфное поведение класса оказывается распространенным не на все возможные классы;
- дефектов инстанцирования, при которых во вновь создаваемых объектах класса не устанавливаются корректные значения по умолчанию параметров и внутренних данных класса.

Однако, согласно [], выбор класса в качестве тестируемого модуля имеет и ряд сопряженных проблем:

- **Определение степени полноты тестирования класса.** В том случае, если в качестве тестируемого модуля выбран класс, не совсем ясно, как определять степень полноты его тестирования. С одной стороны, можно использовать классический критерий полноты покрытия программного кода тестами – если полностью выполнены все структурные элементы всех методов – как публичных, так и скрытых, то тесты можно считать полными.

Однако существует альтернативный подход к тестированию класса, согласно которому все публичные методы должны предоставлять пользователю данного класса согласованную схему работы и достаточно проверить типичные корректные и некорректные сценарии работы с данным классом. Т.е., например, в классе, объекты которого представляют записи в телефонной книжке, одним из типичных сценариев работы будет «Создать запись◊искать запись и найти ее◊удалить запись◊искать запись вторично и получить сообщение об ошибке».

Различия в этих двух методах напоминают различия между тестированием черного и белого ящиков, но на самом деле второй подход отличается от черного ящика тем, что функциональные требования на систему могут быть составлены на уровне более высоком, чем отдельные классы и установление адекватности тестовых сценариев требованиям остается на откуп тестировщику.

- **Протоколирование состояний объектов и их изменений.** Некоторые методы класса предназначены не для выдачи информации пользователю, а для изменения внутренних данных объекта класса. Значение внутренних данных объекта определяет его состояние в каждый отдельный момент времени, а вызов методов, изменяющих данные, изменяет и состояние объекта. При тестировании классов необходимо проверять, что класс адекватно реагирует на внешние вызовы в любом из состояний. Однако, зачастую, из-за инкапсуляции данных невозможно определить внутреннее состояние класса программными способами внутри драйвера.

В этом случае может помочь составление схемы поведения объекта, как конечного автомата с определенным набором состояний (подобно тому, как это было описано в теме 2 в разделе «Генераторы сигналов. Событийно-управляемый код»). Такая схема может входить в низкоуровневую проектную документацию (например, в составе описания архитектуры системы), а может составляться тестировщиком или разработчиком на основе функциональных требований к системе. В последнем случае для определения всех возможных состояний может потребоваться ручной анализ программного кода и определение его соответствия требованиям. Автоматизированное тестирование в этом случае может лишь определить, по всем ли выявленным состояниям осуществлялись переходы и все ли возможные реакции проверялись.

- **Тестирование изменений.** Как уже упоминалось выше, модульные тесты – мощный инструмент проверки корректности изменений, внесенных в исходный код при рефакторинге. Однако, в результате рефакторинга только одного класса как правило не меняется его

внешний интерфейс с другими классами (интерфейсы меняются при рефакторинге сразу нескольких классов). В результате обычных эволюционных изменений системы у класса может меняться внешний интерфейс, причем как по формальным признакам (изменяются имена и состав методов, их параметры), так и по функциональным (при сохранении внешнего интерфейса меняется логика работы методов). Для проведения модульного тестирования класса после таких изменений потребуется изменение драйвера и, возможно, заглушек. Но только модульного тестирования в данном случае недостаточно, необходимо также проводить и интеграционное тестирование данного класса вместе со всеми классами, которые связаны с ним по данным или по управлению.

Вне зависимости от того, на какие модули, подвергаемые тестированию, разбивается система, рекомендуется изложить принципы выделения тестируемых модулей в плане и стратегии тестирования, а также составить на базе структурной схемы архитектуры системы новую структурную схему, на которой отметить все тестируемые модули. Это позволит спрогнозировать состав и сложность драйверов и заглушек, требуемых для модульного тестирования системы. Такая схема также может использоваться позже на этапе модульного тестирования для выделения укрупненных групп модулей, подвергаемых интеграции.

6.3. Подходы к проектированию тестового окружения

Вне зависимости от того, какая минимальная единица исходных кодов системы выбирается за минимальный тестируемый модуль, существует еще одно различие в подходах к модульному тестированию.

Первый подход к модульному тестированию основывается на предположении, что функциональность каждого вновь разработанного модуля должна проверяться в автономном режиме без его интеграции с системой. При таком подходе для каждого вновь разрабатываемого модуля создается тестовый драйвер и заглушки, при помощи которых выполняется набор тестов. Только после устранения всех дефектов в автономном режиме производится интеграция модуля в систему и проводится тестирование на следующем уровне. Достоинством данного подхода является более простая локализация ошибок в модуле, поскольку при автономном тестировании исключается влияние остальных частей системы, которое может вызывать маскировку дефектов (эффект четного числа ошибок). Основной недостаток данного метода – повышенная трудоемкость написания драйверов и заглушек, поскольку заглушки должны адекватно моделировать поведение системы в различных ситуациях, а драйвер должен не только создавать тестовое окружение, но и имитировать внутреннее состояние системы, в составе которой должен функционировать модуль.

Второй подход построен на предположении, что модуль все равно работает в составе системы и если модули интегрировать в систему по одному, то можно протестировать поведение модуля в составе всей системы. Этот подход свойственен большинству современных «облегченных» методологий разработки, в том числе и XP.

В результате применения такого подхода резко сокращаются трудозатраты на разработку заглушек и драйверов – в роли заглушек выступает уже оттестированная часть системы, а драйвер выполняет только функции передачи и приема данных, не моделируя внутреннее состояние системы.

Тем не менее, при использовании данного метода возрастает сложность написания тестовых примеров – для приведения системы в нужное состояние системы заглушек, как правило, требуется только установить значения тестовых переменных, а для приведения в нужное состояние части реальной системы необходимо выполнить сценарий, приводящий в это состояние. Каждый тестовый пример в этом случае должен содержать такой сценарий.

Кроме того, при этом подходе не всегда удается локализовать ошибки, скрытые внутри модуля и которые могут проявиться при интеграции следующих модулей.

6.4. Организация модульного тестирования

Модульное тестирование с точки зрения тестировщика – это комплекс работ по выявлению дефектов в тестируемых модулях. В эти работы включается анализ требований, разработка тест-требований и тест-планов, разработка тестового окружения, выполнение тестов, сбор информации об их прохождении.

Однако, с точки зрения руководителя группы тестирования (или с точки зрения руководителя проекта, если в нем не выделена отдельная группа тестирования) модульное тестирование является более широким понятием. Для того, чтобы процесс модульного тестирования мог функционировать совместно с другими процессами разработки, он должен включать в себя несколько фаз: планирование процесса, разработку тестов, выполнение тестов, сбор статистики, управление отчетами о выявленных дефектах.

Согласно стандарту IEEE 1008 [] процесс модульного тестирования состоит из трех фаз, в состав которых входит 8 видов деятельности (этапов):

1. Фаза планирования тестирования
    1. Этап планирования основных подходов к тестированию, ресурсное планирование и календарное планирование
    2. Этап определения свойств, подлежащих тестированию

3. Этап уточнения основного плана, сформированного на этапе (а)

2. Фаза получения набора тестов

1. Этап разработки набора тестов
2. Этап реализации уточненного плана

3. Фаза измерений тестируемого модуля

1. Этап выполнения тестовых процедур
2. Этап определения достаточности тестирования
3. Этап оценки результатов тестирования и тестируемого модуля.

Во время этапа планирования основных подходов в качестве входных данных используется общий план проекта (модульное тестирование, как часть проектных работ, должно укладываться в общий график) и требования к системе (для оценки трудоемкости работ и любого планирования необходимо проводить анализ сложности системы на основании требований к ней).

Основные задачи, решаемые в ходе этапа планирования, включают в себя:

- **определение общего подхода к тестированию модулей** – определяются риски и на их основе – степень полноты и охвата тестирования системы. Определяются источники входных и выходных данных. Определяются технологии проверки результатов тестирования и форматы записи данных о проведенном тестировании. Описывается внешний интерфейс тестируемых модулей и их информационное окружение;
- **определение требований к полноте тестирования** – определяется необходимая степень покрытия программного кода различных участков тестируемого модуля, определяется подходы к классам эквивалентности (требуется ли тестирование за границами диапазона);
- **определение требований к завершению тестирования** – определяются условия, проверка которых позволяет утверждать, что тестирование модуля завершено и условия при которых дальнейшее тестирование модуля считается невозможным до его изменения и доработки. Примером таких условий может служить достижение определенного уровня покрытия исходного кода тестами и невозможность компиляции модуля соответственно;
- **определение требований к ресурсам –** для разработки и выполнения тестов, а также для анализа результатов тестирования необходимы ресурсы – как технические (компьютеры и программное обеспечение), так и людские (тестировщики). При решении этой задачи необходимо указывать требования к программному и аппаратному обеспечению, требования к необходимой квалификации людей, а также должно

определяться необходимое для проведения количество ресурсов и время их занятости;
- **определение общего плана-графика работ** – на основании общего плана проекта составляется план работ по модульному тестированию. Основной критерий начала работ по тестированию – готовность модулей, т.е. общий план работ по тестированию согласуется по датам начала работ с датами окончания работ общего плана разработки.

После завершения этапа планирования начинается этап определения свойств системы, подлежащих тестированию.

Основные задачи, решаемые в ходе деятельности по определению свойств системы, подлежащих тестированию, включают в себя:

- **изучение функциональных требований** – определение тестопригодности требований, при необходимости запрашивается уточнение требований;
- **определение дополнительных требований и связанных процедур** – определение требований, не попадающих под функциональные требования, но которые могут быть протестированы на уровне модульного тестирования (например, это могут быть требования к производительности системы, входящие в состав системных требований);
- **определение состояний тестируемого модуля** – если тестируемый модуль может быть представлен в виде конечного автомата с определенным набором состояний, то каждое состояние должно быть идентифицировано, а также должны быть выделены все требования, относящиеся к этому состоянию;
- **определение характеристик входных и выходных данных** – для всех данных, которые поступают в модуль, а также выходят из него, должны быть определены форматы, частота поступления, допустимые значения и т.п.;
- **выбор элементов, подвергаемых тестированию** – в случае, когда не может применяться полное тестирование, необходимо выбрать элементы тестируемого модуля, которые будут подвергаться тестированию. Основной источник информации здесь – данные о рисках, проанализированные на уровне структуры исходного кода тестируемого модуля. Для тестирования в первую очередь должны отбираться элементы с максимальной степенью риска.

И наконец, в завершение фазы планирования производится уточнение основного плана – уточняется общий подход к тестированию, формулируются специальные и дополнительные требования к ресурсам, составляется детальный план-график работ.

По завершению этих этапов фаза планирования считается оконченной и начинается фаза разработки тестов. При этом процесс разработки тестов подчиняется тем планам и требованиям, которые были созданы на предыдущем этапе. Таким образом, если на первом этапе основную роль выполнял руководитель группы тестирования, то на втором этапе основную роль начинает играть тестировщик, действующий в согласии с указаниями руководителя.

Фаза разработки тестов начинается с собственно разработки набора тестов, который будет использован для тестирования модуля. Основные документы, которые используются на этом этапе: функциональные требования к модулю, архитектура модуля, список элементов, подвергаемых тестированию, план-график работ, определения тестовых примеров от предыдущей версии модуля (если они существовали) и результаты тестирования прошлой версии (если они существовали).

В ходе этого этапа должны быть решены следующие задачи:

- **разработка архитектуры тестового набора** – под тестовым набором здесь понимается не набор конкретных тестовых примеров, а общая структура системы тестов для проверки функциональности тестируемого модуля. Организация тестов в такой системе как правило отражает структуру функциональных требований и зачастую представляет собой иерархию, на каждом уровне которой определяется свой набор тестов;
- **разработка явных тестовых процедур (тест-требований)** – в случае достаточно подробных функциональных требований и четко прописанной концепции разработки тестов, явные тестовые процедуры могут и не разрабатываться. Однако, в случае наличия необходимых ресурсов, разработка тест-требований позволит более четко интерпретировать подвергаемые тестированию функциональные требования – тест-требования снижают риск неоднозначной трактовки функциональных требований;
- **разработка тестовых примеров** – тестовые примеры должны соответствовать требованиям к полноте тестирования и составляться либо на базе тест-требований, либо на основании функциональных требований. Данный вид деятельности наиболее продолжителен во времени;
- **разработка тестовых примеров, основанных на архитектуре (в случае необходимости)** – некоторые тестовые примеры основываются не на функциональных требованиях, а на особенностях архитектуры тестируемого модуля. Для разработки тестовых примеров, основанных на архитектуре, необходимо использовать подход стеклянного ящика. Эти тестовые примеры пишутся либо на основе низкоуровневых требований к системе, либо на основе низкоуровневых тест-

требований, если они разрабатывались на одном из предыдущих этапов;
- **составление спецификации тестовых примеров** – результатом деятельности тестировщика в ходе данного этапа составляется документ Test Design Specification (формат которого описан в стандарте IEEE 829 []).

На следующем этапе проводится реализация тестов (например, в виде тестового окружения и формализованных описаний тестовых примеров). В ходе этого этапа формируются тестовые наборы данных, которые используются в тестовых примерах, создается тестовое окружение, осуществляется интеграция тестового окружения с тестируемым модулем.

После того, как все тесты реализованы, они выполняются на тестовом стенде в ручном или автоматическом режиме. Вне зависимости от вида тестирования в ходе этого этапа решаются две задачи: выполнение тестовых примеров, и сбор и анализ результатов тестирования.

Сбору подлежит следующая информация:

- результат выполнения каждого тестового примера (прошел/не прошел);
- информация об информационном окружении системы в случае, если тест не прошел;
- информация о ресурсах, которые потребовались для выполнения тестового примера.

По результатам анализа этой информации составляются запросы на изменение проектной документации, программного кода тестируемого модуля или тестового окружения.

Этапы разработки (доработки), реализации и выполнения тестов продолжаются до тех пор, пока не будет достигнут критерий завершения модульного тестирования. Примером такого критерия может служить отсутствие не прошедших тестовых примеров при 75% покрытии строк исходного кода.

После прекращения тестирования выполняются работы по оценке проведенного тестирования, в ходе которых:

- описываются отличия реального процесса тестирования от запланированного;
- отличия поведения тестируемого модуля от описанного в требованиях (с целью дальнейшей коррекции требований);

- составляется общий отчет о прохождении тестов, включающий в себя и информацию о покрытии.

В завершение модульного тестирования необходимо проверить, что все созданные в его ходе артефакты – документы, программный код, файлы отчетов и данных – помещены в базу данных проекта, хранящую все данные, используемые и создаваемые в процессе разработки программной системы.

ТЕМА 7.Интеграционное тестирование (лекция 12)

7.1. Задачи и цели интеграционного тестирования

Результатом тестирования и верификации отдельных модулей, составляющих программную систему, является заключение о том, что эти модули являются внутренне непротиворечивыми и соответствуют требованиям. Однако отдельные модули редко функционируют сами по себе, поэтому следующая задача после тестирования отдельных модулей – тестирование корректности взаимодействия нескольких модулей, объединенных в единое целое. Такое тестирование называют *интеграционным*. Его цель – удостовериться в корректности совместной работы компонент системы.

Интеграционное тестирование называют еще *тестированием архитектуры системы*. С одной стороны это связано с тем, что, интеграционные тесты включают в себя проверки всех возможных видов взаимодействий между программными модулями и элементами, которые определяются в архитектуре системы – таким образом, интеграционные тесты проверяют *полноту* взаимодействий в тестируемой реализации системы. С другой стороны, результаты выполнения интеграционных тестов – один из основных источников информации для процесса улучшения и уточнения архитектуры системы, межмодульных и межкомпонентных интерфейсов. Т.е. с этой точки зрения интеграционные тесты проверяют *корректность* взаимодействия компонент системы.

Примером проверки корректности взаимодействия могут служить два модуля, один из которых накапливает сообщения протокола о принятых файлах, а второй выводит этот протокол на экран. В функциональных требованиях на систему записано, что сообщения должны выводиться в обратном хронологическом порядке. Однако, модуль хранения сообщений сохраняет их в прямом порядке, а модуль вывода – использует стек для вывода в обратном порядке. Модульные тесты, затрагивающие каждый модуль по отдельности, не дадут здесь никакого эффекта – вполне реальна обратная ситуация, при которой сообщения хранятся в обратном порядке, а выводятся с использованием очереди. Обнаружить потенциальную проблему можно только проверив взаимодействие модулей при помощи

интеграционных тестов. Ключевым моментом здесь является, что в обратном хронологическом порядке сообщения выводит система в целом, т.е. проверив модуль вывода и обнаружив, что он выводит сообщения в прямом порядке, мы не сможем гарантировать того, что мы обнаружили дефект.

В результате проведения интеграционного тестирования и устранения всех выявленных дефектов получается согласованная и целостная архитектура программной системы, т.е. можно считать, что интеграционное тестирование – это тестирование архитектуры и низкоуровневых функциональных требований.

Интеграционное тестирование, как правило, представляет собой итеративный процесс, при котором проверяется функциональной все более и более увеличивающейся в размерах совокупности модулей.

7.2. Организация интеграционного тестирования

7.2.1. Структурная классификация методов интеграционного тестирования

Как правило, интеграционное тестирование проводится уже по завершении модульного тестирования для всех интегрируемых модулей. Однако это далеко не всегда так. Существует несколько методов проведения интеграционного тестирования:

- восходящее тестирование;
- монолитное тестирование;
- нисходящее тестирование.

Все эти методики основываются на знаниях об архитектуре системы, которая часто изображается в виде структурных диаграмм или диаграмм вызовов функций []. Каждый узел на такой диаграмме представляет собой программный модуль, а стрелки между ними представляют собой зависимость по вызовам между модулями. Основное различие методик интеграционного тестирования заключается в направлении движения по этим диаграммам и в широте охвата за одну итерацию.

Восходящее тестирование. При использовании этого метода подразумевается, что сначала тестируются все программные модули, входящие в состав системы и только затем они объединяются для интеграционного тестирования. При таком подходе значительно упрощается локализация ошибок: если модули протестированы по отдельности, то ошибка при их совместной работе есть проблема их интерфейса. При таком подходе у область поиска проблем у тестировщика достаточно узка, а поэтому гораздо выше вероятность правильно идентифицировать дефект.

Рис. 23 Разработка драйверов и заглушек при восходящем интеграционном тестировании

Однако, у восходящего метода тестирования есть существенный недостаток – необходимость в разработке драйвера и заглушек для модульного тестирования перед проведением интеграционного тестирования и необходимость в разработке драйвера и заглушек при интеграционном тестировании части модулей системы (Рис. 23)

С одной стороны драйверы и заглушки – мощный инструмент тестирования, с другой – их разработка требует значительных ресурсов, особенно при изменении состава интегрируемых модулей. Т.е. может потребоваться один набор драйверов для модульного тестирования каждого модуля, отдельный драйвер и заглушки для тестирования интеграции двух модулей из набора, отдельный – для тестирования интеграции трех модулей и т.п. В первую очередь это связано с тем, что при интеграции модулей отпадает необходимость в некоторых заглушках, а также требуется изменение драйвера, поддерживающее новые тесты, затрагивающие несколько модулей.

Монолитное тестирование предполагает, что отдельные компоненты системы серьезного тестирования не проходили. Основное преимущество данного метода – отсутствие необходимости в разработке тестового окружения, драйверов и заглушек. После разработки всех модулей выполняется их интеграция, после чего система проверяется вся в целом, как она есть. Этот подход не следует путать с системным тестированием,

которому посвящена следующая тема. Несмотря на то, что при монолитном тестировании проверятся работа всей системы в целом, основная задача этого тестирования – определить проблемы взаимодействия отдельных модулей системы. Задачей же системного тестирования является оценка качественных и количественных характеристик системы с точки зрения их приемлемости для конечного пользователя.

Монолитное тестирование имеет ряд серьезных недостатков:

- Очень трудно выявить источник ошибки (идентифицировать ошибочный фрагмент кода). В большинстве модулей следует предполагать наличие ошибки. Проблема выглядит как определение того, какая из ошибок во всех вовлечённых модулях привела к полученному результату. При этом возможно наложение эффектов ошибок. Кроме того, ошибка в одном модуле может блокировать тестирование другого.
- Трудно организовать исправление ошибок. В результате тестирования тестировщиком фиксируется найденная проблема. Дефект в системе, вызвавший эту проблему будет устранять разработчик. Поскольку, как правило, тестируемые модули написаны разными людьми, возникает проблема – кто из них является ответственным за поиск устранение дефекта? При такой «коллективной безответственности» скорость устранения дефектов может резко упасть.
- Процесс тестирования плохо автоматизируется. Преимущество (нет дополнительного программного обеспечения, сопровождающего процесс тестирования) оборачивается недостатком. Каждое внесённое изменение требует повторения всех тестов.

Нисходящее тестирование предполагает, что процесс интеграционного тестирования движется следом за разработкой. Сначала при нисходящем подходе тестируют только самый верхний управляющий уровень системы, без модулей более низкого уровня. Затем постепенно с более высокоуровневыми модулями интегрируются более низкоуровневые. В результате применения такого метода отпадает необходимость в драйверах (роль драйвера выполняет более высокоуровневый модуль системы), однако сохраняется нужда в заглушках (Рис. 24).

Рис. 24 Постепенная интеграция модулей при нисходящем методе тестирования

У разных специалистов в области тестирования разные мнения по поводу того, какой из методов более удобен при реальном тестировании программных систем. Йордан доказывает, что нисходящее тестирование наиболее приемлемо в реальных ситуациях [], а Майерс полагает, что каждый из подходов имеет свои достоинства и недостатки, но в целом восходящий метод лучше [].

В литературе часто упоминается метод интеграционного тестирования объектно-ориентированных программных систем, основанный на выделении кластеров классов, имеющих вместе некоторую замкнутую и законченную функциональность []. По своей сути такой подход не является новым типом интеграционного тестирования, просто меняется минимальный элемент, получаемый в результате интеграции. При интеграции модулей на процедурных языках программирования можно интегрировать любое количество модулей при условии разработки заглушек. При интеграции классов в кластеры существует достаточно нестрогое ограничение на законченность функциональности кластера. Однако, даже в случае объектно-ориентированных систем возможно интегрировать любое количество классов при помощи классов-заглушек.

Вне зависимости от применяемого метода интеграционного тестирования, необходимо учитывать степень покрытия интеграционными тестами функциональности системы. В работе [] был предложен способ оценки степени покрытия, основанный на управляющих вызовах между функциями и потоках данных. При такой оценки код всех модулей на структурной диаграмме системы должен быть выполнен (должны быть покрыты все узлы), все вызовы должны быть выполнены хотя бы раз (должны быть покрыты все связи между узлами на структурной диаграмме), все последовательности вызовов должны быть выполнены хотя бы один раз (все пути на структурной диаграмме должны быть покрыты) [].

7.2.2. Временная классификация методов интеграционного тестирования

На практике чаще всего в различных частях проекта применяются все рассмотренные в предыдущем разделе методы в совокупности. Каждый модуль тестируют по мере готовности отдельно, а потом включают в уже готовую композицию. Для одних частей тестирование получается нисходящим, для других – восходящим. В связи с этим представляется полезным рассмотреть еще один тип классификации типов интеграционного тестирования – классификацию по времени интеграции.

В рамках этой классификации выделяют:

- тестирование с поздней интеграцией;
- тестирование с постоянной интеграцией;
- тестирование с регулярной или послойной интеграцией.

Тестирование с поздней интеграцией – практически полный аналог монолитного тестирования. Интеграционное тестирование при такой схеме откладывается на как можно более поздние сроки проекта. Этот подход оправдывает себя в том случае, если система представляет собой конгломерат слабо связанных между собой модулей, взаимодействующих по какому-либо стандартному интерфейсу, определенному вне проекта (например, в случае, если система состоит из отдельных Web-сервисов). Схематично тестирование с поздней интеграцией может быть изображено в виде цепочки R-C-V-R-C-V-R-C-V-I-R-C-V-R-C-V-I, где R – разработка требований на отдельный модуль, C – разработка программного кода, V – тестирование модуля, I – интеграционное тестирование всего, что было сделано раньше.

Тестирование с постоянной интеграцией подразумевает, что, как только разрабатывается новый модуль системы, он сразу же интегрируется со всей остальной системой. При этом тесты для этого модуля проверяют как сугубо его внутреннюю функциональность, так и его взаимодействие с остальными модулями системы. Таким образом, этот подход совмещает в себе модульное тестирование и интеграционное. Разработки заглушек при таком подходе не требуется, но может потребоваться разработка драйверов. В настоящее время именно этот подход называют unit testing, несмотря на то, что в отличие от классического модульного тестирования здесь не проверяется функциональность изолированного модуля. Локализация ошибок межмодульных интерфейсов при таком подходе несколько затруднена, но все же значительно ниже, чем при монолитном тестировании. Большая часть таких ошибок выявляется достаточно рано именно за счет частоты интеграции и за счет того, что за одну итерацию тестирования проверяется сравнительно небольшое число межмодульных интерфейсов.

Схематично тестирование с постоянной интеграцией может быть изображено в виде цепочки R-C-I-R-C-I-R-C-I, в которой фаза тестирования модуля намеренно опущена и заменена на тестирование интеграции.

При **тестировании с регулярной или послойной интеграцией** интеграционному тестированию подлежат сильно связанные между собой группы модулей (слои), которые затем также интегрируются между собой. Такой вид интеграционного тестирования называют также *иерархическим интеграционным тестированием*, поскольку укрупнение интегрированных частей системы, как правило, происходит по иерархическому принципу. Однако, в отличие от нисходящего или восходящего тестирования, направление прохода по иерархии в этом подходе не задано.

Таблица 6 Основные характеристики различных видов интеграционного тестирования

Вид интеграции **Свойство**	**Восходящее**	**Нисходящее**	**Монолитное**	**Поздняя интеграция**	**Постоянная интеграция**	**Регулярная интеграция**
Время интеграции	поздно (после тестирования модулей)	рано (параллельно с разработкой)	поздно (после разработки всех модулей)	поздно (после разработки всех модулей)	рано (параллельно с разработкой)	рано (параллельно с разработкой)
Частота интеграции	редко	часто	редко	редко	часто	часто
Нужны ли драйверы	да	нет	нет	нет	да	да
Нужны ли заглушки	да	да	нет	нет	нет	да

В табл. 6 приведены основные характеристики рассмотренных в данной теме видов интеграционного тестирования. Время интеграции характеризует момент, когда проводится первое интеграционное тестирование и все последующие. Частота интеграции – насколько часто при разработке выполняется интеграция. Необходимость в драйверах и заглушках определена в последних двух строках таблицы.

7.2.3. Планирование интеграционного тестирования

Процесс организации и планирования интеграционного тестирования во многом схож с процессом организации модульного тестирования, рассмотренном в предыдущей теме. Однако интеграционное тестирование имеет ряд организационных особенностей, перечисленных ниже.

На этапе планирования разрабатывается концепция и стратегия интеграции – документ, который описывает общий подход к определению последовательности, в которой должны интегрироваться модули. Как правило, концепция основывается на одном из видов интеграции, рассмотренных выше (например, на нисходящей), но учитывает особенности конкретной системы (например, вначале должны интегрироваться компоненты работы с базой данных, затем пользовательского интерфейса, затем интерфейсные компоненты и компоненты работы с БД интегрируются вместе).

Составляется интеграционный тест-план, например, кластерного типа [,], в котором для каждого кластера из интегрированных модулей определяется следующее:

- кластеры, от которых зависит данный кластер;
- кластеры, которые должны быть протестированы до тестирования данного кластера;
- описание функциональности тестируемого кластера;
- список модулей в кластере;
- описание тестовых примеров для проверки кластера;

Планирование интеграционного тестирования должно быть синхронизировано с общим планом проекта, причем выделяемые для интеграционного тестирования кластеры и сроки их тестирования должны учитывать приоритеты важности частей системы. Зачастую рассмотрение приоритетов связано с тем, что системы разрабатываются в несколько этапов, на каждом из которых в эксплуатацию вводится только часть новой системы. Интеграционное тестирование в данном случае должно укладываться в общий план-график проекта и учитывать затраты ресурсов на тестирование интеграции с уже работающими частями системы.

ТЕМА 8.Системное тестирование (лекция 13)

8.1. Задачи и цели системного тестирования

По завершению интеграционного тестирования все модули системы являются согласованными по интерфейсам и функциональности. Начиная с этого момента можно переходить к тестированию системы в целом, как единого объекта тестирования – к *системному тестированию*. На уровне интеграционного тестирования тестировщика интересовали в основном

структурные аспекты системы, на уровне системного тестирования интересуют поведенческие аспекты системы. Как правило, для системного тестирования используется подход черного ящика, при этом в качестве входных и выходных данных используются реальные данные, с которыми работает система, или данные подобные им.

Системное тестирование – один из самых сложных видов тестирования. На этом этапе проводится не только функциональное тестирование, но и оценка характеристик качества системы – ее устойчивости, надежности, безопасности и производительности. На этом этапе выявляются многие проблемы внешних интерфейсов системы, связанные с неверным взаимодействием с другими системами, аппаратным обеспечением, неверным распределением памяти, отсутствием корректного освобождения ресурсов и т.п.

После завершения системного тестирования разработка переходит в фазу приемо-сдаточных испытаний (для программных систем, разрабатываемых на заказ) или в фазу альфа- и бета-тестирования (для программных систем общего применения).

Поскольку системное тестирование – процесс, требующих значительных ресурсов, для его проведения часто выделяют отдельный коллектив тестировщиков, а зачастую системное тестирование выполняется организацией, не связанной с коллективом разработчиков и тестировщиков, выполнявших работы на предыдущих этапах тестирования. При этом необходимо отметить, что при разработке некоторых типов программного обеспечения (например, авиационного бортового) требование независимого тестирования на всех этапах разработки является обязательным.

Системное тестирование проводится в несколько фаз, на каждой из которых проверяется один из аспектов поведения системы, т.е. проводится один из типов системного тестирования. Все эти фазы могут протекать одновременно или последовательно. Следующий раздел посвящен рассмотрению особенностей каждого из типов системного тестирования на каждой фазе.

8.2. Виды системного тестирования

Принято выделять следующие виды системного тестирования:

- функциональное тестирование;
- тестирование производительности;
- нагрузочное или стрессовое тестирование;
- тестирование конфигурации;
- тестирование безопасности;
- тестирование надежности и восстановления после сбоев;

- тестирование удобства использования.

В ходе системного тестирования проводятся далеко не все из перечисленных видов тестирования – конкретный их набор зависит от тестируемой системы.

Исходной информацией для проведения перечисленных видов тестирования являются два класса требований: функциональные и нефункциональные. Функциональные требования явно описывают, что система должна делать и какие выполнять преобразования входных значений в выходные. Нефункциональные требования определяют свойства системы, напрямую не связанные с ее функциональностью. Примером таких свойств может служить время отклика на запрос пользователя (например, не более 2 секунд), время бесперебойной работы (например, не менее 10000 часов между двумя сбоями), количество ошибок, которые допускает начинающий пользователь за первую неделю работы (не более 100) и т.п.

Рассмотрим каждый вид тестирования подробнее.

Функциональное тестирование. Данный вид тестирования предназначен для доказательства того, что вся система в целом ведет себя в соответствии с ожиданиями пользователя, формализованными в виде системных требований. В ходе данного вида тестирования проверяются все функции системы с точки зрения ее пользователей (как пользователей-людей, так и «пользователей»-других программных систем). Система при функциональном тестировании рассматривается как черный ящик, поэтому в данном случае полезно использовать классы эквивалентности. Критерием полноты тестирования в данном случае будет полнота покрытия тестами системных функциональных требований (или системных тест-требований) и полнота тестирования классов эквивалентности, а именно:

- все функциональные требования должны быть протестированы;
- все классы допустимых входных данных должны корректно обрабатываться системой;
- все классы недопустимых входных данных должны быть отброшены системой, при этом не должна нарушаться стабильность ее работы;
- в тестовых примерах должны генерироваться все возможные классы выходных данных системы;
- во время тестирования система должна побывать во всех своих внутренних состояниях, пройдя при этом по всем возможным переходам между состояниями.

Результаты системного тестирования протоколируются и анализируются совершенно аналогично тому, как это делается для модульного и интеграционного тестирования. Основная сложность здесь заключается в

локализации дефектов в программном коде системы и определении зависимостей одних дефектов от других (эффект «четного числа ошибок»).

Тестирование производительности. Данный вид тестирования направлен на определение того, что система обеспечивает должный уровень производительности при обработке пользовательских запросов. Тестирование производительности выполняется при различных уровнях нагрузки на систему, на различных конфигурациях оборудования. Выделяют три основных фактора, влияющие на производительность системы: количество поддерживаемых системой потоков (например, пользовательских сессий), количество свободных системных ресурсов, количество свободных аппаратных ресурсов.

Тестирование производительности позволяет выявлять узкие места в системе, которые проявляются в условиях повышенной нагрузки или нехватки системных ресурсов. В этом случае по результатам тестирования проводится доработка системы, изменяются алгоритмы выделения и распределения ресурсов системы.

Все требования, относящиеся к производительности системы должны быть четко определены и обязательно должны включать в себя числовые оценки параметров производительности. Т.е., например, требование «Система должна иметь приемлемое время отклика на запрос пользователя» является непригодным для тестирования. Напротив, требование «Время отклика на запрос пользователя не должно превышать 2 секунды» может быть протестировано.

То же самое относится и к результатам тестирования производительности. В отчетах по данному виду тестирования сохраняют такие показатели, как загрузка аппаратного и системного программного обеспечения (количество циклов процессора, выделенной памяти, количество свободных системных ресурсов и т.п.). Также важны скоростные характеристики тестируемой системы (количество обработанных в единицу времени запросов, временные интервалы между началом обработки каждого последующего запроса, равномерность времени отклика в разные моменты времени и т.п.).

Для проведения тестирования производительности требуется наличие генератора запросов, который подает на вход системы поток данных, типичных для сеанса работы с ней. Тестовое окружение должно включать в себя кроме программной компоненты еще и аппаратную, причем на таком тестовом стенде должна существовать возможность моделирования различного уровня доступных ресурсов.

Стрессовое тестирование. Стрессовое тестирование имеет много общего с тестированием производительности, однако его основная задача – не определить производительность системы, а оценить производительность и

устойчивость системы в случае, когда для своей работы она выделяет максимально доступное количество ресурсов, либо когда она работает в условиях их критической нехватки. Основная цель стрессового тестирования – вывести систему из строя, определить те условия, при которых она не сможет далее нормально функционировать. Для проведения стрессового тестирования используются те же самые инструменты, что и для тестирования производительности. Однако, например, генератор нагрузки при стрессовом тестировании должен генерировать запросы пользователей с максимально возможной скоростью, либо генерировать данные запросов таким образом, чтобы они были максимально возможными по объему обработки.

Стрессовое тестирование очень важно при тестировании web-систем и систем с открытым доступом, уровень нагрузки на которые зачастую очень сложно прогнозировать.

Тестирование конфигурации. Большинство программных систем массового назначения предназначено для использования на самом разном оборудовании. Несмотря на то, что в настоящее время особенности реализации периферийных устройств скрываются драйверами операционных систем, которые имеют унифицированный с точки зрения прикладных систем интерфейс, проблемы совместимости (как программной, так и аппаратной) все равно существуют.

В ходе тестирования конфигурации проверяется, что программная система корректно работает на всем поддерживаемом аппаратном обеспечении и совместно с другими программными системами.

В ходе тестирования конфигурации необходимо также проверять, что система продолжает стабильно работать при горячей замене любого поддерживаемого устройства на аналогичное. При этом система не должна давать сбоев ни в момент замены устройства, ни после начала работы с новым устройством.

Также необходимо проверять, что система корректно обрабатывает проблемы, возникающие в оборудовании, как штатные (например, сигнал конца бумаги в принтере), так и нештатные (сбой по питанию).

Тестирование безопасности. Если программная система предназначена для хранения или обработки данных, содержимое которых представляет собой тайну определенного рода (личную, коммерческую, государственную и т.п.), то к свойствам системы, обеспечивающим сохранение этой тайны будут предъявляться повышенные требования. Эти требования должны быть проверены при тестировании безопасности системы. В ходе этого тестирования проверяется, что информация не теряется, не повреждается, ее невозможно подменить, а также к ней невозможно получить

несанкционированный доступ, в том числе при помощи использования уязвимостей в самой программной системе.

В отечественной практике принято проводить сертификацию программных систем, предназначенных для хранения данных для служебного пользования, секретных, совершенно секретных и совершенно секретных особой важности. Существует ряд отечественных стандартов Федеральной службы по техническому и экспортному контролю (ФСТЭК), регламентирующих свойства программных систем по обеспечению необходимого уровня безопасности [] и по отсутствию недокументированных возможностей («закладок») [], которые могут быть использованы злоумышленником для несанкционированного доступа к данным. Кроме того, существует международный стандарт Common Criteria [], также регламентирующий вопросы защиты информации в программных системах.

Несмотря на то, что сертификация – процесс, следующий за верификацией (см. раздел 8.3, требования этих стандартов могут быть использованы и при тестировании системы. Так, стандарт [] ФСТЭК, традиционно сокращенно называемый РД СВТ выделяет следующие группы свойств программной системы, подлежащие проверке (некоторые группы свойств укрупнены для сокращения списка):

- разграничение и контроль доступа – предотвращение доступа к «чужой» информации;
- очистка и защита памяти – предотвращение доступа к остаточной информации после удаления объектов из памяти;
- маркировка и защита информации, передаваемой во внешний мир – сохранение уровня секретности даже вне системы;
- идентификация и аутентификация – предоставление доступа только санкционированным пользователям и отказ в доступе всем остальным;
- регистрация (аудит событий) – регистрация в специальном журнале всех событий системы, связанных с безопасностью для последующего анализа;
- гарантии проектирования и архитектуры – система должна быть спроектирована таким образом, чтобы гарантировать защищенность информации с определенным уровнем уверенности;
- тестирование – все функции по обеспечению безопасности должны быть протестированы во всех режимах;
- целостность и восстановление средств защиты – система должна иметь средства контроля корректности всех правил разграничения доступа и системы безопасности в целом, и средства их восстановления при сбое;
- документация разработчика, администратора и пользователя – все средства системы по обеспечению безопасности должны быть описаны в соответствующих руководствах.

При разработке и верификации программной системы, которая будет подвергаться последующей сертификации работы по сертификации должны включать в себя проверку всех перечисленных свойств.

Тестирование надежности и восстановления после сбоев. Для корректной работы системы в любой ситуации необходимо удостовериться в том, что она восстанавливает свою функциональность и продолжает корректно работать после любой проблемы, прервавшей ее работу (более подробно о классификации таких проблем – см. тему 10). При тестировании восстановления после сбоев имитируются сбои оборудования или окружающего программного обеспечения, либо сбои программной системы, вызванные внешними факторами. При анализе поведения системы в этом случае необходимо обращать внимание на два фактора – минимизацию потерь данных в результате сбоя и минимизацию времени между сбоем и продолжением нормального функционирования системы.

Тестирование удобства использования. Отдельная группа нефункциональных требований – требования к удобству использования пользовательского интерфейса системы. Этот вид тестирования будет рассмотрен в следующей теме.

В результате выполнения всех рассмотренных выше видов тестирования делается заключение о функциональности и свойствах системы, после чего узкие места системы дорабатываются до реализации необходимой функциональности или до достижения системой необходимых свойств.

8.3. Системное тестирование, приемо-сдаточные и сертификационные испытания при разработке сертифицируемого программного обеспечения

При разработке массового («коробочного», COTS) программного обеспечения после проведения системного тестирования система проходит этапы альфа- и бета-тестирования, во время которого работу системы проверяют потенциальные пользователи (либо специально выделенные фокус-группы пользователей, либо все желающие). На этом этапе в программную систему вносятся последние незначительные изменения, не влияющие на суть системы. После завершения этой стадии система поступает в продажу конечным пользователям.

При разработке заказного программного обеспечения фазу альфа- и бета-тестирования заменяют приемо-сдаточные испытания. Во время этих испытаний заказчик удостоверяется, что система работает в соответствии с его потребностями (как зафиксированными в техническом задании на систему, так и не зафиксированными). Заказчик может проводить такие испытания самостоятельно, выполняя заранее подготовленные тесты системы, либо проводить их совместно с представителями коллектива

разработчиков. В этом случае тестовые примеры также готовятся разработчиками, например, на основе тестовых примеров, использовавшихся на этапе системного тестирования.

Завершаются приемо-сдаточные испытания либо подписанием акта приемки, либо выдачей заказчиком дополнительных требований к системе, которые должны быть исправлены до приемки системы. После устранения всех недостатков системы приемо-сдаточные испытания повторяются (возможно, по сокращенной программе). После успешного подписания акта система поступает в эксплуатацию заказчику.

Существует специальный вид программных систем, к свойствам которых предъявляются особые требованиями. Примером таких систем могут служить бортовые авиационные программные системы, для которых особое внимание уделяется вопросам безопасности, надежности и отказоустойчивости. Несмотря на то, что большая часть таких систем может быть отнесена к категории заказного программного обеспечения, для получения разрешения на установку системы на борт требуется получение сертификата на летную пригодность.

Таким образом, после проведения системного тестирования и приемо-сдаточных испытаний проводятся сертификационные испытания. Сертификация программного обеспечения – процесс установления и официального признания того, что разработка ПО проводилась в соответствии с определенными требованиями. В процессе сертификации происходит взаимодействие заявителя, сертифицирующего органа и наблюдательного органа.

Заявитель – это организация, подающая заявку в соответствующий сертифицирующий орган на получения сертификата (соответствия, качества, годности и т.п.) изделия.

Сертифицирующий орган – организация, рассматривающая заявку заявителя о проведении сертификации ПО и либо самостоятельно, либо путем формирования специальной комиссии производящая набор процедур направленных на проведение процесса сертификации ПО заявителя.

Наблюдательный орган – комиссия специалистов, наблюдающих за процессами разработки заявителем сертифицируемой информационной системы и дающих заключение, о соответствии данного процесса определенным требованиям, которое передается на рассмотрение в сертифицирующий орган. []

Основной объект проверки в ходе сертификационных испытаний – соответствие процесса разработки программной системы регламенту и рекомендациям стандарта, на соответствие которому проводится

сертификация. Такое соответствие определяется при помощи анализа жизненного цикла сертифицируемой системы и документов, создаваемых на ключевых его этапах. Весь процесс анализа и те свойства системы, которые подвергаются сертификации, описывается в плане сертификационных испытаний, который утверждается совместно заявителем и сертифицирующим органом.

В случае сертификации бортовой системы по стандарту DO-178B (или его аналогам КТ-178, JB-12 и т.п.) план дополнительно определяет уровень влияния отказа программной системы на безопасность полета (уровень отказобезопасности) по которому будет проводиться сертификация. Любые вопросы, которые возникают у сертифицирующего органа относительно содержания плана сертификационных испытаний, должны быть разрешены до начала самих испытаний.

Согласно требованиями DO-178B план сертификационных испытаний (план программных аспектов сертификации) должен включать:

- обзор системы: Этот раздел описывает систему, включая описание ее функций и их размещение в программном и аппаратном обеспечении, ее архитектуру, используемый процессор (процессоры), аппаратно-программный интерфейс, и особенности отказобезопасности;
- обзор программного обеспечения: Этот раздел коротко описывает функции программного обеспечения с акцентом на концепцию обеспечения отказобезопасности и разделения на обособленные части, например, распределение ресурсов, резервирование, несимметрично резервированное программное обеспечение, устойчивость к отказам, стратегии таймирования и диспетчеризирования;
- сертификационные соображения: Этот раздел содержит сводку сертификационного базиса, включая средства подтверждения соответствия, как это определяется программными аспектами сертификации. В этом разделе также заявляется предложенный уровень (уровни) программного обеспечения и приводятся подтверждения правильности этого уровня, полученные в процессе оценки отказобезопасности системы, включая потенциальный вклад программного обеспечения в отказные ситуации;
- жизненный цикл программного обеспечения: Этот раздел определяет жизненный цикл программного обеспечения, который будет использоваться, а также включает сводку его процессов, детальная информация о которых определяется в соответствующих планах программного обеспечения. В сводке разъясняется, как будут удовлетворяться цели каждого процесса жизненного цикла, указываются вовлекаемые организации, организационная ответственность, а также ответственность за процессы жизненного

цикла системы и за процесс поддержания контактов в ходе сертификации;
- данные жизненного цикла программного обеспечения: Этот раздел определяет данные жизненного цикла, которые будут выпущены и будут контролироваться в процессах жизненного цикла программного обеспечения. Этот раздел также описывает взаимосвязь данных между собой или с другими данными, определяющими систему, данные жизненного цикла программного обеспечения, представляемые сертифицирующим властям, форму данных, и средства, с помощью которых данные жизненного цикла программного обеспечения могут быть сделаны доступными для сертифицирующих властей;
- план-график: Этот раздел описывает средства, которые заявитель будет использовать для того, чтобы обеспечить для сертифицирующих властей обозримость деятельности в процессах жизненного цикла программного обеспечения и, следовательно, возможность планирования проверок;
- дополнительные соображения: Этот раздел описывает особенности, которые могут повлиять на процесс сертификации, например, альтернативные методы подтверждения соответствия, квалификацию инструментальных средств, ранее разработанное программное обеспечение, вариантное программное обеспечение, которое может быть выбрано по желанию, программное обеспечение, доступное для модификации пользователем, готовое программное обеспечение COTS, используемое без модификаций, программное обеспечение, загружаемое в полевых условиях, несимметрично резервированное программное обеспечение или использование истории эксплуатации продукта.

В процессе самих сертификационных испытаний заявитель предоставляет свидетельства того, что процессы жизненного цикла программного обеспечения удовлетворяют планам программного обеспечения. Заявитель организует доступ сертифицирующего органа к данным жизненного цикла программного обеспечения. При этом минимальный перечень этих данных включает в себя:

- план сертификационных испытаний (план программных аспектов сертификации);
- индекс конфигурации программного обеспечения – документ, который должен однозначно идентифицировать каждый компонент проекта (включая требования, исходные коды, объектный и исполняемый код), среду реализации системы, инструкции по компиляции системы, аппаратное и программное обеспечение для работы системы, аппаратное и программное обеспечение для проведения сертификации.
- итоговое заключение о программном обеспечении.

Итоговое заключение по программному обеспечению является основным документом для демонстрации соответствия программного обеспечения Плану программных аспектов сертификации. Итоговое заключение должно включать:

- обзор системы: Этот раздел содержит обзор системы, включая описание ее функций и их размещения в аппаратном и программном обеспечении, архитектуру, используемый процессор (процессоры), аппаратно-программный интерфейс, средства обеспечения отказобезопасности. В этом разделе также описываются все отличия от описания системы, ранее помещенного в план программных аспектов сертификации.
- обзор программного обеспечения: Этот раздел кратко описывает функции программного обеспечения, особое внимание уделяется используемой концепции отказобезопасности и разделения на обособленные части, а также разъясняет отличия от обзора программного обеспечения, ранее помещенного в план программных аспектов сертификации.
- сертификационные соображения: Этот раздел повторно формулирует сертификационные соображения, приведенные в плане программных аспектов сертификации, а также описывает любые отличия от ранее приведенных соображений.
- характеристики программного обеспечения: Этот раздел констатирует данные о размере исполняемого кода, запасах по времени и памяти, ограничениях ресурсов, а также описывает средства для измерения каждой характеристики.
- жизненный цикл программного обеспечения: Этот раздел суммирует реальный жизненный цикл (циклы) программного обеспечения и разъясняет отличия от жизненного цикла программного обеспечения и процессов жизненного цикла, ранее предложенных в плане программных аспектов сертификации.
- данные жизненного цикла программного обеспечения: Этот раздел дает ссылку на данные жизненного цикла программного обеспечения, продуцируемые в процессах разработки программного обеспечения и процессах обеспечения целостности. Он описывает взаимосвязь данных между собой и с другими данными, определяющими систему, и средства, с помощью которых к данным жизненного цикла программного обеспечения может быть обеспечен доступ со стороны сертифицирующих властей. Этот раздел также описывает любые отличия от описания данных жизненного цикла, ранее помещенного в плане программных аспектов сертификации.
- дополнительные соображения: Этот раздел суммирует вопросы, которые могут привлечь внимание сертифицирующих властей, и дает ссылки на данные, применимые к этим вопросам, такие как выпущенные документы или специальные условия.

- идентификация программного обеспечения: Этот раздел идентифицирует конфигурацию программного обеспечения по номенклатурному номеру или версии.
- история изменений: Этот раздел, если это применимо, включает сводку изменений программного обеспечения, особое внимание уделяется изменениям, сделанным для исправления ошибок, влияющих на отказобезопасность, а также идентификацию изменений в процессах жизненного цикла программного обеспечения со времени предыдущей сертификации.
- статус программного обеспечения: Этот раздел содержит сводку сообщений о проблемах, не разрешенных на момент сертификации, включая заявления о функциональных ограничениях.
- заявление о соответствии: Этот раздел включает заявление о соответствии программного обеспечения настоящему документу, а также сводку методов, использованных для демонстрации соответствия с указанием критериев, специфицированных в планах программного обеспечения. В этом разделе также указываются дополнительные, по отношению к планам программного обеспечения, стандартам и настоящему документу, использованные правила и отклонения от планов, стандартов и настоящего документа.

Полный перечень данных жизненного цикла, которые могут понадобиться при сертификации включает в себя:

- план программных аспектов сертификации;
- план разработки программного обеспечения;
- план верификации программного обеспечения;
- план управления конфигурацией программного обеспечения;
- план гарантии качества программного обеспечения;
- стандарты на требования к программному обеспечению;
- стандарты проектирования программного обеспечения;
- стандарты на код программного обеспечения;
- данные требований на программное обеспечение;
- описание проекта;
- исходный текст;
- исполняемый объектный код;
- тестовые примеры и тестовые процедуры верификации программного обеспечения;
- отчет по результатам верификации программного обеспечения;
- индекс конфигурации окружающей среды жизненного цикла программного обеспечения;
- индекс конфигурации программного обеспечения;
- сообщения о проблемах;
- документы по управлению конфигурацией программного обеспечения;

- документы по гарантии качества программного обеспечения;
- итоговое заключение по программному обеспечению.

Сертифицирующий орган устанавливает т.н. сертификационный базис для системы в ходе консультаций с заявителем. Сертификационный базис определяет конкретные правила вместе с любыми специальными условиями, которые могут дополнять опубликованные правила сертификации, регламентированные стандартом.

Для программного обеспечения установка базиса производится по рассмотрению итогового заключения о программном обеспечении и свидетельств соответствия.

В ходе сертификации сертифицирующий орган оценивает план программных аспектов сертификации на полноту и согласованность с критериями оценки отказобезопасности системы и другими данными жизненного цикла программного обеспечения. Если верны все данные жизненного цикла, являющиеся доказательством того, что в ходе проекта были активны все необходимые процессы разработки и верификации, то сертифицирующий орган выдает положительное решение о выдаче сертификата.

Сертификаты на программное обеспечение можно отнести к двум типам: сертификаты соответствия и сертификаты качества.

- **Сертификат качества** – свидетельство, удостоверяющее качество фактически поставленного товара и его соответствие условиям договора. В сертификате качества дается характеристика товара либо подтверждается соответствие товара определенным стандартам или техническим условиям заказа. Сертификат качества выдается компетентными организациями, торговыми палатами, специальными лабораториями как в стране экспорта, так и импорта. Стороны договора купли-продажи могут договориться о предоставлении сертификатов различных контрольных и проверочных учреждений.
- **Сертификат соответствия** – результат действий третей стороны (документ), доказывающий, что обеспечивается необходимая уверенность в том, что должным образом идентифицированная продукция, процесс или услуга соответствуют конкретному стандарту или другому нормативному документу.

Сертификат на летную пригодность в рассматриваемом примере сочетает в себе свойства обоих типов сертификатов. С одной стороны он удостоверяет, что разработанная система имеет определенный уровень качества реализации, а с другой – что процессы по ее разработке соответствуют международному авиационному отраслевому стандарту.

ТЕМА 9. Тестирование пользовательского интерфейса (лекция 14)

9.1. Задачи и цели тестирования пользовательского интерфейса

Часть программной системы, обеспечивающая работу интерфейса с пользователем – один из наиболее нетривиальных объектов для верификации. Нетривиальность заключается в двоякости восприятия термина «пользовательский интерфейс».

С одной стороны пользовательский интерфейс – часть программной системы. Соответственно, на пользовательский интерфейс пишутся функциональные и низкоуровневые требования, по которым затем составляются тест-требования и тест-планы. При этом, как правило, требования определяют реакцию системы на каждый ввод пользователя (при помощи клавиатуры, мыши или иного устройства ввода) и вид информационных сообщений системы, выводимых на экран, печатающее устройство или иное устройство вывода. При верификации таких требований речь идет о проверке функциональной полноты пользовательского интерфейса – насколько реализованные функции соответствует требованиям, корректно ли выводится информация на экран.

С другой стороны пользовательский интерфейс – «лицо» системы, и от его продуманности зависит эффективность работы пользователя с системой. Факторы, влияющие на эффективность работы, в меньшей степени поддаются формализации в виде конкретных требований к отдельным элементам, однако должны быть учтены в виде общих рекомендаций и принципов построения пользовательского интерфейса программной системы. Проверка интерфейса на эффективность человеко-машинного взаимодействия получила название проверки удобства использования (usability verification, в русскоязычной литературе в качестве перевода термина usability часто используют слово «практичность»).

В данной теме будут рассмотрены общие вопросы как функционального тестирования пользовательских интерфейсов, так и тестирования удобства использования.

9.2. Функциональное тестирование пользовательских интерфейсов

Функциональное тестирование пользовательского интерфейса состоит из пяти фаз:

- анализ требований к пользовательскому интерфейсу;
- разработка тест-требований и тест-планов для проверки пользовательского интерфейса;
- выполнение тестовых примеров и сбор информации о выполнении тестов;

- определение полноты покрытия пользовательского интерфейса требованиями.
- составление отчетов о проблемах в случае несовпадения поведения системы и требований, либо в случае отсутствия требований на отдельные интерфейсные элементы.

Все эти фазы точно такие же, как и в случае тестирования любого другого компонента программной системы. Отличия заключаются в трактовке некоторых терминов в применении к пользовательскому интерфейсу и в особенностях автоматизированного сбора информации на каждой фазе.

Так, тест-планы для проверки пользовательского интерфейса как правило представляют собой сценарии, описывающие действия пользователя при работе с системой. Сценарии могут быть записаны либо на естественном языке, либо на формальном языке какой-либо системы автоматизации пользовательского интерфейса. Выполнение тестов при этом производится либо оператором в ручном режиме, либо системой, которая эмулирует поведение оператора.

При сборе информации о выполнении тестовых примеров как правило применяются технологии анализа выводимых на экран форм и их элементов (в случае графического интерфейса) или выводимого на экран текста (в случае текстового), а не проверка значений тех или иных переменных, устанавливаемых программной системой.

Под полнотой покрытия пользовательского интерфейса понимается то, что в результате выполнения всех тестовых примеров каждый интерфейсный элемент был использован хотя бы один раз во всех доступных режимах.

Отчеты о проблемах в пользовательском интерфейсе могут включать в себя как описания несоответствий требований и реального поведения системы, так и описания проблем в требованиях к пользовательскому интерфейсу. Основной источник проблем в этих требованиях – их тестонепригодность, вызванная расплывчатостью формулировок и неконкретностью.

9.2.1. Проверка требований к пользовательскому интерфейсу

9.2.1.1. Типы требований к пользовательскому интерфейсу

Требования к пользовательскому интерфейсу могут быть разбиты на две группы:

- требования к внешнему виду пользовательского интерфейса и формах взаимодействия с пользователем;

- требования по доступу к внутренней функциональности системы при помощи пользовательского интерфейса.

Другими словами, первая группа требований описывает взаимодействие подсистемы интерфейса с пользователем, а вторая – с внутренней логикой системы.

К первой группе можно отнести следующие типы требований:

- **Требования к размещению элементов управления на экранных формах**

Данные требования могут определять общие принципы размещения элементов пользовательского интерфейса или требования к размещению конкретных элементов. Например, общие требования по размещению элементов на графической экранной форме могут выглядеть следующим образом:

Каждое окно приложения должно быть разбито на три части: строка меню, рабочая область и статусная строка. Строка меню должна быть горизонтальной и прижатой к верхней части окна, статусная строка должна быть горизонтальной и прижата к нижней части окна, рабочая область должна находиться между строкой меню и статусной строкой и занимать всю оставшуюся площадь окна.

При тестировании данного требования достаточно определить, что в каждом окне системы действительно присутствуют три части, которые расположены и прижаты согласно требованиям даже при изменении размеров окна, его сворачивании/разворачивании, перемещении по экрану, при перекрытии его другими окнами.

Примером требований по размещению конкретного элемента может служить следующее:

Кнопка «Начать передачу» должна находиться непосредственно под строкой меню в левой части рабочей области окна.

При тестировании такого требования также необходимо определить, сохраняется ли расположение элемента при изменении размера окна, а также при использовании элемента (в данном случае – при нажатии).

- **Требования к содержанию и оформлению выводимых сообщений**

Требования к содержанию и оформлению выводимых сообщений определяют текст сообщений, выводимых системой, его шрифтовое и цветовое оформление. Также часто в таких требованиях определяется, в каких случаях выводится то или иное сообщение.

Так, например, для тестирования требования

Сообщение «Невозможно открыть файл» должно выводиться в статусную строку прижатым к левому краю красным цветом полужирным шрифтом в случае недоступности открываемого файла по чтению.

необходимо проверить, что при возникновении указанной ситуации сообщение действительно выводится согласно требованиям.

Однако в случае тестирования требования вида

Сообщения об ошибках должны выводиться в статусную строку прижатыми к левому краю красным цветом полужирным шрифтом.

необходимо проверять форматы всех возможных сообщений об ошибках программы во всех возможных ошибочных ситуациях. Таким образом, можно видеть, что при тестировании пользовательского интерфейса не всегда можно однозначно определить количество тестовых примеров, которые понадобятся для тестирования требования. Эта проблема вызвана тем, что требования к пользовательскому интерфейсу зачастую кажутся слишком очевидными для их точной формулировки. Эта неконкретность требований и вызывает большое количество тестов для каждого требования.

- **Требования к форматам ввода**

Данная группа требований определяет, в каком виде информация поступает от пользователя в систему. При этом кроме собственно требований, определяющих корректный формат, к этой группе относятся требования, определяющие реакцию системы на некорректный ввод. Для проверки таких требований необходимо проверять как корректный ввод, так и некорректный. Желательно при этом разбивать различные варианты ввода на классы эквивалентности (как минимум на два – корректные и некорректные).

Ко второй группе относятся следующие типы требований:

- **Требования к реакции системы на ввод пользователя**

Данный тип требований определяет связь внутренней логики системы и интерфейсных элементов. Например,

При нажатии кнопки «Сброс» значение таймера синхронизации передачи должно сбрасываться в 0.

Для проверки такого требования в тестовом примере должно быть сымитировано нажатие на кнопку «Сброс», после чего должна проводиться проверка значения таймера. Однако некоторые требования определяют в качестве реакции системы не то, как меняется ее внутреннее состояние, а реакцию пользовательского интерфейса. Например, в требовании

При нажатии кнопки «Отложенный сброс» должно выводиться окно «Ввод значения времени для отложенного сброса».

в качестве реакции на использование одного интерфейсного элемента определяется появление другого интерфейсного элемента. Такие требования проверяются при помощи имитации ввода пользователя и анализа появляющихся интерфейсных элементов.

- **Требования к времени отклика на команды пользователя**

В качестве отдельного типа требований можно выделить требования к времени отклика системы на различные пользовательские операции. Это связано с тем, что подсознательно пользователь воспринимает операции продолжительностью более 1 секунды как длительные. Если в этот момент система не сообщает пользователю о том, что она выполняет какую-либо операцию, пользователь начнет считать, что система зависла или работает в неверном режиме. В связи с этим либо все предельные времена отклика должны быть указаны в требованиях и пользовательской документации, либо во время длительных операций должны выводиться информационные сообщения (например, индикатор прогресса). Значения предельного времени и равномерность увеличения значений индикатора прогресса должны проверяться соответствующими тестами.

9.2.1.2. Тестопригодность требований к пользовательскому интерфейсу

Некоторые требования к пользовательскому интерфейсу могут оказаться тестонепригодными, либо их тестирование будет значительно затруднено. К таким требованиям в первую очередь относятся требования, описывающие субъективные характеристики интерфейса, которые не могут быть точно определены или измерены при выполнении тестовых примеров. При анализе требований к пользовательскому интерфейсу необходимо четко представлять, какой элемент интерфейса и каким образом будет проверяться, какая его характеристика будет измеряться в ходе тестирования.

Примером тестонепригодного требования может служить классическое требование

Пользовательский интерфейс должен быть интуитивно понятным.

Без определения четких критериев интуитивной понятности проверка такого требования невозможна. При этом необходимо понимать, что критерий в данном случае может быть двух видов: детерминированным или вероятностным. Примером детерминированного критерия может быть дополнение к требованию вида

Под интуитивной понятностью интерфейса понимается доступность любой функции системы при помощи не более чем 5 щелчков мыши по интерфейсным элементам.

Требование с таким поддается как ручному, так и автоматизированному тестированию, более того, результат такого тестирования не будет зависеть от субъективного мнения тестировщика (понятия об интуитивной понятности у всех разные).

Примером вероятностного критерия может служить следующее дополнение:

Под интуитивной понятностью интерфейса понимается, что пользователь обращается к руководству пользователя не чаще, чем раз в пять минут на этапе обучения и не чаще, чем раз в 2 часа на этапе активного использования системы. Значения должны быть получены на репрезентативной выборке пользователей не менее 1000 человек.

Проверка требования с таким дополнением не является задачей классической верификации и относится уже скорее к проверке удобства использования пользовательского интерфейса (см. раздел 9.3). Однако здесь также вводится четкий критерий, при использовании которого результаты тестирования могут быть воспроизведены.

9.2.2. Полнота покрытия пользовательского интерфейса

При определении понятия покрытия пользовательского интерфейса можно ввести следующие его уровни:

- **функциональное покрытие** – покрытие требований к пользовательскому интерфейсу;
- **структурное покрытие** – для обеспечения полного структурного покрытия каждый интерфейсный элемент должен быть использован в тестовых примерах хотя бы один раз;
- **структурное покрытие с учетом состояния элементов интерфейса** – для обеспечения этого уровня покрытия необходимо не только использовать каждый элемент интерфейса, но и привести его во все возможные состояния (например, для чек-боксов – отмечен/не отмечен,

для полей ввода – пустое/заполненное не целиком/заполненное полностью и т.п.)

- **структурное покрытие с учетом состояния элементов интерфейса и внутреннего состояния системы** – поведение некоторых интерфейсных элементов может изменяться в зависимости от внутреннего состояния системы. Каждое такое различимое поведение интерфейсного элемента должно быть проверено. Например, система может иметь два режима работы – нормальный и для начинающего пользователя, в котором нажатие каждого элемента сопровождается появлением всплывающей подсказки. В этом случае нужно проверить оба режима, при этом проверить, что подсказки появляются только в режиме для начинающих.

При определении степени покрытия необходимо учитывать, что реакция на некоторые интерфейсные элементы определяется не программной системой, а на уровне операционной системы или среды выполнения. Так, например, реакция на использование многих интерфейсных элементов стандартного диалогового окна открытия файла определяется операционной системой и может не тестироваться.

Если уровень покрытия интерфейсных элементов тестами недостаточен, то это является либо сигналом к уточнению требований к пользовательскому интерфейсу, либо к снижению степени подробности тестирования.

9.2.3. Методы проведения тестирования пользовательского интерфейса, повторяемость тестирования пользовательского интерфейса

Функциональное тестирование пользовательского интерфейса может проводиться различными методами – как вручную при непосредственном участии оператора, так и при помощи различного инструментария, автоматизирующего выполнение тестовых примеров. Рассмотрим эти методы более подробно.

9.2.3.1. Ручное тестирование

Ручное тестирование пользовательского интерфейса проводится тестировщиком-оператором, который руководствуется в своей работе описанием тестовых примеров в виде набора сценариев. Каждый сценарий включает в себя перечисление последовательности действий, которые должен выполнить оператор и описание важных для анализа результатов тестирования ответных реакций системы, отражаемых в пользовательском интерфейсе. Типичная форма записи сценария для проведения ручного тестирования – таблица, в которой в одной колонке описаны действия (шаги сценария), в другой – ожидаемая реакция системы, а третья предназначена

для записи того, совпала ли ожидаемая реакция системы с реальной и перечисления несовпадений (Таблица 7).

Таблица 7 Пример сценария для ручного тестирования пользовательского интерфейса

№ п/п	Действие	Реакция системы	Результат
1	Щелкните на пиктограмме System и выберите пункт меню ‘System Management Applet’.	Появится окно ввода логина и пароля	Верно
2	Введите в появившееся окно ввода имя пользователя ‘guest1’ и пароль ‘guest’. Затем нажмите кнопку ‘Login’.	Появится окно ‘System Management Applet’. В верхнем правом углу должно быть выведено имя вошедшего пользователя guest1.. Все опции в окне должны быть отключены (выведены серым цветом).	Неверно Окно имеет название ‘System Management Application’
3	Завершите сеанс работы с апплетом щелчком по пиктограмме ‘Logout’.	Окно ‘System Management Applet’ должно быть закрыто.	Верно

Ручное тестирование пользовательского интерфейса удобно тем, что контроль корректности интерфейса проводится человеком, т.е. основным «потребителем» данной части программной системы. К тому же при чисто косметических изменениях в интерфейсах системы, не отраженных в требованиях (например, при перемещении кнопок управления на 10 пикселей

влево) анализ успешности прохождения теста будет выполняться не по формальным признакам, а согласно человеческому восприятию.

При этом ручное тестирование имеет и существенный недостаток – для его проведения требуются значительные человеческие и временные ресурсы. Особенно сильно этот недостаток проявляется при проведении регрессионного тестирования и вообще любого повторного тестирования – на каждой итерации повторного тестирования пользовательского интерфейса требуется участие тестировщика-оператора. В связи с этим в последнее десятилетие получили распространение средства автоматизации тестирования пользовательского интерфейса, снижающие нагрузку на тестировщика-оператора.

9.2.3.2. Сценарии на формальных языках

Естественный способ автоматизации тестирования пользовательского интерфейса – использование программных инструментов, эмулирующих поведение тестировщика-оператора при ручном тестировании пользовательского интерфейса.

Такие инструменты используют в качестве входной информации сценарии тестовых примеров, записанные на некотором формальном языке, операторы которого соответствуют действиям пользователя – вводу команд, перемещению курсора, активизации пунктов меню и других интерфейсных элементов.

При выполнении автоматизированного теста инструмент тестирования имитирует действия пользователя, описанные в сценарии, и анализирует интерфейсную реакцию системы. При этом для определения ожидаемого состояния пользовательского интерфейса могут использоваться различные методы – либо анализ снимков экрана и сравнение их с эталонными, либо доступ к данным интерфейсных элементов средствами операционной системы (например, доступ ко всем кнопкам окна по их дескрипторам и получение значений текста).

И при передаче информации в тестируемый интерфейс и при получении информации для анализа могут использоваться два способа доступа к элементам интерфейса:

- позиционный, при котором доступ к элементу осуществляется при помощи задания его абсолютных (относительно экрана) или относительных (относительно окна) координат и размеров;
- по идентификатору, при котором доступ к элементу осуществляется при помощи получения интерфейсного элемента при помощи его уникального идентификатора в пределах окна.

При внесении изменений в пользовательский интерфейс при использовании первого метода в результате проведения регрессионного тестирования будет выявлено большое количество не прошедших тестов – достаточно изменения местоположения одного ключевого интерфейсного элемента, как все сценарии начнут работать неверно. Соответственно при таком методе автоматизации тестирования необходимо менять значительную часть сценариев в системе тестов при каждом изменении интерфейса системы. Такой метод автоматизации тестирования подходит для систем с устоявшимся и редко изменяемым интерфейсом.

Второй метод автоматизации тестирования более устойчив к изменению расположения интерфейсных элементов, но изменения тестовых примеров могут потребоваться и здесь в случае изменения логики работы интерфейсных элементов. Например, пусть в первой версии системы при нажатии на кнопку «Передать данные» передача данных начиналась сразу и выводилось окно с индикатором прогресса. Сценарий тестового примера в этом случае включает в себя имитацию нажатия на кнопку и обращение к индикатору прогресса для получения значения прогресса в процентах.

Если во второй версии системы после нажатия на кнопку «Передать данные» вначале выводится окно «Вы уверены?» с кнопками «Да» и «Нет», то для проверки работы индикатора прогресса в тестовый сценарий необходимо добавить имитацию нажатия кнопки «Да».

Даже при обращении при помощи идентификаторов без модификации тестового примера тест не будет корректно выполняться. Тем не менее, этот способ обращения к интерфейсным элементам хорошо подходит для тестирования программных систем, в т.ч. с не устоявшимся пользовательским интерфейсом.

9.3. Тестирование удобства использования пользовательских интерфейсов

Удобство использования пользовательского интерфейса (usability) – показатель его качества, определяющий 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. Данный раздел

курса не претендует на полноту изложения вопроса и дает самые общие представления о проблематике, связанной с тестированием удобства использования пользовательского интерфейса.

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, в процессе их работы с системой измеряются показатели их работы с системой (например, количество допущенных ошибок), а также им предлагается высказать собственные впечатления от работы с системой при помощи заполнения опросных листов.

Выделяют следующие этапы тестирования удобства использования пользовательского интерфейса []:

1. **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.
2. **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.
3. **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.
4. **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 в своей работе [] 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, она 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. Например, в работе [] приводится список контрольных вопросов, которые желательно проверять при тестировании удобства использования web-сайтов. С

некоторыми изменениями эти вопросы применимы и для обычных оконных интерфейсов.

ТЕМА 10.Методы разработки устойчивого кода (лекция 15)

10.1. Классификация проблем, возникающих при работе программных систем

Данная лекция посвящена внешним эффектам, проявляющихся в результате наличия дефектов в работающей программной системе. Эти эффекты различаются в первую очередь по степени серьезности последствий от проявления дефекта и времени нахождения системы в неработоспособном состоянии.

Первое проявление дефекта – сбой в работе системы. Сбои имеют небольшую продолжительность во времени и могут быть устранены без длительных процедур восстановления. Как правило, сбой вызывает либо кратковременную порчу данных пользователя без прекращения работы всей системы в целом. Последствия сбоя могут быть существенными с точки зрения пользователя, особенно если данные являются критически важными, однако бесперебойная работа системы не нарушается.

Отказ – более серьезное проявление дефекта в системе, при котором вся система или ее часть выходят из строя, выходя при этом из *работоспособного состояния*, т.е. состояния в котором все аспекты функционирования системы соответствуют требованиям. В случае отказа системы для ее возврата к нормальному функционированию требуется вмешательство оператора. Для программных систем причиной отказа может служить скрытый дефект, проявляющихся только с течением большого промежутка времени (переполнение внутреннего счетчика времени, переполнение данных и т.п.).

Авария – отказ системы, при котором система выходит из строя таким образом, что восстановление ее работоспособного состояния либо невозможно, либо занимает весьма значительное время. В случае программных систем можно избежать возникновения аварийных ситуаций при помощи полного дублирования системы как про выполняемому программному коду, так и по данным.

Сбои и отказы являются причиной отказных ситуаций, т.е. ситуаций в которых работоспособное состояние системы нарушается временно. Аварии являются причиной аварийных ситуаций, т.е. ситуаций, в которых работоспособное состояние системы нарушается навсегда или на длительный срок.

10.1.1. Сбои

Можно выделить следующие три вида сбоев, вызывающих отказные ситуации:

- **Сбои в системном программном обеспечении** – возникают при нештатном использовании системных средств – операционной системы, системы управления базами данных и т.п. Как правило, последствия данных сбоев наиболее тяжелые. В некоторых случаях возможна полная потеря, как данных системы, так и данных о состоянии системы на момент сбоя – дампов. Такие случаи наиболее сложны для диагностики и исправления.
- **Сбои в приложении** – возникают при недостаточном качестве тестирования прикладной системы, либо при нештатном ее использовании. Как правило, сбор информации о таких сбоях возможен средствами самого приложения. В критических случаях, например при полном крахе приложения возможен сбор о его информационном окружении средствами операционной системы, либо операционной среды, под управлением которой работает приложение.
- **Сбои - следствие неверной технологии использования** – возникают при неправильном (непредусмотренном) порядке действий пользователя при работе с системой. Сбои, наиболее сложные для анализа и устранения – их проявления могут заключаться не в отказах системы, а в ее действиях, неправильных или неочевидных с точки зрения пользователя. При этом не происходит автоматической рассылки информации разработчикам, единственная информация, на которую приходится опираться – обратная связь от пользователей. Устранение причин этих сбоев может вестись по нескольким направлениям. Следует отметить следующие: а) доработка руководства пользователя – не всегда эффективно, поскольку внимательно читает руководство лишь небольшое количество пользователей; б) привлечение к разработке специалиста по автоматизируемой предметной области и/или специалиста по эргономике – это позволит сделать пользовательский интерфейс системы более удобным и понятным.

Для классификации сбоев по категориям выделим следующие параметры сбоя:

Точка возникновения сбоя – строка или оператор программного кода, вызвавший отказную ситуацию. Данный оператор может находиться в коде системных библиотек, либо в коде приложения пользователя. Вовсе не обязательно, что сбой вызван именно этим оператором, но при помощи анализа окружения вызова и исходных текстов программ обычно удается обнаружить причину отказа.

Информационное окружение системы в момент сбоя – состояние системы в момент сбоя. К информационному окружению в данном случае относятся данные, которые могут помочь при анализе причины сбоя и его устранении – например, состояние стека, значение переменных окружения, пользовательской сессии и т.п. Данный набор параметров позволяет проследить ход выполнения программы, который привел к ее сбою и оценить несоответствия в данных, которые могли привести к сбою.

Наличие и тип сообщения о сбое – сообщение о сбое может быть создано автоматическим модулем оповещения о сбоях, и содержать вышеуказанную информацию, а может создаваться пользователем вручную. Если исходить из допущения, что автоматически создаваемые сообщения посылаются разработчику всегда в случае сбоев, не приводящих к полному краху системы и вызывающих нештатное поведение системы с точки зрения операционной среды, либо системы времени выполнения, то эта информация также помогает оценить тип сбоя.

В табл. 7 приведена классификация типов сбоев по указанным выше признакам:

Таблица 8 Классификация типов сбоев системы

Тип сбоя Параметры сбоя	**Сбой в системном ПО**	**Сбой в прикладном ПО**	**Сбой из-за неверной технологии**
Точка возникновения сбоя	Системные библиотеки или Приложение	Приложение	Неприменимо
Информационное окружение	Ненормальное	Ненормальное	Нормальное
Сообщение о сбое	Пользовательское или автоматическое	Автоматическое	Пользовательское

10.1.2. Отказы и аварии

Отказы вызывают длительное нарушение функционирования системы, или, по ГОСТ 27.002-89 [] приводят ее в *предельное состояние.* Предельное состояние – это состояние, при котором дальнейшая эксплуатация системы недопустима или нецелесообразна, либо восстановление ее работоспособного состояния невозможно или нецелесообразно. Тем самым в ГОСТ 27.002-89 не делается разницы между отказом и аварией. Будем называть отказом

состояние системы, при котором восстановление ее работоспособного состояния возможно.

Отказы классифицируются согласно ГОСТ 27.002-89 следующим образом:

По временным характеристикам:

Внезапный отказ – отказ, вызванный резким скачкообразным изменением одного из параметров системы или обрабатываемых системой данных. Ситуации, вызывающие такие отказы могут моделироваться при нагрузочном тестировании при помощи резкого повышения уровня нагрузки на систему (например, количества одновременно подключившихся пользователей) с последующей быстрой стабилизацией нагрузки.

Постепенный отказ – отказ, вызванный постепенным изменением одного из параметров системы или обрабатываемых системой данных. Такой отказ может возникать, например, при переполнении внутреннего буфера, хранящего информацию о состоянии системы в каждый момент времени. Если время работы системы больше размера буфера или не предусмотрена его очистка – рано или поздно возникнет переполнение.

Перемежающийся отказ – многократно возникающий самоустраняющийся сбой одного и того же характера. Поскольку в данном случае речь идет уже о систематически проявляющемся дефекте системы, то можно говорить именно об отказе, а не о серии сбоев.

Деградационный отказ – отказ, обусловленный естественным износом оборудования, на котором функционирует программная система, даже при соблюдении всех норм и правил проектирования, эксплуатации и сопровождения системы. Эти отказы не вызваны конструктивными дефектами системы, однако, для их предупреждения в состав системы может входить модуль мониторинга ее состояния, сообщающий о превышении степени износа частей системы. В случае, если планируется длительная эксплуатация системы, отсутствие требований и реализации такого модуля должно быть выявлено в процессе верификации.

По причинам:

Ресурсный отказ – отказ, в результате которого система достигает предельного состояния, т.е. такой отказ вызван в первую очередь нехваткой ресурсов (например, дискового пространства) для работы системы. Ситуации, вызывающие такие отказы, могут моделироваться при нагрузочном тестировании.

Конструктивный отказ – отказ, вызванный нарушением процесса проектирования и разработки системы или неверным проектированием.

Процесс верификации и тестирования в первую очередь направлен на обнаружение дефектов, вызывающих конструктивные отказы.

Производственный отказ – отказ, связанный с нарушением процесса производства или сопровождения системы. В применении к программным системам производственные отказы могут возникать в случае неверного выполнения профилактических работ при сопровождении системы. Например, в результате выполнения профилактических работ могут быть утеряны файлы настройки системы, в результате чего она переходит в режим работы по умолчанию, несовместимый с текущими настройками оборудования. Предупреждение таких отказов состоит в первую очередь в корректном составлении эксплуатационной документации и документации сопровождения, которая должна быть верифицирована.

Эксплуатационный отказ – отказ, связанный с нарушением правил эксплуатации. Причины, вызывающие данный вид отказов, связаны в первую очередь с человеческим фактором. Поэтому основные способы выявления таких отказов – проведение тестирования системы на удобство использования, верификация эксплуатационной документации, введение в систему защитных механизмов, блокирующих потенциальные ошибки оператора.

По способу обнаружения

Явный отказ – отказ, который обнаруживается сразу после его возникновения штатными средствами контроля состояния системы.

Скрытый отказ – отказ, который не обнаруживается штатными средствами контроля состояния системы, либо обнаруживается ими спустя некоторое время после возникновения отказа. Такой отказ может послужить причиной для одного или нескольких зависимых отказов.

По связи с другими отказами

Независимый отказ – отказ, возникновение которого не обусловлено другими отказами

Зависимый отказ – отказ, возникновение которого вызвано другими отказами

Процесс верификации не гарантирует отсутствия в системе всех дефектов, которые могут вызвать сбои, отказы или аварии – речь идет только об определенном уровне отсутствия этих дефектов. Поэтому когда речь идет о системах, к надежности которых предъявляются повышенные требования, для повышения их надежности кроме верификации используются различные методы разработки устойчивого кода.

При этом в результате верификации из системы устраняются дефекты, которые могут быть выявлены при анализе требований и/или кода, а методы разработки устойчивого кода дают дополнительную гарантию того, что система сохранит работоспособность в случаях, не предусмотренных требованиями. Однако, не следует расценивать эти методы как замену верификации или грамотного проектирования.

10.2. Методы разработки устойчивого кода

Как уже было сказано в первых разделах данного курса, верификация не может гарантировать того, что в программной системе отсутствуют абсолютно все ошибки. В лучшем случае мы можем сделать заключение о том, что система ведет себя в соответствии с требованиями, работая на заданном оборудовании. Для того, чтобы нивелировать возможные негативные последствия от ошибок, не обнаруженных в процессе верификации, применяются методы разработки устойчивого программного кода или методы *защитного программирования*.

Защитное программирование – это метод организации программного кода таким образом, чтобы при работе системы последствия проявления дефектов в ней не приводили к сбоям, отказам и авариям. При этом защитное программирование, как правило, не дает нам никакой информации о том, где в системе находится дефект, поэтому защитное программирование нельзя рассматривать как полную замену верификации – эти два аспекта промышленной разработки систем лишь дополняют друг друга.

Основной метод защитного программирования – внедрение в программный код системы различного рода проверок на допустимость обрабатываемых системой данных или допустимость состояния системы в заданный момент времени. Таким образом, подход защитного программирования можно сформулировать таким образом: «Прежде чем делать что-то – проверь, с корректными ли данными и в корректный ли момент времени ты начинаешь это делать». Если все данные для работы корректны – система функционирует в нормальном режиме. В случае, если данные неверны, запускается специально разработанная часть системы, предназначенная для восстановления правильности функционирования и предотвращения сбоя (либо при помощи приведения данных к корректному виду, либо при помощи извещения оператора).

В настоящее время существует два основных механизма защитного программирования – проверка допущений в критических точках и обработка исключительных ситуаций.

10.2.1. Критические точки и допущения (assertions)

В любой программе есть участки, требующие предварительных проверок перед выполнением. Например, представим себе два массива, один из которых хранит имена и фамилии людей, а второй – номера их телефонов. В нормальной ситуации количество элементов в обоих массивах должно совпадать и, при поиске фамилии в первом массиве, полученный индекс может быть использован для получения номера телефона найденного человека

```
string_vector surname;

string_vector phone;

…

int index = surname.find(“Петров”);

foundPhone = phone.at(index);
```

Однако, в случае рассинхронизации двух массивов, индекс может оказаться неверным, например, выходить за границы массива. Вообще, при использовании индекса, мы делаем неявное допущение о его корректности. В большинстве современных языков программирования (C, C++, C#, Java, Eiffel) существуют средства для явного задания таких допущений.

Например, в C существует функция assert(), определенная в заголовочном файле <assert.h>. Аргументом этой функции может выступать любое булево выражение. В случае, если оно равно false, функция прерывает работу программы. Таким образом, при помощи булевых выражений могут быть описаны допущения в критических точках программы. Предыдущий пример при использовании функции assert() будет выглядеть как

```
#include <assert.h>

…

string_vector surname;

string_vector phone;

…

int index = surname.find(“Петров”);

assert( (index > 0) && (index < phone.size()) );

foundPhone = phone.at(index);
```

Часто программисты определяют свою собственную функцию assert(), например, следующим образом:

```
#ifdef NODEBUG

#define assert(ignore) 0

#else

#define assert(ex) \

((ex) ? 1 : \

( printf("Assertion failed "), \

abort(), 0))

#endif // NODEBUG
```

Эта функция отличается от стандартной тем, что в финальной сборке системы с установленным макросом NODEBUG, выдача предупреждений функцией assert() отключается. Такая организация функции assert() связана с широко распространенным заблуждением касательно того, что частые вызовы функции assert() значительно замедляют выполнение программы. Поэтому многие программисты используют допущения только на стадии отладки, считая, что они не могут сработать в конечном продукте. Однако достаточно небольшой проигрыш в скорости окупается дополнительной гарантией надежности системы.

Другая причина того, что программисты предпочитают отключать допущения в финальной версии кода, заключается в том, что при срабатывании допущения выполнение программы прерывается. Однако, существует метод использования допущений совместно с обработкой исключений (см. следующий раздел), при котором возможно определить функции, исправляющие ошибочное состояние системы и продолжающие ее выполнение.

Существует три причины использовать допущения в критических точках:

- упрощение процесса создания корректных программ – явно задавая условия, необходимые для корректной работы программы в критических точках, мы защищаем себя от выдачи программой неверных данных;
- помощь в сопровождении документации и документировании – явно заданные условия позволяют проще определить, синхронизирован ли программный код с проектной документацией, которая зачастую

определяет корректные и некорректные диапазоны значений обрабатываемых данных;
- помощь в отладке – неверные допущения проявятся в процессе отладки, в результате упростится уточнение допущений.

В зависимости от того, в какой критической точке проверяется допущение, выделяют следующие их типы:

- *предусловия* – такие допущения помещаются в начале функций или процессов обработки данных и предназначены для проверки того, все ли необходимые данные корректны;
- *постусловия* – такие допущения помещаются в конце функций или процессов обработки данных и предназначены для проверки полученного результата на корректность до того, как передать его дальше;
- *инварианты классов* – такие допущения предназначены для периодической проверки состояния данных объектов классов, которые не должны меняться в течение жизненного цикла объекта, или должны меняться строго определенным образом;
- *инварианты циклов* – такие допущения предназначены для проверки условий, которые должны быть всегда истинны во время выполнения цикла. Примером такого условия может служить допущение о том, что значение итератора цикла не должно превышать количества элементов массива, по которому организован цикл.

Применение допущений является основной для контрактного программирования, при котором в каждом классе явно определяются предусловия условия для использования методов этого класса. Таким образом, между объектами этого класса и другими объектами заключается своего рода «контракт» в котором четко прописан интерфейс взаимодействия не только по форматам вызова методов, но и по пересылаемым данным.

Основным источником информации для определения контрактов являются требования на систему – функциональные или архитектурные. Поэтому верификацию допущений необходимо проводить в два этапа – на первом этапе проверяется корректность условий каждого допущения по отношению к требованиям, и только затем на втором этапе моделируются ситуации, приводящие к нарушению допущения. Начинающие тестировщики часто забывают о первом этапе, переходя от функционального тестирования по требованиям к структурному тестированию по коду.

10.2.2. Обработка исключений

В C++, C# и Java механизм допущений был значительно расширен. При возникновении неверных данных в критической точке вызывается не единая

общесистемная функция, прерывающая выполнение программы, а пользовательская функция, которая может либо предпринять попытку разрешить проблему с данными, либо прервать выполнение программы. Возникновение проблемы в критической точке получило название *исключительной ситуации* или *исключения*, а вызываемая функция пользователя – *обработчик исключения*.

Рассмотрим класс, реализующий тип данных «вектор». Для получения значения элемента вектора в нем перегружается оператор [], в котором делается проверка допустимости значения индекса элемента. В качестве второго параметра функции Assert указывается метод класса, обрабатывающего исключительную ситуацию в случае, если значение индекса элемента недопустимо.

```
class safeVector : public vector {

public:

class outOfRangeException {

int l;

public:

outOfRangeException (const char *,

const char *, int line)

: l (line) {}

int line (void) { return l; }

};

int& safeVector::operator[] (int index) {

Assert ((index >= 0) && (index < size),

safeVector::outOfRangeException);

...

};

int process(safeVector &v, int index) {

int elem;
```

```
try {
elem = v[index];
}
catch (safeVector::outOfRangeException &e) {
cerr << "Safe Vector range exception:\n";
exit (1);
}
return elem;
}
```

Для того, чтобы обработчик исключения сработал, обращение к элементу массива помещается внутрь структурного блока try { }, в котором содержатся команды, результат выполнения которых потенциально может вызвать исключительную ситуацию. Затем внутри синтаксического блока catch { } определяется реакция на возникшую исключительную ситуацию outOfRangeException. Подобным образом можно определить реакцию на любую исключительную ситуацию.

Все исключительные ситуации, для которых определяются пользовательские функции-обработчики, должны быть описаны в проектной документации и протестированы. Хорошей практикой является создание отдельной группы тестов, которая генерирует различные исключительные ситуации и проверяет реакцию системы на них. Однако следует учесть, что не все исключительные ситуации могут быть промоделированы при тестировании, особенно интеграционном или системном – некоторые обработчики пишутся «про запас» - на случай изменения других модулей системы и в реальности никогда не вызываются.

10.2.3. Сбор и обработка информации о сбоях и отказах

Информация о сбоях и отказах – такой же источник данных для программистов, как и информация о неуспешно пройденных тестах. Поэтому, при возникновении сбоя, отказа или аварии рекомендуется составлять отчет о проблеме, который будет передан коллективу разработчиков.

Отчет о проблеме, содержащий информацию о сбое, отказе или аварии будет несколько отличаться от отчета, созданного в ходе тестирования. Специфика

такого отчета заключается в том, что он либо составляется пользователем системы, не имеющим доступ к проектной документации и исходным текстам системы, либо специалистом службы технической поддержки со слов пользователя. Из-за этого в отчете может отсутствовать информация, необходимая для точной локализации вызвавшего проблему дефекта.

Для упрощения работы службы поддержки и программистов рекомендуется включать в отчет служебную информацию о состоянии и окружении системы в момент сбоя – переменных окружения, активных объектах, состоянии сеанса пользователя и т.п.

Поскольку напрямую пользователю эта информация недоступна, то ее сбор должен осуществляться обработчиками исключительных ситуаций. В результате их работы должен генерироваться выходной дамп состояния системы, прикладываемый к отчету, передаваемому разработчикам для анализа причин возникновения проблемы.

ТЕМА 11.Поддержка процесса тестирования при промышленной разработке программного обеспечения (лекция 16)

11.1. Управление качеством

11.1.1. Задачи и цели управления качеством

Промышленная разработка программного обеспечения, как часть промышленной разработки сложных систем – высокотехнологичный процесс, в который вовлечено множество различающихся по численности, сфере деятельности и квалификации коллективов разработчиков. Одна из основных задач при совместной работе большого количества разработчиков или их коллективов – обеспечение единой схемы работы, позволяющей планировать выполнение работ, обеспечивать целостность и непротиворечивость различных узлов системы, а также давать гарантии соответствия системы ожиданиям заказчика.

Выполнение этой задачи облегчается при подчинении процесса разработки технологическим требованиям, регламентирующим различные аспекты разработки: жизненный цикл проекта и разрабатываемой системы, требования к процессам разработки, внедрения и сопровождения системы.

При разработке систем различного типа (информационных, встроенных, систем реального времени, систем безопасности) технологические требования могут различаться и освещать те или иные аспекты процесса разработки. Тем не менее, обычно внедрение технологии в разработку преследует одну основную цель – обеспечение и гарантию качества разрабатываемой системы. Это послужило предпосылкой к

созданию документов, определяющих требования к технологии разработки систем – стандартов качества.

Согласно подходу стандартов системы качества: качество - это совокупность характеристик объекта, имеющая отношение к его способности удовлетворить установленные и предполагаемые требования потребителя. При этом, что важно, под объектом качества может пониматься как собственно продукция (товары или услуги), процесс ее производства, так и производитель (организация, система или даже отдельный работник).

В этих стандартах определяются критерии качества продукции, специфичные для каждого из этапов жизненного цикла продукции. Основное назначение стандартов качества – определение требований к технологическим процессам, в т.ч. процессам разработки, соблюдение которых позволяет гарантировать постоянный с точки зрения выбранных критериев уровень качества создаваемых систем (Рис. 25).

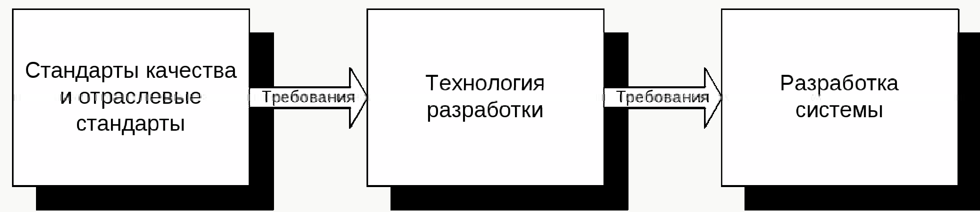


Рис. 25 Место стандартов качества в разработке системы

Неизбежные различия в процессах разработки и эксплуатации систем в разных отраслях послужили предпосылкой создания отраслевых стандартов – стандартов, регламентирующих процессы разработки и сертификации систем, предназначенных для узкоспециального использования (авиационные двигатели, системы информационной безопасности).

Поскольку область применения стандартов качества достаточно широка (от отдельной отрасли до общей применимости), основным объектом их рассмотрения являются не конкретные методики разработки, а общие технологические процессы.

Большинство стандартов являются ориентированными на процессы, не зависящие напрямую от конкретного жизненного цикла системы. Для каждого процесса определяются цели и описываются средства удовлетворения этих целей. Для данных жизненного цикла в стандартах представляется описание, которое показывает, что цели удовлетворены.

Основной процесс, рассматриваемый стандартами – выпуск продукции. В случае разработки программных систем этот процесс - проект разработки программного обеспечения, вне зависимости от того, какой жизненный цикл

программного обеспечения был выбран. Процесс разработки обычно распадается на более мелкие подпроцессы.

Подпроцессы могут быть отнесены к трем категориям: процесс планирования программного обеспечения, процессы разработки программного обеспечения, которые обычно включают разработку требований к программному обеспечению, проектирование программного обеспечения, кодирование и комплексирование программного обеспечения; и процессы обеспечения целостности, которые включают в себя верификацию программного обеспечения, гарантию качества программного обеспечения, управление конфигурациями программного обеспечения и взаимодействие при сертификации.

11.1.2. Система менеджмента качества по ISO 9000

Семейство стандартов ISO 9000 – группа международных стандартов, устанавливающих правила менеджмента качества при выпуске продукции. Отечественная группа стандартов, соответствующая международным ISO 9000 получила название ГОСТ Р ИСО 9000 []. Под понятие «выпуск продукции» попадает и разработка программного обеспечения.

При этом стандарты ISO 9000 проводят различие между требованиями к системам менеджмента качества и требованиями к продукции. Стандарт не гарантирует качество продукции – качество продукции в стандарте прямо не упоминается, тем самым он отличается от руководящих документов по проверке качества выпуска продукции различного рода (в т.ч. и программных систем).

Требования к системам менеджмента качества установлены в стандарте ISO 9001 (ГОСТ Р ИСО 9001). Они являются общими и применимыми к организациям в любых секторах промышленности или экономики независимо от категории продукции.

Требования к продукции могут быть установлены потребителями или организацией, исходя из предполагаемых запросов потребителей или требований регламентов. Требования к продукции и в ряде случаев к связанным с ней процессам могут содержаться, например в технических условиях, стандартах на продукцию, стандартах на процессы, контрактных соглашениях и регламентах.

Кратко повторимся, сделав важное уточнение - этот стандарт может быть сформулирован следующим образом: все процессы, которые могут существенно повлиять на качество готовой продукции, должны быть документированы, за выполнение этих правил должна быть назначена персональная ответственность, регулярно должна проводиться проверка соответствия реальных процессов документированным требованиям. Важно,

что обязательным требование является установление ответственности за качество процессов.

Итак "система качества" - это совокупность организационной структуры, методик, процессов и ресурсов, необходимых для общего руководства качеством.

В основе ISO 9000 лежат 8 принципов:

а) Ориентация на потребителя

Организации зависят от своих потребителей, и поэтому должны понимать их текущие и будущие потребности, выполнять их требования и стремиться превзойти их ожидания.

б) Лидерство руководителя

Руководители обеспечивают единство цели и направления деятельности организации. Им следует создавать и поддерживать внутреннюю среду, в которой работники могут быть полностью вовлечены в решение задач организации.

в) Вовлечение работников

Работники всех уровней составляют основу организации, и их полное вовлечение дает возможность организации с выгодой использовать их способности.

г) Процессный подход

Желаемый результат достигается эффективнее, когда деятельностью и соответствующими ресурсами управляют как процессом.

д) Системный подход к менеджменту

Выявление, понимание и менеджмент взаимосвязанных процессов как системы содействуют в результативности и эффективности организации при достижении её целей.

е) Постоянное улучшение

Постоянное улучшение деятельности организации в целом следует рассматривать как ее неизменную цель.

ж) Принятие решений, основанное на фактах

Эффективные решения основываются на анализе данных и информации.

и) Взаимовыгодные отношения с поставщиками

ISO 9000 определяет следующие основные процессы верхнего уровня:

- Система менеджмента качества
- Ответственность руководства
- Управление ресурсами
- Выпуск продукции
- Измерения, анализ, улучшения

Система менеджмента качества является основой основ при достижении показателей качества. Основной задачей системы менеджмента качества является определение процессов и их применение во всей организации, последовательность и взаимодействие процессов. Эффективность работы процессов достигается управлением необходимыми для процессов ресурсами и документами. Оценка эффективности ведется при помощи мониторинга, измерения и анализа заданных для процессов критериев результативности. Все требования системы менеджмента качества фиксируются в стандартах предприятия на соответствующие процессы.

Ответственность руководства с точки зрения ISO 9000 заключается в принятии обязательств по внедрению и поддержанию на предприятии системы менеджмента качества. Это достигается путем доведения до сведения сотрудников информации о необходимости достижения заданного качества продукции и обеспечение процессов, ориентированных на выпуск продукции, необходимыми ресурсами. Основные документы, разрабатываемые руководством – политика и цели в области качества, определяющие текущие и будущие пути развития предприятия для достижения необходимого уровня качества.

Стандарты качества также регламентируют требования к различным этапам процесса выпуска продукции, направленные на повышение ее качества. Основная суть этих требований заключается в обеспечении максимально прослеживаемого технологического процесса, задокументированного в стандартах предприятия.

11.1.3. Аудит процессов разработки и верификации

Аудит является процессом, позволяющим собирать данные о функционировании системы менеджмента качества для последующего их анализа с целью улучшения существующих на предприятии процессов. Аудиты проводятся периодически, согласно разработанной программе или внепланово, на основании анализа результатов проведенных аудитов.

Внутренние аудиты проводятся силами службы качества предприятия, а аудиторы назначаются из числа опытных сотрудников. Цель таких аудитов – текущий контроль системы менеджмента качества предприятия, позволяющий своевременно выявить проблемы.

Внешние аудиты проводят аудиторы из аудиторской компании, цель таких аудитов – подтверждение соответствия системы менеджмента качества предприятия требованиям стандарта качества.

В отличие от процесса верификации, который проверяет качество разрабатываемой программной системы, аудит в составе процесса управления качеством направлен на проверку технологических процессов – как разработки, так и верификации. Так, верификация отвечает на вопрос «разработана ли программная система в соответствии с требованиями?», а аудит менеджмента качества отвечает на вопрос «соответствовали ли процессы разработки и верификации установленным нормам и схемам технологических процессов, определенных в стандартах проекта и/или предприятия?».

Т.е. аудит качества процессов разработки – это тоже проверка соответствия, но в роли требований здесь выступают стандарты системы менеджмента качества, а в роли проверяемой системы – процессы.

Результаты аудитов используются в качестве одного из информационных потоков для проведения анализа результатов функционирования и соответствия процессов.

После проведения аудита, в случае выявления несоответствий (явно противоречащих стандартам качества случаев) и/или наблюдений (случаев, не противоречащим стандартам качества, но требующих модификации для повышения эффективности работы технологического процесса), предпринимаются корректирующие и/или предупреждающие действия.

11.1.4. Корректирующие действия и коррекция процессов

В соответствии с требованиями стандарта ISO 9001, предприятие, в случае возникновения несоответствий (т.е. не выполнения установленных требований к выпускаемой продукции, к системе менеджмента качества или к ее отдельным процессам) или предпосылок к их появлению, должно определить действия, которые позволят не только исправить данную проблему и устранить причины ее появления, но и предотвратить возможность возникновения ее в будущем посредством разработки и внедрения корректирующих и предупреждающих действий.

Основным документов, определяющим проблему является записка по качеству, идентифицирующая суть проблемы и сотрудника, выявившего проблему.

Как правило, для решения выявленных и классифицированных проблем применяются три типа мероприятий: коррекция, корректирующее действие и предупреждающее действие. Термин «коррекция» имеет отношение к ремонту, переделке или регулировке и относится к устранению имеющегося несоответствия. Термин «корректирующее действие» относится к устранению причины несоответствия.

Термин «предупреждающее действие» относится к устранению причин потенциального несоответствия, дефекта или другой нежелательной потенциально возможной ситуации с тем, чтобы предотвратить их возникновение.

Таким образом, пришедшая в процесс менеджмента качества записка по качеству либо закрывается, либо порождает корректирующее/предупреждающее действие (КД/ПД), а порой и коррекцию. В зависимости от уровня значимости, поиск решений проблемы может производиться либо на уровне процесса менеджмента качества, либо на уровне руководства предприятия, процесса или отдельного проекта. Заметим, что причин несоответствия может быть несколько и в ряде случаев единственным способом устранения является коррекция требований (стандартов).

Следует также заметить, что выполнение мероприятий по устранению причин несоответствий (КД/ПД) и самих несоответствий (коррекция) по сути являются такими же работами, как и остальные плановые мероприятия.

11.2. Конфигурационное управление

Процессы поддержания целостности, рассмотренные в разделе 11.1.1 остаются активными в ходе всего жизненного цикла программного обеспечения, в их число входит и процесс конфигурационного управления.

Основная задача данного процесса – обеспечение структурной целостности разрабатываемой системы. Все данные, входящие в проект, рассматриваются как единая конфигурация, структурная целостность которой достигается при помощи контроля всех входящих в нее компонент, обеспечения их физической сохранности, контроля и управления изменениями компонент системы.

Применение процесса конфигурационного управления является основным требованием при разработке систем, к их надежности (т.е., согласно ГОСТ 13377-75 [] - свойству объекта выполнять заданные функции, сохраняя во

времени значения установленных эксплуатационных показателей в заданных пределах, соответствующих заданным режимам и условиям использования, технического обслуживания, ремонта, хранения и транспортирования) которых предъявляются повышенные требования, в частности при разработке бортовых авиационных систем, информационных систем большого масштаба, систем безопасности (см., например, DO-178B []).

Основные задачи и цели процессов конфигурационного управления (КУ) состоят в следующем []:

- **Идентификация**: присвоение уникальных имен объектам конфигурационного управления (ОКУ). Каждый объект в конфигурации должен отличаться от всех других объектов. Данное требование не может быть удовлетворено созданием поисковых образов объектов (поиск документов по внутреннему содержимому), т.к. одно из применений идентификации – обеспечение трассируемости объектов. При помощи трасс между документами создаются логические ссылки, позволяющие проследить зависимости между различными группами проектной документации, покрытия требований и т.п.
- **Управление**: управление выпуском продукта и его изменениями в течение всего жизненного цикла. В данный набор целей входит предоставление данных, необходимых руководству для контроля изменений в данных, относящихся к выпускаемому продукту.
- **Вычисление статусов:** хранение и создание отчетов по состояниям объектов конфигурационного управления и запросов на изменение. Кроме вычисления статусов в виде отчетов, подобных индексам конфигурации, функция вычисления статусов должна позволять вычислять статус отдельно взятого ОКУ и определять жизненные циклы, являющиеся совокупностью статусов и правил их изменения.
- **Аудит и инспекции:** проверка завершенности и полноты продукта и поддержание целостности и непротиворечивости связей всех его компонент. Процесс КУ может регламентировать как процедуры проведения аудитов и их фазы, так и только точки жизненного цикла, в которых проводятся аудиты конфигурации.
- **Выпуск:** управление сборкой и построением окончательной версии продукта. В данном процессе обычно используются индексы конфигураций, в которых перечисляются ОКУ, необходимые в процессе сборки версии для той или иной целевой платформы.
- **Управление процессом:** проверка соблюдения организацией правил проведения процедур и модели жизненного цикла. Управление осуществляется в виде постоянного контроля за процедурами либо в рабочем порядке, либо в виде регулярных аудитов и позволяет удостовериться в технологичности процессов разработки, что в конечном итоге положительно сказывается на качестве продукции.

- **Коллективная работа:** управление работой и взаимодействием между множеством пользователей, работающих над продуктом, в том числе управление проектом и распределение задач между исполнителями. Также в процессе обеспечения коллективной работы должен проводиться контроль выполнения разработчиками поставленных перед ними задач путем отслеживания статуса документа.

Рассмотрим процесс конфигурационного управления в рамках документа DO-178B (Software Considerations in Airborne Systems and Equipment Certification), определяющего требования к процессам разработки авиационного бортового программного обеспечения. Требования этого стандарта к процессу конфигурационного управления являются достаточно жесткими и в целом сопоставимы с требованиями других стандартов (ISO 10007 [], IEEE 1042 []).

В стандарте DO-178B определяются шесть основных процессов программного проекта, из которых три можно отнести к производственным: планирование, разработка и верификация, а три к поддерживающим: обеспечение качества, взаимодействие с сертифицирующим органом и конфигурационное управление. Производственным процессам посвящены соответственно главы 4, 5 и 6.

Процесс конфигурационного управления рассматривается в седьмой главе документа. При этом некоторые его аспекты затрагиваются в четвертой главе, посвященной планированию, а некоторые в главе 11, посвященной данным процесса разработки.

11.2.1. Задачи процесса конфигурационного управления

Основной задачей процесса конфигурационного управления является обеспечение гарантии того, что организация – разработчик имеет все необходимые данные для подтверждения факта соответствия произведенного продукта требованиям.

Конфигурационное управление можно определить как процесс, с помощью которого руководство проекта имеет возможность на постоянной основе идентифицировать, устанавливать связи, сопровождать и управлять различными компонентами проекта. Этот процесс гарантирует целостность компонент и прослеживаемость всех изменений, возникающих в любой момент жизненного цикла проекта.

Базовым понятием процесса является объект (элемент) конфигурационного управления (ОКУ, Configuration Item). Под объектами конфигурационного управления могут пониматься все основные результаты деятельности проекта. Такие результаты идентифицируются и контролируются с помощью

процесса конфигурационного управления. Объектами конфигурационного управления могут быть элементы аппаратуры, программы, документация, процедуры и материалы обучения, средства обслуживания и т.д. В целях идентификации объектам конфигурационного управления могут быть присвоены номера.

Еще один термин, используемый в процессе конфигурационного управления – базовая конфигурация (Baseline). Под базовой конфигурацией (БК) понимается объект конфигурационного управления (отдельный элемент или совокупность элементов), который прошел процедуру утверждения и может быть изменен только в рамках процедуры управления изменениями.

Базовая конфигурация - это своего рода фотоснимок, «замороженная ситуация» требований, спецификаций или результатов, находящихся в разработке. Базовая конфигурация может быть представлена документом или набором документов. Создание базовой конфигурации обычно представляет собой фиксацию некоторого условия, возникающего при завершении каждого из основных шагов процесса разработки.

Базовая конфигурация может состоять из совокупности однородных документов, например, совокупность требований, коды совокупности программных модулей. Но базовая конфигурация может состоять и из совокупности разнородных по своей сути ОКУ. Например, требования и соответствующие программный код, тест-план, результаты прогона тест-плана.

Таким образом процесс конфигурационного управления призван обеспечивать:

- Объективность и контролируемость данных проекта;
- Доступность и восстанавливаемость данных проекта, включая объектные коды программ (например, объектный код может не хранится, но хранится исходный код и зафиксирована процедура трансляции);
- Контролируемость входных и выходных данных процедур проекта, обеспечивая их целостность и повторяемость;
- Точки контроля, возможность вычисления статуса конфигурации и управления изменениями через управление ОКУ и создание БК;
- Фиксацию проблем и отслеживание принятых по ним решений;
- Возможность прослеживания состояния проекта путем контроля выходных данных его жизненного цикла;
- Гарантии соответствия производимой продукции предъявляемым к ней требованиям;
- Ограничения доступа и сохранность ОКУ.

Сохранность ОКУ предполагает выполнение действий по архивированию данных, находящихся под опекой процесса конфигурационного управления, и проведение аудитов конфигураций. То есть данные не только должны быть ограждены от случайного (несанкционированнного) искажения, но и сохранены на случай выхода из строя средств хранения (пожары, затопления, разрушение носителей и т.п.).

Конфигурационное управление позволяет команде разработчиков программы или проекта точно определять статус любой компоненты во все время ее жизненного цикла и позволяет перевоссоздать любую версию в любой момент времени. Компонентами могут быть любые комбинации аппаратуры, программ, обслуживания и обучения.

11.2.2. Процедуры процесса конфигурационного управления

Конфигурационное управление построено как композиция нескольких подпроцессов, функционирующих совместно:

- Идентификация конфигураций;
- Управление изменениями;
- Формирование базовых конфигураций;
- Вычисление статусов конфигураций;
- Поддержка ограничений доступа;
- Архивирование, аудиты/обзоры конфигураций.

Следует особенно заметить, что процесс конфигурационного управления отнюдь не заканчивается с завершением работ проекта по производству продукции. Процесс конфигурационного управления продолжает функционировать и, возможно, весьма значительное время, обеспечивая поддержку сопровождения программного продукта.

11.2.2.1. Идентификация конфигураций

Целью процедуры идентификации является присвоение каждому ОКУ уникального имени (кода), обеспечивающего его опознание среди прочих ОКУ. Следует заметить, что процедура идентификации с очевидностью должна предшествовать процедуре прослеживания (трассировки). В тех случаях, когда идентификация объекта не может быть достигнута путем нанесения на него идентификационного кода (например, для объектного кода программы), она должна быть обеспечена косвенным путем, например, идентификационным полем, значение которого может быть проконтролировано вспомогательным программным средством.

Процедура КУ проекта должна устанавливать систему идентификации для каждой отдельно конфигурируемой компоненты программного обеспечения.

Поскольку сама компонента может при этом состоять из отдельных составных частей, то идентификация должна рекурсивно распространяться на все ее составные части до достижения уровня атомарного ОКУ (т.е. файла).

Идентификация ОКУ происходит путем помещения этих объектов в базу данных проекта. В самом простом случае база данных проекта может являться общедоступным сетевым каталогом на сервере, идентификация ОКУ и их версий при этом должна проводиться вручную.

Существует ряд систем конфигурационного управления, которые представляют собой инструменты для обеспечения коллективной работы с базой данных проекта. Эти системы берут на себя все основные функции конфигурационного управления, перечисленные выше, в т.ч. сохранность ОКУ, автоматическую нумерацию версий, предотвращение неавторизованных действий над ОКУ и учет состояния ОКУ.

Идентификатором ОКУ служит имя файла в совокупности с путем внутри базы данных. Имя файла присваивается менеджером конфигураций; он же имеет право переименовывать ОКУ.

Чтобы исключить возможность появления в базе данных проекта неправильно поименованных, неправильно размещенных и не подлежащих хранению в репозитории объектов, операции New Folder, Introduce и Rename могут выполнять только руководитель проекта и менеджер конфигураций.

Составные ОКУ идентифицируются путем составления индексов конфигураций, в которых перечисляются ОКУ (с указанием номера версии), входящие в состав конфигурации данного ОКУ. Индекс конфигурации, в свою очередь, является объектом конфигурационного управления и может входить как составная часть в другой ОКУ. Таким образом, ОКУ, в общем случае, образуют иерархию, вершиной которой является конфигурация продукта, включающая в себя все другие ОКУ.

11.2.2.2. Базовые конфигурации и прослеживаемость

Базовые конфигурации обычно используются как основа для перехода от одной процедуры жизненного цикла проекта к другой. В тот момент, когда процесс разработки переходит от одного шага к другому, специфические для этого шага результаты (документы, спецификации или продукты) инспектируют, чтобы убедиться в их качестве и связях (трассируемость) с предыдущими результатами. Специфические версии объектов конфигурации, принадлежащие им, идентифицируются. Когда (и если) результаты (выходы) шага прошли процедуру инспекции (только после этого), они могут быть объявлены базовой конфигурацией и становятся готовыми к использованию на следующем шаге процесса в качестве входа.

Принципиальным является то факт, что базовая конфигурация может изменяться только через процедуру Управления Изменениями. При этом требования DO-178B обязывают обеспечить прослеживаемость «происхождения» базовой конфигурации. Другими словами, должно быть обеспечено указание того, из какой предшествующей БК получена данная и с помощью какой процедуры.

В документе DO-178B используется единственный термин «трассируемость». Он используется и для обозначения ссылок от кода программы к требованиям и для указания на родительскую базовую конфигурацию. Возможно, что в целях более точной идентификации, следует в ряде случаев различать понятия прослеживаемость (эволюция БК) и трассируемость – связи разнотипных документов (отображение преобразования входа производственной процедуры в ее выход). Обычно под трассируемостью понимается возможность идентифицировать и историю, и текущее состояние (статус) каждого объекта конфигурации в любой точке жизненного цикла проекта. Необходимой также является и возможность трассировки объектов конфигурации относительно требований заказчика, как первичного входа проекта.

11.2.2.3. Управление изменениями

Большинство конфигурационных объектов в ходе жизненного цикла проекта претерпивают изменения. В процессе фиксации этих изменений возникает дерево версий, представляющее варианты объекта по степени его «завершенности». Каждая новая ветвь и лист такого дерева представляют новую версию ОКУ. Только последняя корректная версия любого объекта должна распространяться «по умолчанию» разработчикам в ответ на их запросы, устаревшие версии могут быть архивированы. При этом процедура архивирования должна обеспечивать восстанавливаемость (хранение или вычисление) любой запрошенной версии ОКУ.

Процедура управления изменениями определяет оценку и трассировку запросов на изменения, анализ потенциального влияния изменений и принятие решений по внесению изменений в объекты конфигурации. Эта процедура должна обеспечивать предотвращение «случайного» внесения изменения в базовую конфигурацию.

Реализация запросов на изменения должна включать трассировку данного изменения через процедуру фиксации проблемы, выработки и воплощения изменения, контроля его результатов и фиксации соответствующих данных жизненного цикла проекта.

Результаты процедуры внесения изменений должны инспектироваться, то собственно и приводит к возможности утверждения новой базовой конфигурации.

Версии (version) и Редакции (release): - эти термины иногда взаимозаменяемы. В данном документе термин "версия" используется прежде всего для ссылок на каждое новое проявление объекта конфигурационного управления, которому присвоен уникальный идентификационный номер, и подразумевает наличие цепочки ОКУ связанных отношением прослеживаемости. Другими словами одна версия ОКУ получается из другой через процедуру управления изменениями.

Редакцией здесь называется специфическая версия объекта конфигурации, предназначенная для "внешнего" использования. Как правило это внешнее использование – загрузка кода программы в целевой процессор или отгрузка результатов заказчику. Редакциями могут быть версии объектов, образующие комплект системы для внутреннего тестирования («лоады», «билды»). Такой объект (базовую конфигурацию) нельзя изменить. Она уже отослана и начинает жить «своей жизнью» вне проекта. Можно образовать новую редакцию и переслать ее заново.

11.2.2.4. Вычисление статуса конфигурации

После того, как изменение объекта конфигурации санкционировано, обычно возникает некоторая временная задержка на время реализации изменения. Руководству и участникам проекта необходимо иметь сведения о процессе внесения изменений, а не только о факте его завершения. Процедура вычисления статуса - механизм, используемый для прослеживания эволюции каждого объекта системы и его текущего состояния.

Процедура вычисления статуса должна обеспечивать возможность реководству проекта на основании отслеживания состояний отдельных ОКУ судит о состоянии разработки в целом. Эта процедура обеспечивает проектного менеджера большим количеством данных о его продукте, включая то, как он разрабатывается и все ли требуемые свойства действительно реализованы. В конечном счете, процедура обеспечивает актуальную и объективную информацию о статусе каждого объекта конфигурации в любой момент процесса разработки, которая включает данные следующих типов данных:

- Время возникновения каждой БК и изменения (проблемы);
- Время определения каждого объекта конфигурации;
- Описательная информация о каждом объекте конфигурации;
- Статус запросов на изменения (принят, отклонен, ожидает выполнения, выполнен)
- Описание статусов
- Описательная информация о каждом запросе на изменение
- Статус изменения
- Описательная информация о каждом изменении

Вычисление статуса, как функция увеличивает свою сложность по мере развития разработки. Эта сложность в основном выражается в быстром росте объемов данных, которые записываются и обрабатываются.

11.2.2.5. Архивирование, аудиты и обзоры конфигураций

Как уже говорилось, процедура архивирования должна обеспечивать восстанавливаемость (хранение или вычисление) любой запрошенной версии ОКУ.

Сохранность данных подразумевает не только возможность восстановления искомой версии ОКУ во все время действия процесса конфигурационного управления, но и защиту данных проекта от не санкционированного доступа. Иными словами изменения объекта могут производиться только тем лицом, которому это изменение доверено руководством проекта.

Менеджер проекта должен быть уверен, что требуемое управление конфигурацией реализуется - другими словами, все принятые изменения реализованы, а результат представляет собой то, что специфицировано в его проектной документации. Для достижения необходимо высокого уровня доверия он должен планировать регулярные аудиты и обзоры процесса конфигурационного управления и его данных.

Требования для этих аудитов и обзоров обычно специфицируются в плане конфигурационного управления, но также они могут быть заданы в планах проекта или плане гарантии качества, как это покажется подходящим. Ответственность за нормальную реализацию аудитов конфигураций лежит на менеджере конфигураций, если эта должность предусмотренав проекте, а если нет - на менеджере проекта или менеджере качества.

Аудитам конфигураций адресуются следующие вопросы, относящиеся к измененным объектам конфигурации:

- Проведены ли изменения так, как они специфицированы и проведены ли соответствующие технические инспекции?
- Выдержано ли следование соответствующим стандартам?
- Выдержано ли следование установленным процедурам управления конфигурациями для записи и выдачи отчетов?
- Модифицированы ли все связанные с изменением объекты конфигурации?

11.2.2.6. Управление инструментальными средствами

Инструментальные средства, используемые в проекте, должны храниться в репозитории проекта. Исключением могут быть инструментальные средства

внешнего происхождения (например, покупные или предоставленные заказчиком), хранение которых в репозитории может оказаться невозможным из-за их большого объема.

Хранению в репозитории проекта подлежат, как минимум, пользовательская документация, исполняемый код инструментального средства, а также иные файлы, необходимые для работы программного средства, такие как библиотеки, базы данных, параметры настройки и т.п. Если использованию инструментального средства должна предшествовать процедура установки или генерации, то хранению подлежат все файлы, необходимые для выполнения такой процедуры.

Кроме того, инструментальные средства собственной разработки должны храниться вместе с исходным кодом и проектной документацией, а также прочими данными, необходимыми для воспроизведения исполняемого кода данного инструментального средства.

Процедуры проекта должны точно идентифицировать конфигурацию используемых инструментальных средств. Модификация этой конфигурации допускается только через управление изменениями. Настройки СКУ должны предотвращать несанкционированное изменение инструментальных средств.

11.2.3. Уровни управления данными

Документ DO-178B в третьем разделе седьмой главы выделяет две категории управления данными жизненного цикла программного проекта. Они соответственно названы CC1 и CC2 категории управления данными. Выбор следования той или иной категории определяется приложением A. В упрощенном виде различие между категориями можно проследить по следующим соответствиям:

идентификация объектов (CC1,CC2)

Формирование базовых конфигураций (CC1)

обеспечение трассируемости (CC1,CC2)

фиксация проблем (CC1)

управление изменениями (фиксация данных) (CC1,CC2)

анализ изменений (CC1)

вычисление статуса конфигурации (CC1)

изменения (CC1,CC2)

обеспечение ограничений доступа (CC1,CC2)

контроль загрузочных модулей (CC1)

хранение данных (CC1,CC2)

11.3. Управление качеством и конфигурационное управление при разработке сертифицируемого программного обеспечения.

Стандарт DO-178B обеспечивает авиационное сообщество руководящими указаниями, предназначенными для установления в конструктивной манере и с приемлемым уровнем достоверности факта того, что программное обеспечение бортовых авиационных систем и оборудование согласуются с требованиями летной годности. [, раздел 1].

Процесс конфигурационного управления в данном стандарте рассмотрен в разделах 7 (Процесс конфигурационного управления при разработке программного обеспечения), 11.4 (План конфигурационного управления при разработке программного обеспечения) и 12.1.5 (Соображения о конфигурационном управлении при разработке программного обеспечения).

Основные цели процесса конфигурационного управления, согласно требованиям данного стандарта, включают в себя следующие положения [, раздел 7.1]:

- Поддержание четко определенной и управляемой конфигурации программного обеспечения в течение всего его жизненного цикла.
- Предоставление возможности повторяемой репликации исполняемого объектного кода, т.е. возможности заново сгенерировать объектный код с целью проверки или после модификации
- Обеспечение контроля входов и выходов процессов жизненного цикла с целью обеспечения целостности и повторяемости действий процессов
- Определение четких моментов времени для проведения инспекций, вычисления статусов и управления изменениями, достигаемое при помощи управления объектами конфигурациями и определения их базовых состояний.
- Предоставление средств и методик, дающих гарантию того, что все обнаруженные проблемы принимаются во внимание, а изменения протоколируются, санкционируются и реализовываются.
- Предоставление подтверждений санкционирования изменений в программном обеспечении при помощи управления выходами процессов жизненного цикла программного обеспечения.
- Помощь в оценке соответствия программного обеспечения требованиям

- Обеспечение надежного архивирования, восстановления и управления объектами конфигурации.

В целом процесс конфигурационного управления, охватываемый стандартом DO-178B, направлен на поддержание целостности данных, создаваемых в ходе всех стадий жизненного цикла продукции. Основная специфика процесса КУ, регламентируемого данным стандартом состоит в учете аспектов сертификации на летную годность, которую должно проходить все программное обеспечение, используемое в бортовых авиационных системах. Данные процесса КУ используются в качестве основных данных, предоставляемых сертифицирующему органу.

Сертифицирующим органам предоставляются индексы конфигураций – списки уникальным образом идентифицированных элементов (исходных текстов, файлов данных, объектного и исполняемого кода), входящих в программное обеспечение. Для подтверждения соответствия качества программного обеспечения заданному уровню критичности бортового ПО, представляются результаты его тестирования, проведенные в соответствии с требованиями к данному уровню. Процесс конфигурационного управления должен обеспечивать трассируемость всех требований, исходных текстов, тестов и их результатов.

Для проведения сертификации все элементы конфигурации должны иметь соответствующие пометки о готовности к сертификации, а все проблемы, возникающие в ходе разработки, должны быть тем или иным образом закрыты. Данный аспект сертификации обеспечивается предоставлением информации о ведении проблем и включением сообщений о проблемах и связанных с ними запросов на изменение в индекс конфигурации, подлежащей сертификации.

Все рассмотренные выше требования являются общими требованиям к процессу КУ, но процесс разработки и выполнения бортового ПО имеет одну особенность – средой разработки служат те или иные компьютеры общего назначения, а средой времени выполнения служит соответствующее системное ПО бортового вычислителя. Стандартом DO-178B выдвигаются требования, учитывающие эту особенность - существуют требования по обеспечению поддержки процесса загрузки объектного кода приложения на бортовой вычислитель, гарантирующего загрузку исполняемого кода с применением соответствующих мер безопасности, обеспечивающих его неизменность и целостность [, Раздел 4.2.8].

Таким образом, система конфигурационного управления, удовлетворяющая требованиям стандарта DO-178B должна обеспечивать идентификацию и трассируемость объектов конфигурационного управления (ОКУ), поддержку создания базовых версий (индексов конфигураций), управление

сообщениями и проблемах и вызванными ими изменениями, учет состояния конфигурации и ОКУ, а также обеспечивать процессы резервного копирования и восстановления данных и управление процессом загрузки объектного кода на бортовой вычислитель.

Библиографический список

1. А.Гаврилов. Технологии разработки программного обеспечения – MSF (/Rus/Msdnaa/Curricula/Default.mspx)
2. Г. Майерс. Надежность программного обеспечения. М.: «Мир», 1980. 360 с.
3. Александр Петренко, Елена Бритвина, Сергей Грошев, Александр Монахов, Ольга Петренко Тестирование на основе моделей // Открытые системы, #09/2003
4. Гостехкомиссия России. Руководящий документ. Защита от несанкционированного доступа к информации. Часть 1. Программное обеспечение средств защиты информации. Классификация по уровню контроля отсутствия недекларированных возможностей. М.: Гостехкомиссия РФ, 1999
5. Гостехкомиссия России. Руководящий документ. Средства вычислительной техники. Защита от несанкционированного доступа к информации. Показатели защищенности от несанкционированного доступа к информации. М.: Гостехкомиссия РФ, 1992
6. ГОСТ Р ИСО 9001-2001. Системы менеджмента качества. Требования. М.: ИПК Изд-во стандартов, 2001, – 140 с.
7. RTCA/DO-178B. Software considerations in airborne system and equipment certification, RTCA Inc, 1992, – 138 p.
8. AS9100A. Quality Management Systems – Aerospace – Requirements. G-14 Americas Aerospace Quality Group (AAQG), SAE, 2003, – 69 p.
9. AS9006A. Aerospace Software Supplement for AS9100A. G-14 Americas Aerospace Quality Group (AAQG), SAE, 2003, – 24 p.
10. Burnstein I. Practical Software Testing. A process-oriented approach. Springer-Verlag, New York, 2003, - 732 p.
11. Стивенс У. UNIX. Взаимодействие процессов. – СПб.: Питер, 2002. – 576 с.
12. Синицын С.В., Налютин Н.Ю. Операционные системы: учебное пособие. М.: МИФИ, 2006. 213 с.
13. Leblanc P., Encontre V. Object-oriented Real-time Techniques: Method Guide.
14. IEEE 1012-1998. IEEE Standard for Software Verification and Validation. Institute of Electrical and Electronics Engineers. 01-May-1998, 75 p.
15. IEEE 829-1998. IEEE Standard for Software Test Documentation. Institute of Electrical and Electronics Engineers. 01-May-1998, 62 p.

16. Dart S. Concepts in Configuration Management Systems. Proc. 3rd International Workshop on Software Configuration Management, Trondheim, Norway, 1991, – 166 p, pp. 1-18.
17. U. Linnenkugel, M. Mullerburg. Test data selection criteria for (software) integration testing. Proc. First International Conf. Systems Integration, April 1990, pp. 709–717.
18. Белов С. Модульное тестирование и Test-Driven Development, или Как управлять страхом в программировании // "IT News", #21/2005
19. IEEE 1008-1987. IEEE Standard for Software Unit Testing. Institute of Electrical and Electronics Engineers. 01-May-1987, 28 p.
20. Murphy G., Townsend P., Wong P. Experiences with cluster and class testing. Communications of the ACM, Vol. 37, No. 9, 1994, pp. 39–47.
21. ГОСТ 27.002-89. Надёжность в технике. Основные понятия. Термины и определения. – М.: Издательство стандартов, 1990. – 37 с.
22. ГОСТ 13377-75. Надежность в технике. Термины и определения. М.: Изд-во стандартов, 1975.
23. ISO 10007. Quality management - Guidelines for configuration management. International Organization for Standardization. 01-Apr-1995, 14 p.
24. IEEE 1042-1987. IEEE Guide to Software Configuration Management. Institute of Electrical and Electronics Engineers. 10-Sep-1987, 92 p.
25. An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion. Office of Aviation Research Washington, D.C., 2001, 214 p.
26. Hayhurst K.J. et al. A Practical Tutorial on Modified Condition/Decision Coverage. NASA, 2001, 85 p.
27. Э. Йордан С. Аргила. Структурные модели в объектно-ориентированном анализе и проектировании. М.:Лори, 1999, 288 с.
28. Майерс Г. Искусство тестирования программ. М.:Финансы и статистика, 1982, 176 с.
29. Rational Unified Process. Методологияитехнология. Материалы компании Interface Ltd. (http://www.interface.ru/home.asp?artId=779)
30. Nielsen J. Ten Usability Heuristics (/papers/heuristic/heuristic_list.html)
31. ISO 13407:1999. Human-centred design processes for interactive systems. International Organization for Standardization. 01-Jun-1999, 26 p.
32. ISO/IEC 9126-1:2001. Software engineering -- Product quality -- Part 1: Quality model. International Organization for Standardization/International Electrotechnical Commission. 01-Jun-2001, 25 p.
33. Общий оценочный лист тестирования usability web-сайта. Публикациякомпании IT-Online. (/lib/it-online/site-usability-checklist.htm)
34. КТ-178А. Квалификационные требования часть 178А, АОЗТ "ИСПАС", Жуковский, 1997.

35. Microsoft Solutions Framework. Методология создания программных решений. (/Rus/Msdn/msf/Default.mspx)
36. Бек К. Экстремальное программирование. С-Пб.:Питер, 2002, 224 с.
37. ISO/IEC 15408. Information technology – Security techniques – Evaluation criteria for IT security. International Organization for Standardization. 50 p. (part 1), 248 p. (part 2), 168 p. (part 3).