МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

ГЛАВНОЕ УПРАВЛЕНИЕ ПО ОБРАЗОВАНИЮ
МОГИЛЕВСКОГО ОБЛАСТНОГО ИСПОЛНИТЕЛЬНОГО КОМИТЕТА

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«МОГИЛЕВСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ КОЛЛЕДЖ»

УТВЕРЖДАЮ
Директор колледжа
_____ С.Н.Козлов
29.08.2018

ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ
ПО ИЗУЧЕНИЮ УЧЕБНОЙ ДИСЦИПЛИНЫ,
ЗАДАНИЯ НА ДОМАШНЮЮ КОНТРОЛЬНУЮ РАБОТУ
ДЛЯ УЧАЩИХСЯ ЗАОЧНОЙ ФОРМЫ ОБУЧЕНИЯ
ПО СПЕЦИАЛЬНОСТИ 2-40 01 01
«ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ»

2018

Автор: Кашпарова Ю.А., преподаватель учреждения образования «Могилевский государственный политехнический колледж»

Рецензент: Сенькевич Е.А., преподаватель учреждения образования «Могилевский государственный политехнический колледж»

Разработано на основе типовой учебной программы по учебной дисциплине «Тестирование и отладка программного обеспечения», утвержденной Министерством образования Республики Беларусь, 26.01.2017

Обсуждено и одобрено на
заседании цикловой комиссии
специальности «Программное
обеспечение информационных технологий»

Протокол № _____ от _____

Пояснительная записка

Программа учебной дисциплины «Тестирование и отладка программного обеспечения» предусматривает изучение видов, назначения и применения методов тестирования программного обеспечения.

Изучение учебной дисциплины базируется на знаниях, умениях, и навыках, полученных при изучении следующих учебных дисциплин: «Основы алгоритмизации и программирования», «Конструирование программ и языки программирования», «Структуры и алгоритмы обработки данных», «Технология разработки программного обеспечения».

Основной целью изучения учебной дисциплины «Тестирование и отладка программного обеспечения» является формирование профессиональной компетентности учащихся в вопросах тестирования и отладки программного обеспечения.

Основные задачи изучения учебной дисциплины:

– сформировать целостное представление о значении тестирования и отладки в современном мире;

– сформировать умения использовать составляющие современных информационных технологий: электронные таблицы, баг–трекинговые системы и др.;

– развивать познавательный интерес к тестированию и его применению, интеллектуальные и творческие способности учащихся.

В результате изучения учебной дисциплины учащиеся должны знать на уровне представления:

– виды, уровни, направления и методы тестирования;

– критерии выбора тестов и оценки качества программного обеспечения;

– понятие верификации программного обеспечения;

– особенности документирования дефектов с использованием систем отслеживания проблем;

знать на уровне понимания:

– значение основных терминов, используемых в области тестирования и отладки программного обеспечения;

– особенности проведения модульного, системного и интеграционного тестирования;

– требования к составлению отчетов об ошибках;

– особенности тестирования Web–приложений;

– основы тестирования безопасности, производительности, регрессивного тестирования;

– особенности выполнения автоматизированного тестирования;

уметь:

– проводить тестирование структуры программных модулей и их взаимодействия;

– проводить тестирование требований к программному обеспечению;

– выполнять разработку тестовых сценариев;

– составлять отчеты об ошибках;

– проводить отладку и функциональное тестирование Web–ориентированных приложений;

– использовать инструментальные средства при проведении автоматизированного тестирования и отладки программного обеспечения.

В целях контроля усвоения программного учебного материала для учащихся предусмотрено выполнение домашней контрольной работы и проведение обязательной контрольной работы.

Общие методические рекомендации по выполнению домашней контрольной работы

Учащиеся-заочники выполняют одну домашнюю контрольную работу. Основным методом изучения учебной дисциплины является самостоятельная работа, которая должна проводиться в последовательности, предусмотренной программой учебной дисциплины, и быть обязательно систематической.

Задания на домашнюю контрольную работу разработаны в количестве 100 вариантов в соответствии с программой курса. Номера заданий выбираются в соответствии с двумя последними цифрами шифра учащегося, на пересечении соответствующей строки с соответствующим столбцом из таблицы 3.

Заданием являются теоретические вопросы, на которые нужно дать развернутый ответ, привести примеры. Объем – около двух-трех страниц.

При оформлении домашней контрольной работы следует придерживаться следующих требований:

1 Работа выполняется на листах А4 машинописным способом (шрифт 12-14, межстрочный интервал - одинарный). Следует пронумеровать страницы и оставить на них поля: справа – не менее 3 см для замечаний преподавателя, остальные поля – 2,5 см.

2 На титульном листе указываются: шифр, специальность, фамилия, имя, отчество учащегося, учебная дисциплина и номер работы, номер группы.

3 Ответ следует начинать с номера и полного названия вопроса.

Критерии оценки домашней контрольной работы

Домашняя контрольная работа, признанная преподавателем удовлетворительной и содержащая 75% положенного объема, оценивается «зачтено».

Домашняя контрольная работа оценивается «не зачтено», если:

- выполнена не в соответствии с вариантом;
- не раскрыто основное содержание одного теоретического вопроса и есть незначительные недочеты в других заданиях;
- есть существенные недочеты в нескольких теоретических вопросах.

Программа учебной дисциплины и методические рекомендации по ее изучению

Введение

Содержание, цели и задачи учебной дисциплины «Тестирование и отладка программного обеспечения», ее значение в подготовке специалистов среднего звена, связь с другими учебными дисциплинами

Базовые теоретические понятия, которые лежат в основе отладки и тестирования программного обеспечения

Литература: [1]

Методические рекомендации

Под отладкой понимается процесс, позволяющий получить программное обеспечение, функционирующее с требующимися характеристиками в заданной области входных данных.

Отладка не является разновидностью тестирования, хотя слова «отладка» и «тестирование» часто используют как синонимы. Под ними подразумеваются разные виды деятельности:

- тестирование – деятельность, направленная на обнаружение ошибок;
- отладка направлена на установление точной природы известной ошибки, а затем – на исправление этой ошибки;
- результаты тестирования являются исходными данными для отладки.

Тестирование – это неотъемлемая часть (этап) разработки программных продуктов. Целью тестирования является обнаружение дефектов, проверка соответствия программы заявленным требованиям, а также предоставление обратной связи (в общем случае, обратная связь предоставляется в форме отчёта о дефектах) разработчикам, менеджерам и другим заинтересованным персонам. Тестирование выявляет проблемные места в разрабатываемом приложении, вследствие чего тестирование может и даже должно влиять на качество приложения в сторону улучшения.

Раздел 1 Основы тестирования программного обеспечения (ПО)

Тема 1.1 Тестирование как элемент жизненного цикла ПО. Дефекты ПО и их жизненный цикл

Эволюция методов тестирования ПО. Качество ПО

Модели жизненного цикла ПО и роль тестирования в них

Тестирование, верификация и валидация, их различия. Свойства тестирования. Взаимосвязь уровней и целей тестирования ПО

Типы дефектов и ошибок ПО, их жизненный цикл

Литература: [2]

Методические рекомендации

Жизненный цикл программного обеспечения - ряд событий, происходящих с системой в процессе ее создания и дальнейшего использования. А также это время от начального момента создания какого-либо программного продукта, до конца его разработки и внедрения. Жизненный цикл программного обеспечения можно представить в виде моделей.

Модель жизненного цикла программного обеспечения - структура, содержащая процессы действия и задачи, которые осуществляются в ходе разработки, использования и сопровождения программного продукта.

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. На рисунке 1 представлена взаимосвязь тестирования, валидации и верификации.

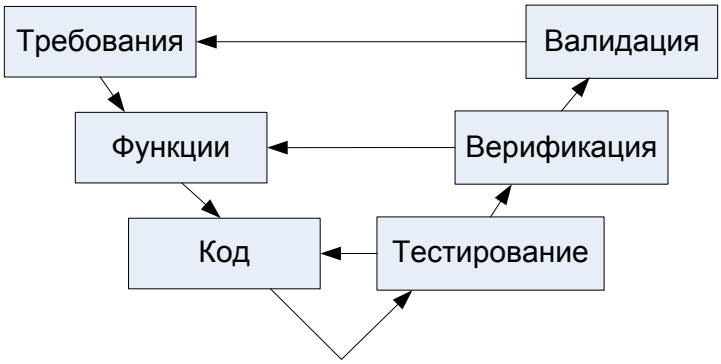

Рисунок 1 - Тестирование, верификация и валидация

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.

Ошибка (error) – это действие человека, которое порождает неправильный результат.

Однако программы разрабатываются и создаются людьми, которые также могут допускать (и допускают) ошибки. Это значит, что недостатки есть и в самом программном обеспечении. Они называются дефектами или багами (оба обозначения равносильны). Здесь важно помнить, что программное обеспечение – нечто большее, чем просто код.

Дефект, Баг (Defect, Bug) – недостаток компонента или системы, который может привести к отказу определенной функциональности. Дефект, обнаруженный во время исполнения программы, может вызвать отказ отдельного компонента или всей системы.

При исполнении кода программы дефекты, заложенные еще во время его написания, могут проявиться: программа может не делать то-

го, что должна или наоборот – делать то, чего не должна, – происходит сбой.

Сбой (failure) – несоответствие фактического результата (actualresult) работы компонента или системы ожидаемому результату (expectedresult).

Сбой в работе программы может являться индикатором наличия в ней дефекта.

Таким образом, баг существует при одновременном выполнении трех условий:

– известен ожидаемый результат;

– известен фактический результат.

– фактический результат отличается от ожидаемого результата.

Важно понимать, что не все баги становятся причиной сбоев – некоторые из них могут никак себя не проявлять и оставаться незамеченными (или проявляться только при очень специфических обстоятельствах).

Причиной сбоев могут быть не только дефекты, но также и условия окружающей среды: например, радиация, электромагнитные поля или загрязнение также могут влиять на работу как программного, так и аппаратного обеспечения.

Всего существует несколько источников дефектов и, соответственно, сбоев:

– ошибки в спецификации, дизайне или реализации программной системы;

– ошибки использования системы;

– неблагоприятные условия окружающей среды;

– умышленное причинение вреда;

– потенциальные последствия предыдущих ошибок, условий или умышленных действий.

Дефекты могут возникать на разных уровнях, и от того, будут ли они исправлены и когда, будет напрямую зависеть качество системы.

Качество (Quality) – степень, в которой совокупность присущих характеристик соответствует требованиям.

Качество программного обеспечения (Software Quality) – это совокупность характеристик программного обеспечения, отражающих его способность удовлетворять установленные и предполагаемые потребности.

Требование (Requirement) – потребность или ожидание, которое установлено. Обычно предполагается или является обязательным.

На рисунке 2 представлены уровни отслеживания дефектов.

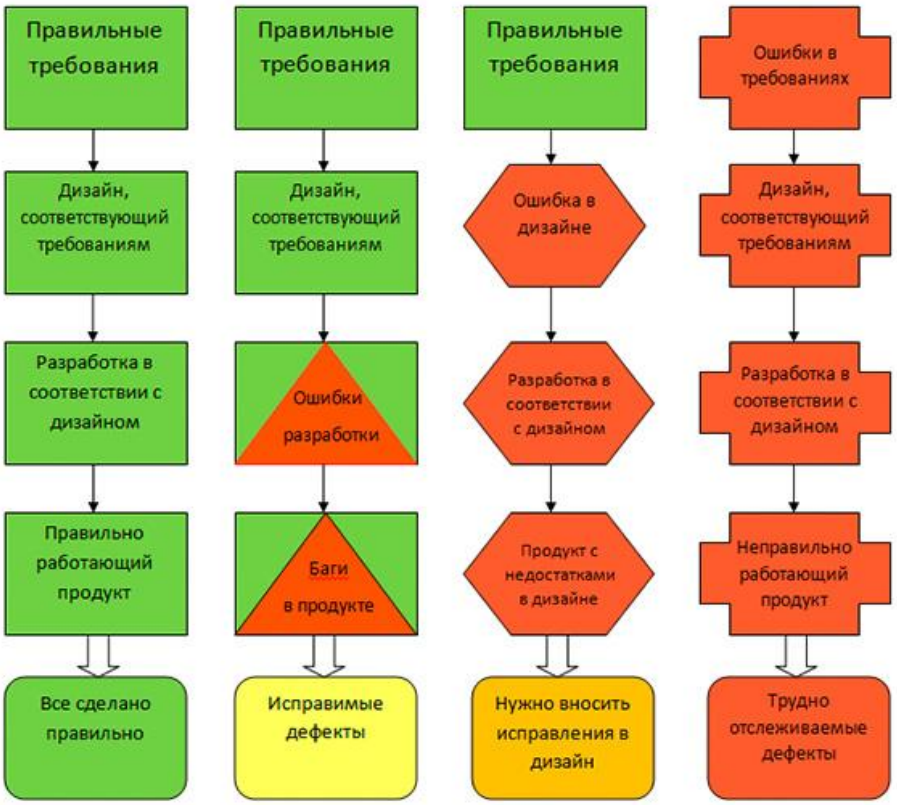

Рисунок 2 – Уровни отслеживания дефектов

В первом случае все было сделано правильно и мы получили продукт, полностью соответствующий ожиданиям заказчика и удовлетворяющий критериям качества.

Во втором случае ошибки были допущены уже при кодировании, что привело к появлению дефектов в готовом продукте. Но на этом уровне баги достаточно легко обнаружить и исправить, поскольку мы видим несоответствие требованиям.

Третий вариант хуже – здесь ошибки были допущены на этапе проектирования системы. Заметить это можно лишь проведя тщательную сверку со спецификацией. Исправить такие дефекты тоже непросто – нужно заново перерабатывать дизайн продукта.

В четвертом случае дефекты были заложены еще на этапе формирования требований; вся дальнейшая разработка и даже тестирование пошли по изначально неправильному пути. Во время тестирования мы не найдем багов – программа пройдет все тесты, но может быть забракована заказчиком.

Тема 1.2 Типы процессов тестирования

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

Модульное тестирование. Интеграционное тестирование. Системное тестирование. Нагрузочное тестирование. Формальные инспекции

Регрессионное тестирование. Комбинирование уровней тестирования

Литература: [3]; [4]

Методические рекомендации

Модульное тестирование. Модульное тестирование, или юнит-тестирование (англ. unit testing) - процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Модульный тест - это автоматизированный фрагмент кода, который вызывает тестируемый метод или класс, а затем проверяет несколько предположений относительно логического поведения метода или класса.

Модульное тестирование (Unit testing) – тестирование каждой атомарной функциональности приложения отдельно, в искусственно созданной среде. Именно потребность в создании искусственной рабочей среды для определенного модуля, требует от тестировщика знаний в автоматизации тестирования программного обеспечения, некоторых навыков программирования. Данная среда для некоторого юнита создается с помощью драйверов и заглушек.

Драйвер – определенный модуль теста, который выполняет тестируемый нами элемент.

Заглушка – часть программы, которая симулирует обмен данными с тестируемым компонентом, выполняет имитацию рабочей системы.

Заглушки нужны для:

– имитирования недостающих компонентов для работы данного элемента;

– подачи или возвращения модулю определенного значения, возможность предоставить тестеру самому ввести нужное значение;

– воссоздания определенных ситуаций (исключения или другие нестандартные условия работы элемента).

Преимущества модульного тестирования:

– модульное тестирование мотивирует программистов писать код максимально оптимизированным, проводить рефакторинг (упрощение кода программы, не затрагивая ее функциональность), так как с помо-

щью Unit-тестирования можно легко проверить работоспособность рассматриваемого компонента;

– необходимость отделения реализации от интерфейса (ввиду особенностей модульного тестирования), что позволяет минимизировать зависимости в системе;

– документация Unit-тестов может служить примером «живого документа» для каждого класса, тестируемого данным способом;

– модульное тестирование помогает лучше понять роль каждого класса на фоне всей программной системы.

Разработка через тестирование (англ. test-driven development, TDD)- техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам. TDD (сокр. от англ. test-driven development - «разработка через тестирование») - это специальная методика разработки ПО, которая основывается на коротких циклах работы, где сначала создаётся тест, а потом функционал.

Все определения, сводятся к цикличности и первичности тестов. Визуально это представлено на рисунке 3.

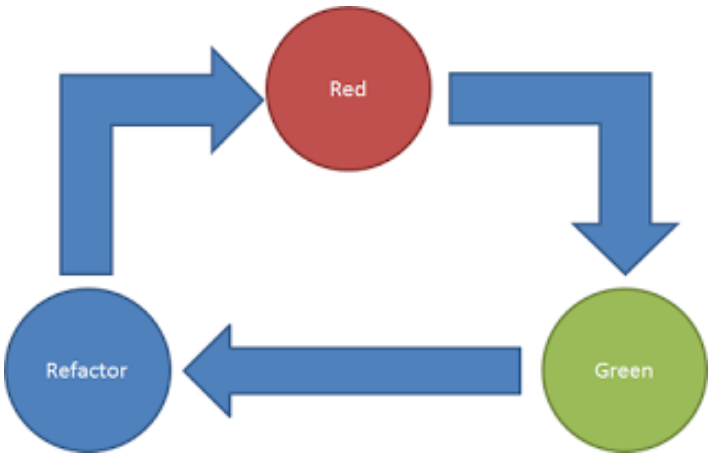

Рисунок 3 - Метод TDD

Красная зона, это зона написания нового теста, который проверяет функционал отсутствующий в приложении (тест не проходит, он красный). Затем идет зеленая зона, в рамках которой необходимо написать минимальное количества кода, чтобы тест стал «зеленым». Ну и если у нас получился не очень красивый код, то мы проводим его рефакторинг. Так как у нас весь ранее написанный функционал покрыт

тестами, то при написании нового и при рефакторинге, если мы что то и сломаем, то сразу об этом узнаем.

Интеграционное тестирование. Интеграционное тестирование - тестирование взаимодействия программных модулей.

Интеграционное тестирование - это тестирование части системы, состоящей из двух и более модулей.

Основная задача интеграционного тестирования - поиск дефектов, связанных с ошибками в реализации и интерпретации интерфейсного взаимодействия между модулями.

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

На рисунке 4 приведена структура комплекса программ K, состоящего из оттестированных на этапе модульного тестирования модулей M1, M2, M11, M12, M21, M22. Задача, решаемая методом интеграционного тестирования, - тестирование межмодульных связей, реализующихся при исполнении программного обеспечения комплекса K.

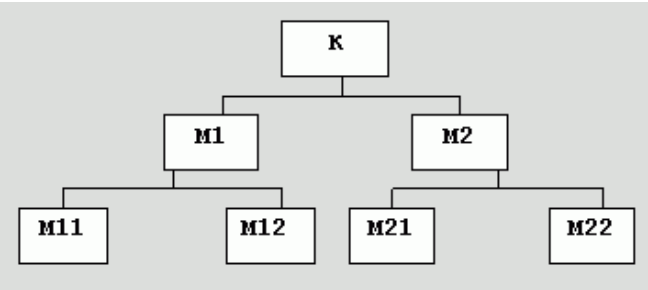

Рисунок 4 – Пример структуры комплекса программ

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

- монолитный, характеризующийся одновременным объединением всех модулей в тестируемый комплекс;

- инкрементальный, характеризующийся пошаговым (помодульным) наращиванием комплекса программ с пошаговым тестированием собираемого комплекса.

В инкрементальном методе выделяют две стратегии добавления модулей:

- «Сверху вниз» и соответствующее ему нисходящее тестирование;

- «Снизу вверх» и соответственно восходящее тестирование.

Системное тестирование. 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).

Нагрузочное тестирование. Нагрузочное тестирование – это технология измерения производительности сервера, которая заключается в отправке имитируемого HTTP-трафика на сервер.

Нагрузочное тестирование выполняется путем запуска специального программного обеспечения на одном компьютере (или в кластере машин). Это ПО генерирует большое количество запросов и отправляет их на веб-сервер на втором компьютере (или в другой инфраструктуре). Обычное программное обеспечение для нагрузочного тестирования используется для определения максимального количества запросов в секунду, которое может обрабатывать сервер. Для этого на сервер отправляется как можно большее количество запросов; затем нужно проверить, сколько из них сервер смог обработать успешно.

Это позволяет на базовом уровне определить максимальные возможности сервера, но это не предоставит много информации о задержках, ежедневной производительности и пользовательском опыте. Перегруженный сервер может возвращать тысячу ответов в секунду, но если обработка каждого ответа занимает десять секунд, пользователи, вероятно, не будут ждать.

Общая тенденция такова: чем выше нагрузка (чем больше запросов в секунду), тем выше задержка. Чтобы получить более реальную картину о задержке сервера при заданной нагрузке, нужно будет протестировать его несколько раз с разным количеством запросов.

Формальные инспекции. 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 оборудовании.

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, и 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. Рисунок 5 иллюстрирует место формальной инспекции в процессе разработки программных систем.

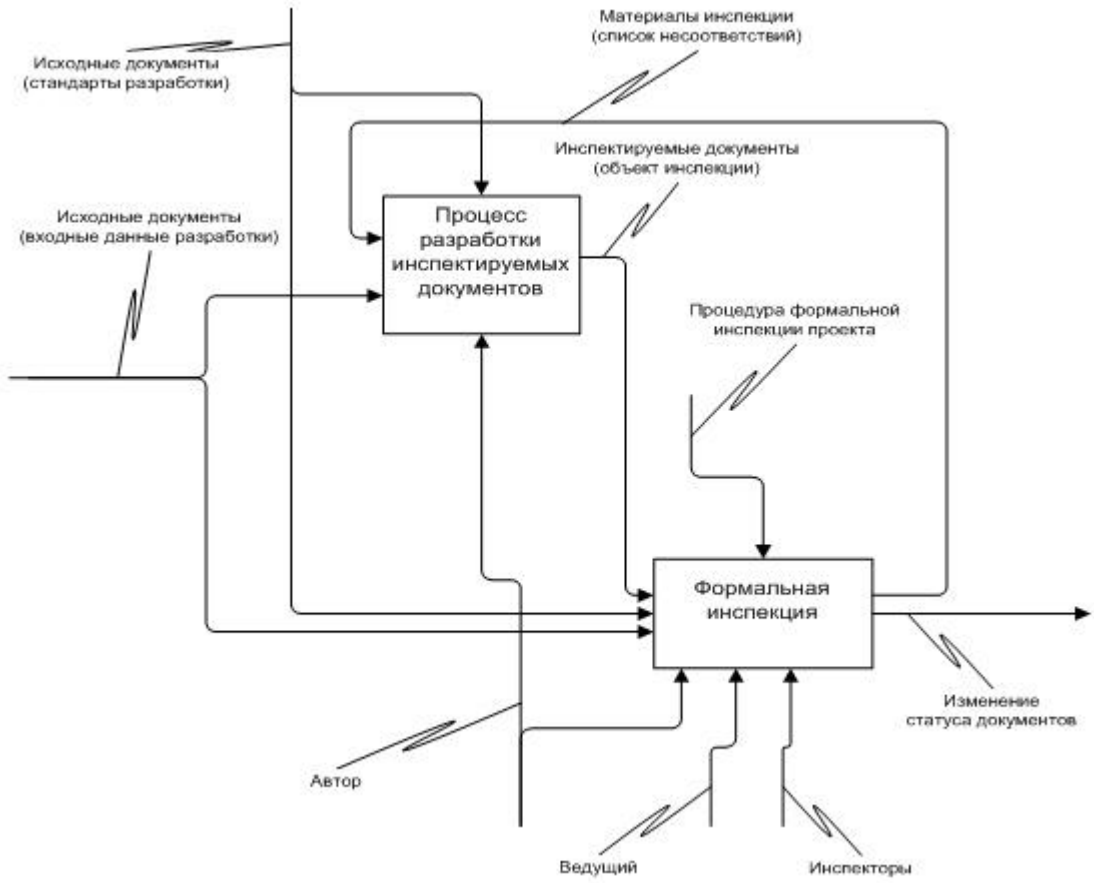

Рисунок 5 – Место формальной инспекции в проекте по разработке программной системы

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 сложиться 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.

Регрессионное тестирование. Регрессионное тестирование - это вид тестирования направленный на проверку изменений, сделанных в приложении или окружающей среде (починка дефекта, слияние кода, миграция на другую операционную систему, базу данных, веб сервер или сервер приложения), для подтверждения того факта, что существующая ранее функциональность работает как и прежде (см. также Санитарное тестирование или проверка согласованности/исправности). Регрессионными могут быть как функциональные, так и нефункциональные тесты.

Как правило, для регрессионного тестирования используются тест кейсы, написанные на ранних стадиях разработки и тестирования. Это дает гарантию того, что изменения в новой версии приложения не повредили уже существующую функциональность. Рекомендуется делать автоматизацию регрессионных тестов, для ускорения последующего процесса тестирования и обнаружения дефектов на ранних стадиях разработки программного обеспечения.

Сам по себе термин «Регрессионное тестирование», в зависимости от контекста использования может иметь разный смысл. Сэм Канер, к примеру, описал 3 основных типа регрессионного тестирования:

Регрессия багов (Bug regression) - попытка доказать, что исправленная ошибка на самом деле не исправлена

Регрессия старых багов (Old bugs regression) - попытка доказать, что недавнее изменение кода или данных сломало исправление старых ошибок, т.е. старые баги стали снова воспроизводиться.

Регрессия побочного эффекта (Side effect regression) - попытка доказать, что недавнее изменение кода или данных сломало другие части разрабатываемого приложения.

Комбинирование уровней тестирования. В каждом конкретном проекте должны быть определены задачи, ресурсы и технологии для каждого уровня тестирования таким образом, чтобы каждый из типов

дефектов, ожидаемых в системе, был «адресован», то есть в общем наборе тестов должны иметься тесты, направленные на выявление дефектов подобного типа. Таблица 1 суммирует характеристики свойств модульного, интеграционного и системного уровней тестирования. Задача, которая стоит перед тестировщиками и менеджерами, заключается в оптимальном распределении ресурсов между всеми тремя типами тестирования. Например, перенесение усилий на поиск фиксированного типа дефектов из области системного в область модульного тестирования может существенно снизить сложность и стоимость всего процесса тестирования.

Таблица 1 – Характеристики модульного, интеграционного и системного тестирования

	Модульное	Интеграционное	Системное
Типы дефектов	Локальные дефекты, такие как опечатки в реализации алгоритма, неверные операции, логические и математические выражения, циклы, ошибки в использовании локальных ресурсов, рекурсия и т. п.	Интерфейсные дефекты, такие как неверная трактовка параметров и их формат, неверное использование системных ресурсов и средств коммуникации, и т. п.	Отсутствующая или некорректная функциональность, неудобство использования, непредусмотренные данные и их комбинации, непредусмотренные или не поддержанные сценарии работы, ошибки совместимости, ошибки пользовательской документации, ошибки переносимости продукта на различные платформы, проблемы производительности, инсталляции и т. п.
Необходимость в системе тестирования	Да	Да	Нет
Цена разработки системы тестирования	Низкая	Низкая до умеренной	Умеренная до высокой или неприемлемой
Цена всего процесса тестирования	Низкая	Низкая	Высокая

Тема 1.3 Критерии выбора тестов

Требования к идеальному критерию

Классы критериев. Структурные критерии. Функциональные критерии. Стохастические критерии. Мутационный критерий

Оценка покрытия программы и проекта. Методика интегральной оценки тестированности

Тест-требования как основной источник информации для создания тестовых примеров

Литература: [1], с.31-50

Методические рекомендации

Требования к идеальному критерию тестирования:

- критерий должен быть достаточным, т.е. показывать, когда некоторое конечное множество тестов достаточно для тестирования данной программы;
- критерий должен быть полным, т.е. в случае ошибки должен существовать тест из множества тестов, удовлетворяющих критерию, который раскрывает ошибку;
- критерий должен быть надежным, т.е. любые два множества тестов, удовлетворяющих ему, одновременно должны раскрывать или не раскрывать ошибки программы;
- критерий должен быть легко проверяемым, например вычисляемым на тестах.

Классы критериев:

- структурные;
- функциональные;
- критерии стохастического тестирования формулируются в терминах проверки наличия заданных свойств у тестируемого приложения, средствами проверки некоторой статистической гипотезы;
- мутационные критерии ориентированы на проверку свойств программного изделия на основе подхода Монте-Карло.

Тема 1.4 Методы тестирования

Методы тестирования:

– «Черный ящик»;

– «Белый (стеклянный) ящик»;

– тестирование моделей;

– анализ программного кода (инспекции).

Текстовое окружение: драйверы и заглушки, генераторы сигналов (событийно-управляемый код)

Литература: [3]

Методические рекомендации

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 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.

Белый (стеклянный) ящик. При тестировании системы как стеклянного ящика тестировщик имеет доступ не только к требованиям к системе, ее входам и выходам, но и к ее внутренней структуре - видит ее программный код.

Тестирование моделей. Тестирование моделей находится несколько в стороне от классических методов верификации программного обеспечения. Причина прежде всего в том, 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 могут быть доказаны формальными средствами), то тестировщик получает в свое распоряжение достаточно мощный инструмент анализа общей целостности системы. На модели 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 из-за трудностей, возникающих при разработке формального описания поведения системы. Одно из немногих исключений - системы связи, алгоритмический и математический аппарат которых достаточно хорошо проработан.

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.

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. Обобщенная схема среды тестирования представлена на рисунке 6.

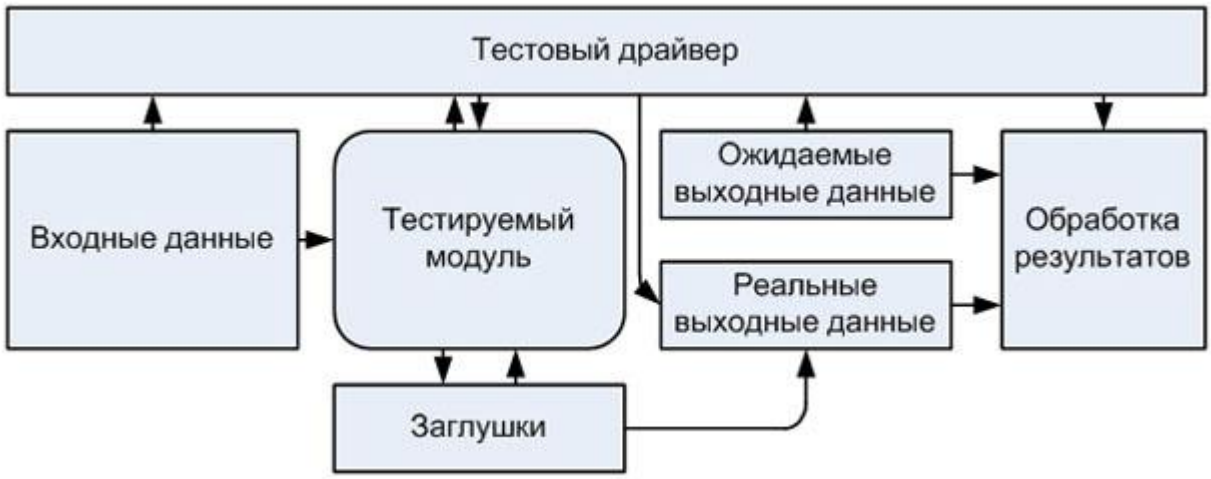

Рисунок 6 – Обобщенная схема среды тестирования

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 зачастую реализованной 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.

Вопросы для самоконтроля

1 Дайте определение понятиям отладка, тестирование.

2 Опишите взаимосвязь тестирования, валидации и верификации.

3 Дайте понятие термину верификация.

4 Дайте понятие термину валидация.

5 Перечислите основные причины появления дефектов в программном коде.

6 Опишите процесс осуществления модульного тестирования.

7 Опишите процесс осуществления интеграционного тестирования.

8 Опишите процесс осуществления системного тестирования.

9 Опишите процесс осуществления загрузочного тестирования.

10 Перечислите и опишите типы инспекций.

11 Дайте определение понятию регрессионное тестирование.

12 Охарактеризуйте модульное, интеграционное и системное тестирование.

13 Перечислите требования к идеальному критерию тестирования.

14 Перечислите классы критериев.

15 Перечислите и опишите методы тестирования.

16 Для чего осуществляется тестирование тестового окружения?

Раздел 2 Структурное и функциональное тестирование ПО
Тема 2.1 Структурное тестирование ПО (тестирование методом «белого ящика»)

Цели и задачи тестирования программного кода. Инспекция кода и прогон. Операторное покрытие и покрытие ветвлений. Покрытие условий и путей

Граф управления потоками. Метрика МакКейба. Базовый метод построения независимых путей для структурного тестирования

Тест-примеры, их типы. Проверка на граничных значениях и робастности. Классы эквивалентности

Тест-план, его типовая структура

Литература: [3]

Методические рекомендации

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

Тестовые примеры. Тестовое окружение, рассмотренное в предыдущей лекции, 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. Цель выполнения любого тестового примера - либо продемонстрировать наличие в системе дефекта, либо доказать его отсутствие.

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.

Например:

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

a 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.

b 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

a 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 ненулевом значении подсчитанной контрольной суммы должно возвращаться значение FALSE.

b 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.

Например:

Тест-требования.

1 Проверка инициализация модуля

Проверить, что начальное значение переменной Empty установлено TRUE.

2 Проверка подсчета контрольной суммы

a Проверить, что в процедуре Set_CRC вычисление контрольной суммы производится по правилам алгоритма CRC32, как определено в секции 2a функциональных требований.

b Проверить, что вычисленное значение контрольной суммы не зависит от начального значения поля CRC.

c Проверить, что вычисленное значение контрольной суммы не зависит от значений байт выравнивания полей записи.

d 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

a Проверить, что при обращении к процедуре Check_CRC вычисление контрольной суммы производится по правилам алгоритма CRC32, как определено в секции 3a функциональных требований.

b Проверить, что возвращаемое значение равно TRUE, если контрольная сумма проверяемой записи правильная, и FALSE - в противном случае.

c Проверить, что проверка правильности значения контрольной суммы не зависит от значений байт выравнивания полей записи.

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

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 либо 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:

1 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.

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

3 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.

4 Определение последовательности тестирования. Одно из важных свойств тестового примера - 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, тесты объединяются в последовательности, где 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.

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 станет слишком громоздкой. Позднее будут рассмотрены табличные формы представления тест-планов, позволяющие записывать их более компактно.

Тема 2.2 Тестирование потоков данных

Свойства и тестирование потоков данных программных модулей

Тестирование графов модулей программ с учетом значений переменных и констант

Тестирование циклов

Литература: [5], с.224-246

Методические рекомендации

В графах потока данных значения объектов связаны с обрабатывающими узлами, в которых эти значения вычисляются. При применении графов потока данных узел может быть использован для представления нескольких различных вычислений и, таким образом, может иметь несколько различных объектов, ассоциированных с исходящими связями. Обычная практика – размещение имен объектов на связях, исходящих из таких узлов. При таком использовании эти имена характеризуются значениями исходящих связей. Поскольку исходящие из одного узла связи являются входящими по отношению к следующему узлу, они называются значениями входящих связей. Узел выбора данных вычисляет специальную функцию значения входящей связи для использования в исходящей связи. Входящие связи узла выбора данных должны быть помечены условием предиката, при котором выбирается данная входящая связь. Его значение выбирает объект данных, соответствующий входящей связи. Обрабатывающий узел с одной или более входящими связями формируется, по крайней мере, с одной исходящей связью. Входящие связи обозначаются именами данных. Исходящая связь обозначает вычисленную функцию этих объектов данных. Запоминающие узлы при последующей обработке должны уточняться, какое именно значение переменной используется: значение, сохраненное на диске, или значение в ОЗУ, предшествующее сохранению. При использовании в сетях с файлами совместного доступа они могут отличаться (и таким образом быть источником ошибок).

Тестирование потоков данных можно разделить на два этапа. Первый этап тестирования состоит в анализе узлов обработки данных, определяющих значения предикатов в операторах выработки логических решений при взаимодействии элементов в модуле программ. Эти решения влияют на изменения маршрутов обработки информации, что сближает в этой части метод тестирования потоков данных с тестированием структуры модуля. Второй этап − тестирование обработки данных на соответствие заданным требованиям. Оно состоит в проверке вычислений по аналитическим формулам или определение численных значений результатов решения задач в зависимости от числовых или логических значений исходных данных.

Модель потока данных – это способ структурировать мышление, необходимый для получения полезных тестов. Если сделать отдельно модель потока данных и модель потока управления, каждая приведет к различным тестам, и при таком подходе не будет значительного пере-

крытия созданных тестов. Поэтому полезно помещать модель потока управления внутрь модели потока данных. Повторяющиеся процедуры в модели большей частью определяются потоками данных. Можно поместить модель потока данных внутрь модели потока управления. Тогда обработка моделируется при помощи графа потока данных. Выбор способа совмещения моделей потоков зависит от того, какой из них является более сложным и доминирующим.

Тема 2.3 Повторяемость тестирования

Цели и задачи обеспечения повторяемости тестирования

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

Зависимость между тестовыми примерами; настройки по умолчанию для тестовых примеров и их групп

Литература: [3]

Методические рекомендации

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. Из-за 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 - например, когда требуется длительное время для выполнения всех тестов, а время, отведенное на процесс тестирования, ограничено. В этом случае часто применяется практика выборочного тестирования отдельных частей системы, затронутых изменениями. Полное тестирование при таком подходе проводится только после накопления доста-

точно 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.

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.

Тема 2.4 Функциональное тестирование ПО (тестирование методом «черного ящика»)

Функциональное тестирование ПО (тестирование методом «черного ящика»)

Разбиение на классы эквивалентности. Анализ граничных значений

Функциональные диаграммы. Тестирование с помощью функциональных диаграмм

Литература: [6]

Методические рекомендации

Тестирование «черного ящика» (функциональное тестирование) позволяет получить комбинации входных данных, обеспечивающих полную проверку всех функциональных требований к программе. Программное изделие здесь рассматривается как «черный ящик», чье поведение можно определить только исследованием его входов и соответствующих выходов. При таком подходе желательно иметь:

- набор, образуемый такими входными данными, которые приводят к аномалиям поведения программы (назовем его IT);
- набор, образуемый такими выходными данными, которые демонстрируют дефекты программы (назовем его OT).

Как показано на рисунке 7, любой способ тестирования «черного ящика» должен:

- выявить такие входные данные, которые с высокой вероятностью принадлежат набору IT;
- сформулировать такие ожидаемые результаты, которые с высокой вероятностью являются элементами набора OT.

Во многих случаях определение таких тестовых вариантов основывается на предыдущем опыте инженеров тестирования. Они используют свое знание и понимание области определения для идентификации тестовых вариантов, которые эффективно обнаруживают дефекты. Тем не менее систематический подход к выявлению тестовых данных, обсуждаемый в данной главе, может использоваться как полезное дополнение к эвристическому знанию.

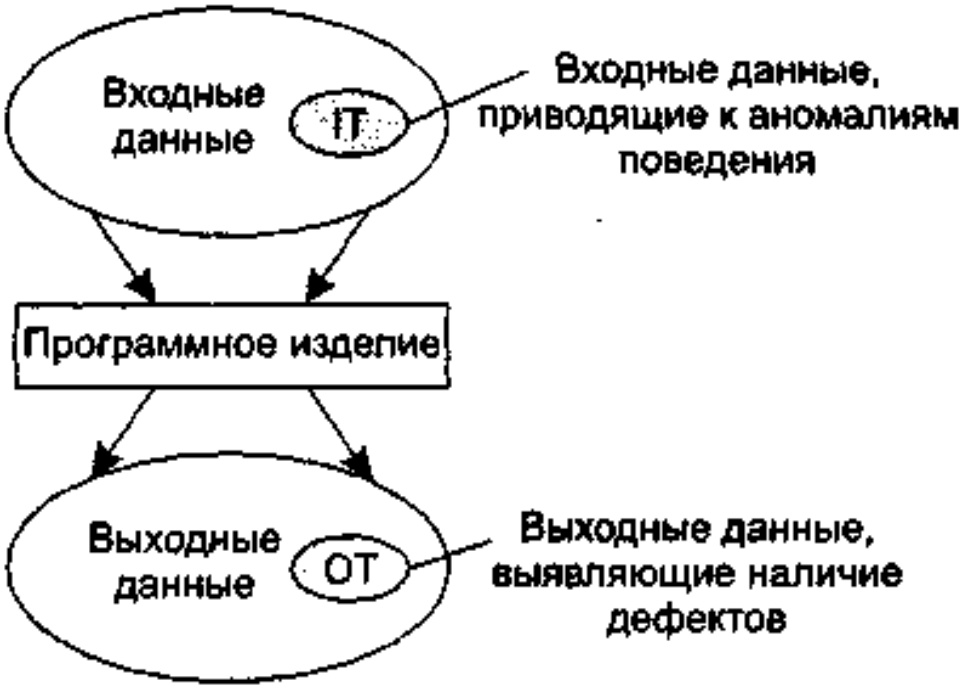

Рисунок 7 – Тестирование «черного ящика»

Принцип «черного ящика» не альтернативен принципу «белого ящика». Скорее это дополняющий подход, который обнаруживает другой класс ошибок.

Тестирование «черного ящика» обеспечивает поиск следующих категорий ошибок:

1) некорректных или отсутствующих функций;

2) ошибок интерфейса;

3) ошибок во внешних структурах данных или в доступе к внешней базе данных;

4) ошибок характеристик (необходимая емкость памяти и т. д.);

5) ошибок инициализации и завершения.

Подобные категории ошибок способами «белого ящика» не выявляются.

В отличие от тестирования «белого ящика», которое выполняется на ранней стадии процесса тестирования, тестирование «черного ящика» применяют на поздних стадиях тестирования. При тестировании «черного ящика» пренебрегают управляющей структурой программы. Здесь внимание концентрируется на информационной области определения программной системы.

Техника «черного ящика» ориентирована на решение следующих задач:

- сокращение необходимого количества тестовых вариантов (из-за проверки не статических, а динамических аспектов системы);

- выявление классов ошибок, а не отдельных ошибок.

Способ разбиения по эквивалентности. Разбиение по эквивалентности - самый популярный способ тестирования «черного ящика».

В этом способе входная область данных программы делится на классы эквивалентности. Для каждого класса эквивалентности разрабатывается один тестовый вариант.

Класс эквивалентности - набор данных с общими свойствами. Обрабатывая разные элементы класса, программа должна вести себя одинаково. Иначе говоря, при обработке любого набора из класса эквивалентности в программе задействуется один и тот же набор операторов (и связей между ними).

На рисунке 8 каждый класс эквивалентности показан эллипсом. Здесь выделены входные классы эквивалентности допустимых и недопустимых исходных данных, а также классы результатов.

Классы эквивалентности могут быть определены по спецификации на программу.

Рисунок 8 – Разбиение по эквивалентности

Например, если спецификация задает в качестве допустимых входных величин 5-разрядные целые числа в диапазоне 15 000...70 000, то класс эквивалентности допустимых ИД (исходных данных) включает величины от 15 000 до 70 000, а два класса эквивалентности недопустимых ИД составляют:

– числа меньшие, чем 15 000;

– числа большие, чем 70 000.

Класс эквивалентности включает множество значений данных, допустимых или недопустимых по условиям ввода.

Условие ввода может задавать:

1) определенное значение;

2) диапазон значений;

3) множество конкретных величин;

4) булево условие.

Сформулируем правила формирования классов эквивалентности.

1 Если условие ввода задает диапазон п...т, то определяются один допустимый и два недопустимых класса эквивалентности:

– V_Class={n.. .т} — допустимый класс эквивалентности;

– Inv_C1ass1={x|для любого x: x < п} — первый недопустимый класс эквивалентности;

– Inv_C1ass2={y|для любого y: y > т} — второй недопустимый класс эквивалентности.

2 Если условие ввода задает конкретное значение a, то определяется один допустимый и два недопустимых класса эквивалентности:

- V_Class={a};
- Inv_Class1 ={x|длялюбого x: x < a};
- Inv_C1ass2={y|для любого y: y > a}.

3 Если условие ввода задает множество значений {a, b, c}, то определяются один допустимый и один недопустимый класс эквивалентности:

- V_Class={a, b, c};
- Inv_C1ass={x|для любого x: (x ≠ a)&(x ≠b)&(x ≠ c)}.

4 Если условие ввода задает булево значение, например true, то определяются один допустимый и один недопустимый класс эквивалентности:

- V_Class={true};
- Inv_Class={false}.

После построения классов эквивалентности разрабатываются тестовые варианты. Тестовый вариант выбирается так, чтобы проверить сразу наибольшее количество свойств класса эквивалентности.

Способ анализа граничных значений. Как правило, большая часть ошибок происходит на границах области ввода, а не в центре. Анализ граничных значений заключается в получении тестовых вариантов, которые анализируют граничные значения [3], [14], [69]. Данный способ тестирования дополняет способ разбиения по эквивалентности.

Основные отличия анализа граничных значений от разбиения по эквивалентности:

1) тестовые варианты создаются для проверки только ребер классов эквивалентности;

2) при создании тестовых вариантов учитывают не только условия ввода, но и область вывода.

Сформулируем правила анализа граничных значений.

1 Если условие ввода задает диапазон п...т, тотестовые варианты должны быть построены:

- для значений п и т;
- для значений чуть левее п ичуть правее т на числовой оси.

Например, если задан входной диапазон -1,0...+1,0, то создаются тесты для значений - 1,0, +1,0, - 1,001, +1,001.

2 Если условие ввода задает дискретное множество значений, то создаются тестовые варианты:

- для проверки минимального и максимального из значений;
- для значений чуть меньше минимума и чуть больше максимума.

Так, если входной файл может содержать от 1 до 255 записей, то создаются тесты для О, 1, 255, 256 записей.

3 Правила 1 и 2 применяются к условиям области вывода.

Рассмотрим пример, когда в программе требуется выводить таблицу значений. Количество строк и столбцов в таблице меняется. Задается тестовый вариант для минимального вывода (по объему таблицы), а также тестовый вариант для максимального вывода (по объему таблицы).

4 Если внутренние структуры данных программы имеют предписанные границы, то разрабатываются тестовые варианты, проверяющие эти структуры на их границах.

5 Если входные или выходные данные программы являются упорядоченными множествами (например, последовательным файлом, линейным списком, таблицей), то надо тестировать обработку первого и последнего элементов этих множеств.

Большинство разработчиков используют этот способ интуитивно. При применении описанных правил тестирование границ будет более полным, в связи с чем возрастет вероятность обнаружения ошибок.

Рассмотрим применение способов разбиения по эквивалентности и анализа граничных значений на конкретном примере. Положим, что нужно протестировать программу бинарного поиска. Нам известна спецификация этой программы. Поиск выполняется в массиве эле-

ментов M, возвращается индекс Iэлемента массива, значение которого соответствует ключу поиска Key.

Предусловия:

1) массив должен быть упорядочен;

2) массив должен иметь не менее одного элемента;

3) нижняя граница массива (индекс) должна быть меньше или равна его верхней границе.

Постусловия:

1) если элемент найден, то флаг Result=True, значение I — номер элемента;

2) если элемент не найден, то флаг Result=False, значение I не определено.

Для формирования классов эквивалентности (и их ребер) надо произвести разбиение области ИД — построить дерево разбиений. Листья дерева разбиений дадут нам искомые классы эквивалентности. Определим стратегию разбиения. На первом уровне будем анализировать выполнимость предусловий, на втором уровне — выполнимость постусловий. На третьем уровне можно анализировать специальные требования, полученные из практики разработчика. В нашем примере мы знаем, что входной массив должен быть упорядочен. Обработка упорядоченных наборов из четного и нечетного количества элементов может выполняться по-разному. Кроме того, принято выделять специальный случай одноэлементного массива. Следовательно, на уровне специальных требований возможны следующие эквивалентные разбиения:

1) массив из одного элемента;

2) массив из четного количества элементов;

3) массив из нечетного количества элементов, большего единицы.

Наконец на последнем, 4-м уровне критерием разбиения может быть анализ ребер классов эквивалентности. Очевидно, возможны следующие варианты:

1) работа с первым элементом массива;

2) работа с последним элементом массива;

3) работа с промежуточным (ни с первым, ни с последним) элементом массива.

Структура дерева разбиений приведена на рисунке 9.

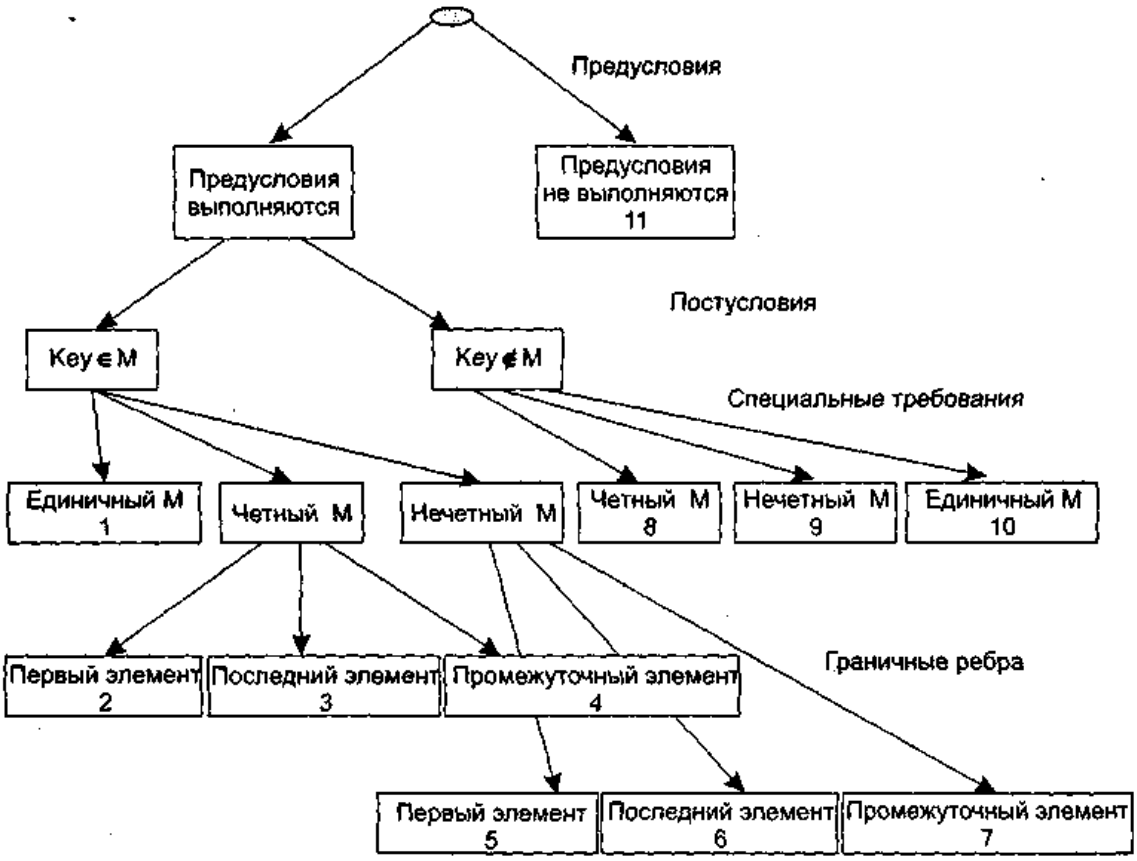

Рисунок 9 – Дерево разбиений области исходных данных бинарного поиска

Это дерево имеет 11 листьев. Каждый лист задает отдельный тестовый вариант. Покажем тестовые варианты, основанные на проведенных разбиениях.

Тестовый вариант 1 (единичный массив, элемент найден) ТВ1:

ИД: M=15; Key=15.

ОЖ.РЕЗ.: Resutt=True; I=1.

Тестовый вариант 2 (четный массив, найден 1-й элемент) ТВ2:

ИД: M=15, 20, 25,30,35,40; Key=15.

ОЖ.РЕЗ.: Result=True; I=1.

Тестовый вариант 3 (четный массив, найден последний элемент) ТВ3:

ИД: M=15, 20, 25, 30, 35, 40; Key=40.

ОЖ.РЕЗ:. Result=True; I=6.

Тестовый вариант 4 (четный массив, найден промежуточный элемент) ТВ4:

ИД: M=15,20,25,30,35,40; Key=25.

ОЖ.РЕЗ.: Result-True; I=3.

Тестовый вариант 5 (нечетный массив, найден 1-й элемент) ТВ5:

ИД: M=15, 20, 25, 30, 35,40, 45; Key=15.

ОЖ.РЕЗ.: Result=True; I=1.

Тестовый вариант 6 (нечетный массив, найден последний элемент) ТВ6:

ИД: M=15, 20, 25, 30,35, 40,45; Key=45.

ОЖ.РЕЗ.: Result=True; I=7.

Тестовый вариант 7 (нечетный массив, найден промежуточный элемент) ТВ7:

ИД: M=15, 20, 25, 30,35, 40, 45; Key=30.

ОЖ.РЕЗ.: Result=True; I=4.

Тестовый вариант 8 (четный массив, не найден элемент) ТВ8:

ИД: M=15, 20, 25, 30, 35,40; Key=23.

ОЖ.РЕЗ.: Result=False; I=?

Тестовый вариант 9 (нечетный массив, не найден элемент) ТВ9;

ИД: M=15, 20, 25, 30, 35, 40, 45; Key=24.

ОЖ.РЕЗ:. Result=False; I=?

Тестовый вариант 10 (единичный массив, не найден элемент) ТВ10:

ИД: M=15; Key=0.

ОЖ.РЕЗ.: Result=False; I=?

Тестовый вариант 11 (нарушены предусловия) ТВ11:

ИД: M=15, 10, 5, 25, 20, 40, 35; Key=35.

ОЖ.РЕЗ.: Аварийное донесение: Массив не упорядочен.

Вопросы для самоконтроля

1 Дайте определение понятию тестирование программного кода.

2 Дайте определение понятию тест-примеры и перечислите их типы.

3 Дайте определение понятию тест-план и опишите его типовую структуру.

4 Из чего состоит формирование и тестирование потоков данных модуля?

5 Опишите цели и задачи обеспечения повторяемости тестирования.

6 Опишите способ тестирования «черного ящика».

7 Тестирование «черного ящика» обеспечивает поиск каких категорий ошибок?

Раздел 3 Организация тестирования ПО
Тема 3.1 Методика тестирования программных систем

Организация тестирования. Три фазы тестирования

Методика тестирования программных систем (ПС). Нисходящая и восходящая стратегии тестирования, особенности тестового окружения для них

Литература: [3]

Методические рекомендации

Тестирование осуществляется на заданном заранее множестве входных данных X и множестве предполагаемых результатов Y – (X,Y), которые задают график желаемой функции. Кроме того, зафиксирована процедура Оракул (oracle), которая определяет, соответствуют ли выходные данные – Yв (вычисленные по входным данным – X) желаемым результатам – Y, т.е. принадлежит ли каждая вычисленная точка (X,Yв) графику желаемой функции (X,Y).

Оракул дает заключение о факте появления неправильной пары (X,Yв) и ничего не говорит о том, каким образом она была вычислена или каков правильный алгоритм – он только сравнивает вычисленные и желаемые результаты. Оракулом может быть даже Заказчик или программист, производящий соответствующие вычисления в уме, поскольку Оракулу нужен какой-либо альтернативный способ получения функции (X,Y) для вычисления эталонных значений Y.

Реализация тестирования разделяется на три этапа:

- создание тестового набора (test suite) путем ручной разработки или автоматической генерации для конкретной среды тестирования (testing environment);
- прогон программы на тестах, управляемый тестовым монитором (test monitor, test driver [IEEE Std 829-1983]) с получением протокола результатов тестирования (test log);
- оценка результатов выполнения программы на наборе тестов с целью принятия решения о продолжении или остановке тестирования.

Основная проблема тестирования – определение достаточности множества тестов для истинности вывода о правильности реализации программы, а также нахождения множества тестов, обладающего этим свойством.

Тема 3.2 Модульное тестирование

Цели и задачи модульного тестирования. Понятие о модуле и его границах

Подходы к проектированию тестового окружения

Организация модульного тестирования

Литература: [3]

Методические рекомендации

Каждая сложная программная система состоит из отдельных частей - модулей, выполняющих ту или иную функцию в составе системы. Для того, чтобы удостовериться в корректной работе всей системы, 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:

– поиск и документирование несоответствий требованиям;

– поддержка разработки и рефакторинга низкоуровневой архитектуры системы и межмодульного взаимодействия;

– поддержка рефакторинга модулей;

– 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, где 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 сопряжены 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.

Тема 3.3 Интеграционное тестирование

Цели и задачи интеграционного тестирования

Организация интеграционного тестирования. Структурная классификация методов интеграционного тестирования. Временная классификация методов интеграционного тестирования. Планирование интеграционного тестирования

Литература: [3]

Методические рекомендации

Результатом тестирования и верификации отдельных модулей, составляющих программную систему, должно быть 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 название обусловлено 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 к системе 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 то, 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, 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.

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 область поиска проблем у тестировщика достаточно узка, и поэтому гораздо выше вероятность правильно идентифицировать дефект.

На рисунке 10 представлена разработка драйверов и заглушек при восходящем интеграционном тестировании.

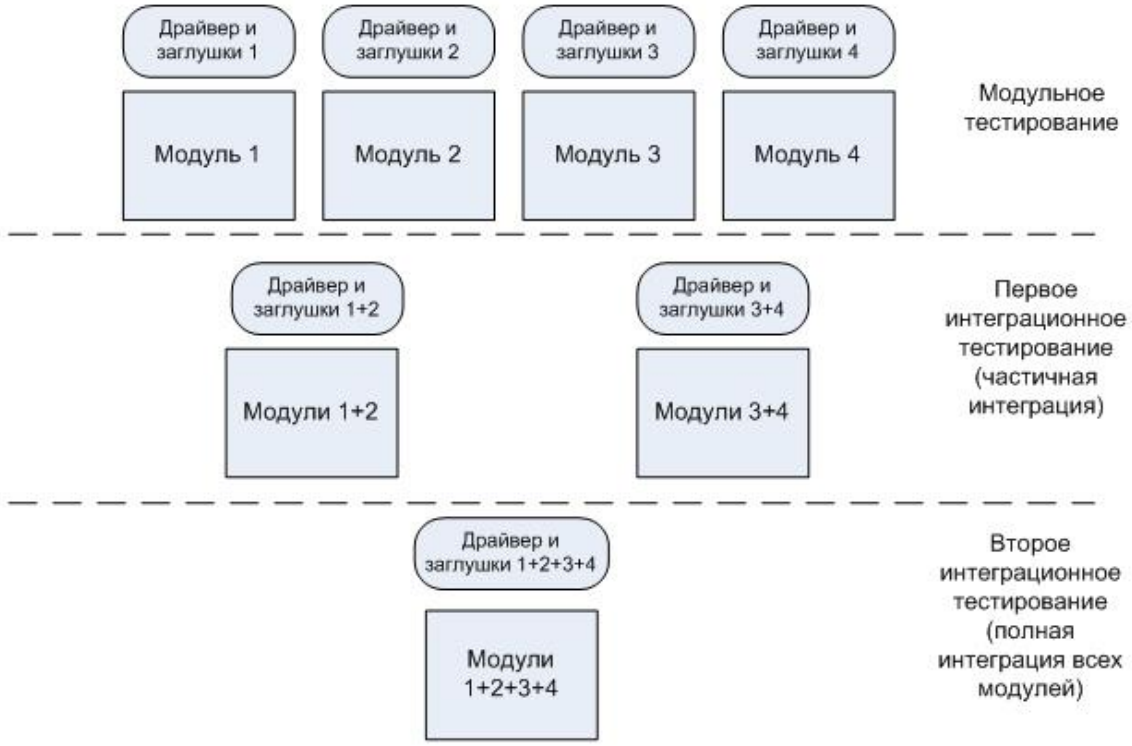

Рисунок 10 – Разработка драйверов и заглушек при восходящем интеграционном тестировании

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 (рисунок 10).

С одной стороны драйверы и заглушки - мощный инструмент тестирования, с другой - их разработка требует значительных ресурсов, особенно при изменении состава интегрируемых модулей, иначе говоря, 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, которое поддерживает 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, затем система проверяется вся в целом. Этот подход не следует путать с системным тестированием, ко-

торому посвящена следующая лекция. 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 сводится к определению 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 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 (рисунок 11).

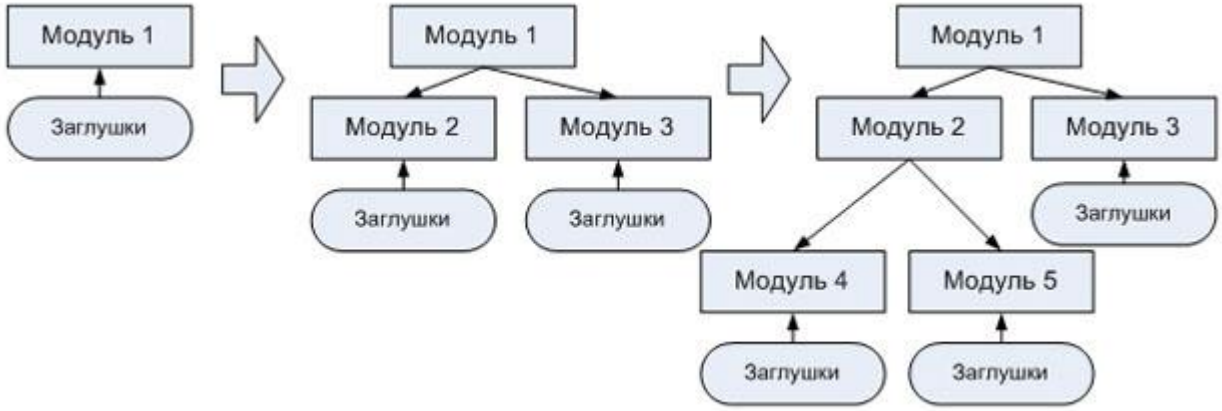

Рисунок 11 – Постепенная интеграция модулей при нисходящем методе тестирования

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, который основан 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.

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 оправдан в том случае, если система является конгломератом слабо связанных между собой модулей, которые взаимодействуют 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 2 – Основные характеристики различных видов интеграционного тестирования

Свойство	Вид интеграции					
	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 таблице 2 приведены основные характеристики рассмотренных 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.

Планирование интеграционного тестирования.

Процесс организации и планирования интеграционного тестирования во многом схож с процессом организации модульного тестирования, рассмотренном в предыдущей лекции. Однако интеграционное тестирование имеет ряд организационных особенностей, перечисленных ниже.

На этапе планирования разрабатывается концепция и стратегия интеграции - документ, где описан общий подход к определению последовательности, в которой должны интегрироваться модули. Как правило, концепция основывается на одном из видов интеграции, рас-

смотренных 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 3.4 Системное тестирование

Цели и задачи системного тестирования

Виды системного тестирования

Системное тестирование, приемо-сдаточные и сертификационные испытания при разработке сертифицируемого программного обеспечения

Литература: [3]

Методические рекомендации

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 применяется 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, которая не связана 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.

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.

Тема 3.5 Отладка ПО, ее виды

Классификация ошибок
Методы отладки ПО
Методы и средства получения дополнительной информации
Общая методика отладки ПО
Литература: [7]

Методические рекомендации

Отладка программы - один их самых сложных этапов разработки программного обеспечения, требующий глубокого знания:

– специфики управления используемыми техническими средствами;

– операционной системы;

– среды и языка программирования;

– реализуемых процессов;

– природы и специфики различных ошибок;

– методик отладки и соответствующих программных средств.

Классификация ошибок. Отладка - это процесс локализации и исправления ошибок, обнаруженных при тестировании программного обеспечения. Локализацией называют процесс определения оператора программы, выполнение которого вызвало нарушение нормального вычислительного процесса. Для исправления ошибки необходимо определить ее причину, т. е. определить оператор или фрагмент, содержащие ошибку. Причины ошибок могут быть как очевидны, так и очень глубоко скрыты.

В целом сложность отладки обусловлена следующими причинами:

– требует от программиста глубоких знаний специфики управления используемыми техническими средствами, операционной системы, среды и языка программирования, реализуемых процессов, природы и специфики различных ошибок, методик отладки и соответствующих программных средств;

– психологически дискомфортна, так как необходимо искать собственные ошибки и, как правило, в условиях ограниченного времени;

– возможно взаимовлияние ошибок в разных частях программы, например, за счет затирания области памяти одного модуля другим из-за ошибок адресации;

– отсутствуют четко сформулированные методики отладки.

Классификация ошибок по этапу обработки программы представлена на рисунке 12.

Рисунок 12 – Классификация ошибок по этапу обработки программы

В соответствии с этапом обработки, на котором проявляются ошибки, различают:

– синтаксические ошибки - ошибки, фиксируемые компилятором (транслятором, интерпретатором) при выполнении синтаксического и частично семантического анализа программы;

– ошибки компоновки - ошибки, обнаруженные компоновщиком (редактором связей) при объединении модулей программы;

– ошибки выполнения - ошибки, обнаруженные операционной системой, аппаратными средствами или пользователем при выполнении программы.

Синтаксические ошибки. Синтаксические ошибки относят к группе самых простых, так как синтаксис языка, как правило, строго формализован, и ошибки сопровождаются развернутым комментарием с указанием ее местоположения. Определение причин таких ошибок, как правило, труда не составляет, и даже при нечетком знании правил языка за несколько прогонов удается удалить все ошибки данного типа.

Следует иметь в виду, что чем лучше формализованы правила синтаксиса языка, тем больше ошибок из общего количества может обнаружить компилятор и, соответственно, меньше ошибок будет обна-

руживаться на следующих этапах. В связи с этим говорят о языках программирования с защищенным синтаксисом и с незащищенным синтаксисом. К первым, безусловно, можно отнести Pascal, имеющий очень простой и четко определенный синтаксис, хорошо проверяемый при компиляции программы, ко вторым - Си со всеми его модификациями. Чего стоит хотя бы возможность выполнения присваивания в условном операторе в Си, например: If (c=n) x=0; /*.

В данном случае не проверятся равенство с и n, а выполняется присваивание с значения n, после чего результат операции сравнивается с нулем, если программист хотел выполнить не присваивание, а сравнение, то эта ошибка будет обнаружена только на этапе выполнения при получении результатов, отличающихся от ожидаемых */.

Ошибки компоновки. Ошибки компоновки, как следует из названия, связаны с проблемами, обнаруженными при разрешении внешних ссылок. Например, предусмотрено обращение к подпрограмме другого модуля, а при объединении модулей данная подпрограмма не найдена или не стыкуются списки параметров. В большинстве случаев ошибки такого рода также удается быстро локализовать и устранить.

Ошибки выполнения. К самой непредсказуемой группе относятся ошибки выполнения. Прежде всего, они могут иметь разную природу, и соответственно по-разному проявляться. Часть ошибок обнаруживается и документируется операционной системой. Выделяют четыре способа проявления таких ошибок:

– появление сообщения об ошибке, зафиксированной схемами контроля выполнения машинных команд, например, переполнении разрядной сетки, ситуации «деление па ноль», нарушении адресации и т.п.;

– появление сообщения об ошибке, обнаруженной операционной системой, например, нарушении зашиты памяти, попытке записи на устройства, защищенные от записи, отсутствии файла с заданным именем и т. п.;

– «зависание» компьютера, как простое, когда удается завершить программу бел перезагрузки операционной системы, так и «тяжелое», когда для продолжения работы необходима перезагрузка;

– несовпадение полученных результатов с ожидаемыми.

Причины ошибок выполнения очень разнообразны, а потому и локализация может оказаться крайне сложной. Все возможные причины ошибок можно разделить на следующие группы:

– неверное определение исходных данных;

– логические ошибки;

– накопление погрешностей результатов вычислений (рисунок 13). Неверное определение исходных данных происходит, если возникают любые ошибки при выполнении операций ввода-вывода: ошибки передачи, ошибки преобразования, ошибки перезаписи и ошибки данных. Причем использование специальных технических средств и программирование с защитой от ошибок позволяет обнаружить и предотвратить только часть этих ошибок, о чем безусловно не следует забывать.

Логические ошибки имеют разную природу. Так они могут следовать из ошибок, допущенных при проектировании, например, при выборе методов, разработке алгоритмов или определении структуры классов, а могут быть непосредственно внесены при кодировании модуля. К последней группе относят:

– ошибки некорректного использования переменных, например, неудачный выбор типов данных, использование переменных до их инициализации, использование индексов, выходящих за границы определения массивов, нарушения соответствия типов данных при использовании явного или неявного переопределения типа данных, расположенных в памяти при использовании нетипизированных переменных, открытых массивов, объединений, динамической памяти, адресной арифметики и т. д.

– ошибки вычислений, например, некорректные вычисления над неарифметическими переменными, некорректное использование целочисленной арифметики, некорректное преобразование типов данных в процессе вычислений, ошибки, связанные с незнанием приоритетов выполнения операций для арифметических и логических выражений, и т. п.;

– ошибки межмодульного интерфейса, например, игнорирование системных соглашений, нарушение типов и последовательности при передаче параметров, несоблюдение единства единиц измерения формальных и фактических параметров, нарушение области действия локальных и глобальных переменных;

– другие ошибки кодирования, например, неправильная реализация логики программы при кодировании, игнорирование особенностей или ограничений конкретного языка программирования.

На рисунке 13 представлена классификация ошибок этапа выполнения по возможным причинам

Рисунок 13 – Классификация ошибок этапа выполнения по возможным причинам

Накопление погрешностей результатов числовых вычислений возникает, например, при некорректном отбрасывании дробных цифр чисел, некорректном использовании приближенных методов вычислений, игнорировании ограничения разрядной сетки представления вещественных чисел в ЭВМ и т. п.

Все указанные выше причины возникновения ошибок следует иметь в виду в процессе отладки. Кроме того, сложность отладки увеличивается также вследствие влияния следующих факторов:

– опосредованного проявления ошибок;

– возможности взаимовлияния ошибок;

– возможности получения внешне одинаковых проявлений разных ошибок;

– отсутствия повторяемости проявлений некоторых ошибок от запуска к запуску - так называемые стохастические ошибки;

– возможности устранения внешних проявлений ошибок в исследуемой ситуации при внесении некоторых изменений в программу, например, при включении в программу диагностических фрагментов может аннулироваться или измениться внешнее проявление ошибок;

– написания отдельных частей программы разными программистами.

Методы отладки программного обеспечения. Отладка программы в любом случае предполагает обдумывание и логическое осмысление всей имеющейся информации об ошибке. Большинство ошибок можно обнаружить по косвенным признакам посредством тщательного анализа текстов программ и результатов тестирования без получения дополнительной информации. При этом используют различные методы:

– ручного тестирования;

– индукции;

– дедукции;

– обратного прослеживания.

Метод ручного тестирования. Это - самый простой и естественный способ данной группы. При обнаружении ошибки необходимо выполнить тестируемую программу вручную, используя тестовый набор, при работе с которым была обнаружена ошибка.

Метод очень эффективен, но не применим для больших программ, программ со сложными вычислениями и в тех случаях, когда ошибка связана с неверным представлением программиста о выполнении некоторых операций. Данный метод часто используют как составную часть других методов отладки.

Метод индукции. Метод основан на тщательном анализе симптомов ошибки, которые могут проявляться как неверные результаты вычислений или как сообщение об ошибке. Если компьютер просто «зависает», то фрагмент проявления ошибки вычисляют, исходя из последних полученных результатов и действий пользователя. Полученную таким образом информацию организуют и тщательно изучают, просматривая соответствующий фрагмент программы. В результате этих действий выдвигают гипотезы об ошибках, каждую из которых проверяют. Если гипотеза верна, то детализируют информацию об ошибке, иначе - выдвигают другую гипотезу. Последовательность выполнения отладки методом индукции показана на рисунке 14 в виде схемы алгоритма. Самый ответственный этап - выявление симптомов ошибки. Организуя данные об ошибке, целесообразно записать все, что известно о ее проявлениях, причем фиксируют, как ситуации, в которых фрагмент с ошибкой выполняется нормально, так и ситуации, в которых ошибка

проявляется. Если в результате изучения данных никаких гипотез не появляется, то необходима дополнительная информация об ошибке. Дополнительную информацию можно получить, например, в результате выполнения схожих тестов.

Рисунок 14 – Схема процесса отладки методом индукции

В процессе доказательства пытаются выяснить, все ли проявления ошибки объясняет данная гипотеза, если не все, то либо гипотеза не верна, либо ошибок несколько.

Метод дедукции. По методу дедукции вначале формируют множество причин, которые могли бы вызвать данное проявление ошибки. Затем анализируя причины, исключают те, которые противоречат имеющимся данным. Если все причины исключены, то следует выполнить дополнительное тестирование исследуемого фрагмента. В противном случае наиболее вероятную гипотезу пытаются доказать. Если гипотеза объясняет полученные признаки ошибки, то ошибка найдена, иначе - проверяют следующую причину (рисунок 15).

Метод обратного прослеживания. Для небольших программ эффективно применение метода обратного прослеживания. Начинают с точки вывода неправильного результата. Для этой точки строится гипотеза о значениях основных переменных, которые могли бы привести к получению имеющегося результата. Далее, исходя из этой гипотезы, делают предположения о значениях переменных в предыдущей точке. Процесс продолжают, пока не обнаружат причину ошибки.

Рисунок 15 – Схема процесса отладки методом дедукции

Методы и средства получения дополнительной информации. Для получения дополнительной информации об ошибке можно выполнить добавочные тесты или использовать специальные методы и средства:

– отладочный вывод;

– интегрированные средства отладки;

– независимые отладчики.

Отладочный вывод. Метод требует включения в программу дополнительного отладочного вывода в узловых точках. Узловыми считают точки алгоритма, в которых основные переменные программы меняют свои значения. Например, отладочный вывод следует предусмотреть до и после завершения цикла изменения некоторого массива значений. При этом предполагается, что, выполнив анализ выведенных значений, программист уточнит момент, когда были получены неправильные значения, и сможет сделать вывод о причине ошибки.

Данный метод не очень эффективен и в настоящее время практически не используется, так как в сложных случаях в процессе отладки может потребоваться вывод большого количества - «трассы» значений многих переменных, которые выводятся при каждом изменении. Кроме того, внесение в программы дополнительных операторов может привести к изменению проявления ошибки, что нежелательно, хотя и позволяет сделать определенный вывод о ее природе.

Примечание. Ошибки, исчезающие при включении в программу или удалению ид нее каких-либо «безобидных» операторов, как правило, связаны с «затиранием» памяти. В результате добавления или удаления операторов область затирания может сместиться в другое место и ошибка либо перестанет проявляться, либо будет проявляться по-другому.

Интегрированные средства отладки. Большинство современных сред программирования (Delphi, Builder C++, Visual Studio и т. д.) включают средства отладки, которые обеспечивают максимально эффективную отладку, Они позволяют:

– выполнять программу по шагам, причем как с заходом в подпрограммы, так и выполняя их целиком;

– предусматривать точки останова;

– выполнять программу до оператора, укачанного курсором;

– отображать содержимое любых переменных при пошаговом выполнении;

– отслеживать поток сообщений и т. п.

Тема 3.6 Тестирование пользовательского интерфейса

Цели, задачи и особенности тестирования пользовательского интерфейса

Функциональное тестирование пользовательских интерфейсов. Проверка требований к пользовательскому интерфейсу. Полнота покрытия. Методы проведения, повторяемость тестирования пользовательского интерфейса. Тестирование удобства использования пользовательских интерфейсов

Литература: [3]

Методические рекомендации

Часть программной системы, обеспечивающая работу интерфейса с пользователем - один из наиболее нетривиальных объектов для верификации. Нетривиальность заключается в двояком восприятии 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 соответствуют 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, слабо 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 лекции будут рассмотрены общие вопросы как функционального тестирования пользовательских интерфейсов, так и тестирования удобства использования.

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.

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 обычно 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.

Требования к пользовательскому интерфейсу могут быть разбиты на две группы:

– требования к внешнему виду пользовательского интерфейса и формам 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.

Тема 3.7 Тестирование объектно-ориентированных ПС

Объектно-ориентированное тестирование. Проектирование объектно-ориентированных тестовых вариантов. Тестирование содержания классов. Тестирование взаимодействия классов

Предваряющее тестирование при экстремальной разработке.

Особенности интеграционного тестирования объектно-ориентированных ПС

Литература: [8]

Методические рекомендации

Необходимость и важность тестирования ПО трудно переоценить. Вместе с тем следует отметить, что тестирование является сложной и трудоемкой деятельностью. Существует мнение, что объектно-ориентированное тестирование мало чем отличается от процедурно-ориентированного тестирования. Конечно, многие понятия, подходы и способы тестирования у них общие, но в целом это мнение ошибочно. Напротив, особенности объектно-ориентированных систем должны вносить и вносят существенные изменения как в последовательность этапов, так и в содержание этапов тестирования. Сгруппируем эти изменения по трем направлениям:

– расширение области применения тестирования;

– изменение методики тестирования;

– учет особенностей объектно-ориентированного ПО при проектировании тестовых вариантов.

Разработка объектно-ориентированного ПО начинается с создания визуальных моделей, отражающих статические и динамические ха-

рактеристики будущей системы. Вначале эти модели фиксируют исходные требования заказчика, затем формализуют реализацию этих требований путем выделения объектов, которые взаимодействуют друг с другом посредством передачи сообщений. Исследование моделей взаимодействия приводит к построению моделей классов и их отношений, составляющих основу логического представления системы. При переходе к физическому представлению строятся модели компоновки и размещения системы.

На конструирование моделей приходится большая часть затрат объектно-ориентированного процесса разработки. Если к этому добавить, что цена устранения ошибки стремительно растет с каждой итерацией разработки, то совершенно логично требование тестировать объектно-ориентированные модели анализа и проектирования. Критерии тестирования моделей: правильность, полнота, согласованность.

О синтаксической правильности судят по правильности использования языка моделирования (например, UML). О семантической правильности судят по соответствию модели реальным проблемам. Для определения того, отражает ли модель реальный мир, она оценивается экспертами, имеющими знания и опыт в конкретной проблемной области. Эксперты анализируют содержание классов, наследование классов, выявляя пропуски и неоднозначности. Проверяется соответствие отношений классов реалиям физического мира.

О согласованности судят путем рассмотрения противоречий между элементами в модели. Несогласованная модель имеет в одной части представления, которые противоречат представлениям в других частях модели.

Тема 3.8 Особенности тестирования Web-приложений

Цели, задачи и особенности тестированияWeb-приложений
Нагрузочное тестирование. Тестирование безопасности
Подходы к проектированию тестового окружения
Организация тестирования Web-приложений
Литература: [3]

Методические рекомендации

Вычислительные и коммуникационные системы используются все чаще и с каждым днем все глубже входят в нашу повседневную жизнь. Компании и отдельные пользователи все больше зависят в сво-

ей работе от web-приложений. Веб-приложения соединяют различные отделы внутри компаний, различные компании и простых пользователей. Веб-приложения очень динамичны, а их функциональные возможности непрерывно растут. Непрерывно возрастает потоковый трафик средств информации и запросов, формируемых переносными и встроенными устройствами. Вследствие этого возрастает сложность систем такого рода. Очевидно, что для понимания, анализа, разработки и управления такими системами нужны количественные методы и модели, которые помогают оценить различные сценарии функционирования, исследовать структуру и состояние больших систем. Наблюдаются тенденции к постоянному росту спроса на Веб-службы. Таким образом, проблемы, связанные с недостаточной производительностью будут возникать и в будущем, и, в конце концов, они станут превалирующими при планировании и вводе в эксплуатацию новых Веб-служб и увеличении пользователей Интернета. Веб-приложения становятся все более распространенными и все более сложными, играя, таким образом, основную роль в большинстве онлайновых проектов. Как и во всех системах, основанных на взаимодействии между клиентом и сервером, уязвимости Веб-приложений обычно возникают из-за некорректной обработки запросов клиента и/или недостаточной проверки входной информации со стороны разработчика.

Тестирование безопасности.

Это очень важный тип тестов, так как от безопасности сервера зависит практически все – и сам бизнес, и доверие пользователей, и сохранность информации. Правда, в отличие от других тестов, тестирование безопасности следует проводить регулярно. Кроме того, тестированию подвергается не только сам конкретный сайт или веб-приложение, а весь сервер полностью – и веб-сервер, и операционная система, и все сетевые сервисы. Как и в случае других тестов, программа "прикидывается" реальным пользователем-взломщиком и пытается применить к серверу все известные ей методы атаки и проверяет все уязвимости. Результатом работы будет отчет о найденных уязвимостях и рекомендации по их устранению.

Ошибки, связанные с проверкой корректности ввода, часто довольно сложно обнаружить в большом объеме кода, взаимодействующего с пользователем. Это является основной причиной того, что разработчики используют методологию тестирования приложений на проникновение для их обнаружения. Веб-приложения, однако, не имеют иммунитета и к более традиционным способам атаки. Весьма распространены плохие механизмы аутентификации, логические

ошибки, непреднамеренное раскрытие информации, а также такие традиционные ошибки для обычных приложений как переполнение буфера. Приступая к тестированию Веб-приложений, необходимо учитывать все перечисленное, и должен применяться методический процесс тестирования ввода/вывода по схеме "черного ящика" в сочетании (если возможно) с аудитом исходного кода.

Существуют различные инструменты позволяющие производить автоматическое тестирование безопасности, выполняя такие задачи как cross site scripting, SQL injection, включая переполнение буфера, подделка параметра, несанкционированный доступ, манипуляции с HTTP запросами и т.д.

Нагрузочное тестирование.

Следующим видом тестирования является тест на устойчивость к большим нагрузкам – Load-testing, stress-test или performance test. Такой тест имитирует одновременную работу нескольких сотен или тысяч посетителей (каждый из которых может "ходить" по сайту в соответствии со своим сценарием), проверяя, будет ли устойчивой работа сайта под большой нагрузкой. Кроме этого, можно имитировать кратковременные пики нагрузки, когда количество посетителей скачкообразно увеличивается – это очень актуально для новостных ресурсов и других сайтов с неравномерной аудиторией. В таком тесте проверяется не только и не столько сам сайт, сколько совместная слаженная работа всего комплекса – аппаратной части сервера, веб-сервера, программного ядра (engine) и других компонентов сайта.

Основными целями нагрузочного тестирования являются:

– оценка производительности и работоспособности приложения на этапе разработки и передачи в эксплуатацию;

– оценка производительности и работоспособности приложения на этапе выпуска новых релизов, патч-сетов;

– оптимизация производительности приложения, включая настройки серверов и оптимизацию кода;

– подбор соответствующей для данного приложения аппаратной (программной платформы) и конфигурации сервера.

Функциональное тестирование (functional testing) – процесс верификации соответствия функционирования продукта его начальным спецификациям. Характерным примером может быть проверка того, что программа подсчета выплат по банковской ссуде выдает корректные выкладки на любые введенные сумму ссуды и срок ее возврата. Обычно подобные проверки проводятся вручную, иногда к этому подключаются конечные пользователи в качестве бета-тестеров. Однако

программные системы становятся все сложнее, а комбинации различных входных параметров и поддерживаемых операционных систем нередко исчисляются десятками и сотнями.

Перечислим некоторые из методов функционального тестирования веб-приложений:

1 Record & Play – основан на возможности средств автоматизации тестирования автоматически генерировать код.

2 Functional Decomposition – в основе лежит разбиение всех компонент фреймворка по функциональному признаку на бизнес-функции (реализуют/проверяют бизнес-функциональность приложения), user-defined функции (вспомогательные функции, которые еще имеют привязку к тестируемому приложению или к конкретному проекту), утилиты (функции общего назначения, не привязанные к конкретному приложению, технологии, проекту).

3 Data-driven – основан на том, что к некоторому тесту или группе тестов привязывается источник данных, и этот тест или набор тестов циклически выполняется для каждой записи из этого источника данных. Вполне может применяться в комбинации с другими подходами.

4 Keyword-driven – представляет собой фактически движок для обработки посылаемых ему команд, а сами инструкции выносятся во внешний источник данных.

5 Object-driven – основан на том, что основные ходовые части фреймворка реализованы в виде объектов, что позволяет собирать тесты по кирпичикам.

6 Model-based – основан на том, что тестируемое приложение (или его части) описывается в виде некоторой поведенческой модели.

Самым распространенным является подход, называемый Capture & Playback (другие названия – Record & Playback, Capture & Replay). Суть этого подхода заключается в том, что сценарии тестирования создаются на основе работы пользователя с тестируемым приложением. Инструмент перехватывает и записывает действия пользователя, результат каждого действия также запоминается и служит эталоном для последующих проверок. При этом в большинстве инструментов, реализующих этот подход, воздействия (например, нажатие кнопки мыши) связываются не с координатами текущего положения мыши, а с объектами HTML-интерфейса (кнопки, поля ввода и т.д.), на которые происходит воздействие, и их атрибутами. При тестировании инструмент автоматически воспроизводит ранее записанные действия и сравнивает их результаты с эталонными, точность сравнения может

настраиваться. Можно также добавлять дополнительные проверки – задавать условия на свойства объектов (цвет, расположение, размер и т.д.) или на функциональность приложения (содержимое сообщения и т.д.).

Основное достоинство этого подхода – простота освоения. Создавать тесты с помощью инструментов, реализующих данный подход, могут даже пользователи, не имеющие навыков программирования.

Вместе с тем, у подхода имеется ряд существенных недостатков. Для разработки тестов не предоставляется никакой автоматизации; фактически, инструмент записывает процесс ручного тестирования. Если в процессе записи теста обнаружена ошибка, то в большинстве случаев создать тест для последующего использования невозможно, пока ошибка не будет исправлена (инструмент должен запомнить правильный результат для проверки). При изменении тестируемого приложения набор тестов трудно поддерживать в актуальном состоянии, так как тесты для изменившихся частей приложения приходится записывать заново.

Тема 3.9 Регрессионное тестирование

Цели, задачи и особенности регрессионного тестирования
Виды регрессионного тестирования
Управляемое регрессионное тестирование
Методы отбора тестов
Литература: [3]

Методические рекомендации

Поскольку регрессионное тестирование представляет собой повторное проведение цикла обычного тестирования, виды регрессионного тестирования совпадают с видами обычного тестирования. Можно говорить, например, о модульном регрессионном тестировании или о функциональном регрессионном тестировании.

Другой способ классификации видов регрессионного тестирования связывает их с типами сопровождения, которые, в свою очередь, определяются типами модификаций. Выделяют три типа сопровождения:

– Корректирующее сопровождение, называемое обычно исправлением ошибок, выполняется в ответ на обнаружение ошибки, не требующей изменения спецификации требований. При корректирующем

сопровождении производится диагностика и корректировка дефектов в программном обеспечении с целью поддержания системы в работоспособном состоянии.

– Адаптивное сопровождение осуществляется в ответ на требования изменения данных или среды исполнения. Оно применяется, когда существующая система улучшается или расширяется, а спецификация требований изменяется с целью реализации новых функций.

– Усовершенствующее (прогрессивное) сопровождение включает любую обработку с целью повышения эффективности работы системы или эффективности ее сопровождения.

В процессе адаптивного или усовершенствующего сопровождения обычно вводятся новые модули. Чтобы отобразить то или иное усовершенствование или адаптацию, изменяется спецификация системы. При корректирующем сопровождении, как правило, спецификация не изменяется, и новые модули не вводятся. Модификация программы на фазе разработки подобна модификации при корректирующем сопровождении, так как из-за обнаружения ошибки вряд ли требуется менять спецификацию программы. За исключением редких моментов крупных изменений, на фазе сопровождения изменения системы обычно невелики и производятся с целью устранения проблем или постепенного расширения функциональных возможностей.

Соответственно, определяют два типа регрессионного тестирования: прогрессивное и корректирующее.

Прогрессивное регрессионное тестирование предполагает модификацию технического задания. В большинстве случаев при этом к системе программного обеспечения добавляются новые модули.

При корректирующем регрессионном тестировании техническое задание не изменяется. Модифицируются только некоторые операторы программы и, возможно, конструкторские решения.

Прогрессивное регрессионное тестирование обычно выполняется после адаптивного или усовершенствующего сопровождения, тогда как корректирующее регрессионное тестирование выполняется во время тестирования в цикле разработки и после корректирующего сопровождения, то есть после того, как над программным обеспечением были выполнены некоторые корректирующие действия. Вообще говоря, корректирующее регрессионное тестирование должно быть более простым, чем прогрессивное регрессионное тестирование, поскольку допускает повторное использование большего количества тестов.

Подход к отбору регрессионных тестов может быть активным или консервативным. Активный подход во главу угла ставит умень-

шение объема регрессионного тестирования и пренебрегает риском пропустить дефекты. Активный подход применяется для тестирования систем с высокой исходной надежностью, а также в случаях, когда эффект изменений невелик. Консервативный подход требует отбора всех тестов, которые с ненулевой вероятностью могут обнаруживать дефекты. Этот подход позволяет обнаруживать большее количество ошибок, но приводит к созданию более обширных наборов регрессионных тестов.

Управляемое регрессионное тестирование.

В течение жизненного цикла программы период сопровождения длится долго. Когда измененная программа тестируется набором тестов T, мы сохраняем без изменений по отношению к тестированию исходной программы P все факторы, которые могли бы воздействовать на вывод программы. Поэтому атрибуты конфигурации, в которой программа тестировалась последний раз (например, план тестирования, тесты tj и покрываемые элементы MT(P,C,tj)), подлежат управлению конфигурацией. Практика тестирования измененной версии программы P' в тех же условиях, в которых тестировалась исходная программа P, называется управляемым регрессионным тестированием. При неуправляемом регрессионном тестировании некоторые свойства методов регрессионного тестирования могут изменяться, например, безопасный метод отбора тестов может перестать быть безопасным. В свою очередь, для обеспечения управляемости регрессионного тестирования необходимо выполнение ряда условий:

– Как при модульном, так и при интеграционном регрессионном тестировании в качестве модулей, вызываемых тестируемым модулем непосредственно или косвенно, должны использоваться реальные модули системы. Это легко осуществить, поскольку на этапе регрессионного тестирования все модули доступны в завершенном виде.

– Информация об изменениях корректна. Информация об изменениях указывает на измененные модули и разделы спецификации требований, не подразумевая при этом корректность самих изменений. Кроме того, при изменении спецификации требований необходимо усиленное регрессионное тестирование изменившихся функций этой спецификации, а также всех функций, которые могли быть затронуты по неосторожности. Единственным случаем когда мы вынуждены положиться на правильность измененного технического задания, является изменение технического задания для всей системы или для модуля верхнего (в графе вызовов) уровня, при условии, что кроме технического задания, не существует никакой дополнительной документации

и/или какой-либо другой информации, по которой можно было бы судить об ошибке в техническом задании.

– В программе нет ошибок, кроме тех, которые могли возникнуть из-за ее изменения.

– Тесты, применявшиеся для тестирования предыдущих версий программного продукта, доступны, при этом протокол прогона тестов состоит из входных данных, выходных данных и траектории. Траектория представляет собой путь в управляющем графе программы, прохождение которого вызывается использованием некоторого набора входных данных. Ее можно применять для оценки структурного покрытия, обеспечиваемого набором тестов.

– Для проведения регрессионного тестирования с использованием существующего набора тестов необходимо хранить информацию о результатах выполнения тестов на предыдущих этапах тестирования.

Предположим, что никакие операторы программы, кроме тех, чье поведение зависит от изменений, не могут неблагоприятно воздействовать на программу. Даже при таком условии существуют некоторые ситуации, требующие особого внимания, например проблема утечки памяти и ей подобные. Ситуации такого рода в разных системах программирования обрабатываются по-разному. Например, язык Java сам по себе включает систему управления памятью. Если же система не контролирует распределение памяти автоматически, мы должны считать, что все операторы работы с памятью также обладают поведением, зависящим от изменений.

Проблема языков типа C и C++, которые допускают произвольные арифметические операции над указателями, состоит в том, что указатели могут нарушать границы областей памяти, на которые они указывают. Это означает, что переменные могут обрабатываться способами, которые не поддаются анализу на уровне исходного кода. Чтобы учесть такие нарушения границ памяти, выдвигаются следующие гипотезы:

Гипотеза 1 (четко определенная память). Каждый сегмент памяти, к которому обращается система программного обеспечения, соответствует некоторой символически определенной переменной.

Гипотеза 2 (строго ограниченный указатель). Каждая переменная или выражение, используемое как указатель, должно ссылаться на некоторую базовую переменную и ограничиваться использованием сегмента памяти, определяемого этой переменной.

Чтобы гарантировать покрытие всех зависящих от изменений компонентов, для которых можно показать, что они затрагиваются су-

ществующими тестами, достаточно одного теста для каждого из таких компонентов. Множество тестов достаточно большого размера (как правило сценарных), может способствовать обнаружению ошибок, вызванных нарушениями условий управляемого регрессионного тестирования.

Существуют и организационные условия проведения регрессионного тестирования. Это ресурс (время), необходимый тестовому аналитику для ознакомления со спецификацией требований системы, ее архитектурой и, возможно, самим кодом.

Тема 3.10 Документирование как основа тестирования

Документация, создаваемая на различных этапах жизненного цикла

Документация, сопровождающая процессы верификации и тестирования

Стратегия и планы верификации. Тест-требования. Тест-планы. Отчеты о прохождении тестов. Отчеты о покрытии программного кода. Отчеты о проблемах. Трассировочные таблицы

Литература: [3]

Методические рекомендации

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

Результатом этапа разработки требований к системе являются сформулированные требования к системе: документы, 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.

На рисунке 16 представлены процессы и документы при разработке программных систем

Рисунок 16 – 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 пользователя. Текст требования, как правило, включает в себя слова "должна, должен" и имеет структуру вида "В случае, если значение температуры на датчике АВС до-

стигает 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 АВС ниже 30 градусов Цельсия система выдает предупреждающий звуковой сигнал" и "Проверить, что в случае, когда значение температуры на датчике АВС выше 30 градусов Цельсия, система не выдает звуковой сигнал".

Одна из проблем, возникающая 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 АВС 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 быть подготовен либо в виде программы на каком-либо языке программирования, либо в виде входного файла данных для инструментария, который выполняет тестируемую систему и передает ей значения, указанные в тест-плане, либо в виде инструкций для пользователя систе-

мы, 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. Непройденные 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.

Вопросы для самоконтроля

1 Опишите на какие этапы разделяется реализация тестирования?

2 Опишите задачи модульного тестирования.

3 Опишите организацию интеграционного тестирования.

Какие методы существуют для проведения интеграционного тестирования.

4 Дайте определение понятию системное тестирование

5 Перечислите виды системного тестирования.

6 Опишите классификацию ошибок.

7 Как работает метод ручного тестирования?

8 Из каких фаз состоит функциональное тестирование пользовательского интерфейса?

9 Назовите особенности интеграционного тестирования объектно-ориентированных ПС.

10 Назовите основные цели нагрузочного тестирования.

11 Какие для обеспечения управляемости регрессионного тестирования необходимо выполнить условия.

12 Назовите процессы и документы при разработке программных.

Раздел 4 Тестирование при промышленной разработке ПО
Тема 4.1 Автоматизация тестирования

Понятие решения об автоматизации тестирования. Особенности организации автоматизированного тестирования

Инструментальные средства автоматизированного тестирования

Планирование тестирования. Разработка тестов. Проведение тестирования. Анализ результатов

Литература: [3]

Методические рекомендации

Использование различных подходов к тестированию определяется их эффективностью применительно к условиям, определяемым промышленным проектом. В реальных случаях работа группы тестирования планируется так, чтобы разработка тестов начиналась с момента согласования требований к программному продукту (выпуск Requirement Book, содержащей высокоуровневые требования к продукту) и продолжалась параллельно с разработкой дизайна и кода продукта. В результате, к началу системного тестирования создаются тестовые наборы, содержащие тысячи тестов. Большой набор тестов обеспечивает всестороннюю проверку функциональности продукта и гарантирует качество продукта, но пропуск такого количества тестов на этапе системного тестирования представляет проблему. Ее решение лежит в области автоматизации тестирования, т.е. в автоматизации разработки.

Структура программы P теста

Загрузка теста (X,Y*)

Запуск тестируемого модуля

Сравнение полученных результатов Y с эталонными Y*

Структура тестируемого комплекса

ModF <- ModF1

ModF2

ModF3 <- ModF31

ModF32

Структура тестирующего модуля

Mod TestModF:

Mod TestМодF1

Mod TestМодF2

```
Mod TestМодF3
P TestМодF

Mod  TestModF1:
P TestМодF1

Mod  TestModF2:
P TestМодF2

Mod  TestModF3:
Mod TestМодF31
Mod TestМодF32
P TestМодF3
```

В этом примере приведены структура теста, структура тестируемого комплекса и структура тестирующего модуля. Особенностью структуры каждого из тестирующих модулей Mi является запуск тестирующей программы Pi после того как каждый из модулей Mij, входящих в контекст модуля Mi, оттестирован. В этом случае запуск тестирующего модуля обеспечивает рекурсивный спуск к программам тестирования модулей нижнего уровня, а затем исполняет тестирование вышележащих уровней в условиях оттестированности нижележащих. Тестовые наборы подобной структуры ориентированы на автоматическое управление пропуском тестового набора в тестовом цикле. Важным преимуществом подобной организации является возможность регулирования нижнего уровня, до которого следует доходить в цикле тестирования. В этом случае контекст редуцированных в конкретном тестовом цикле модулей помечается как базовый, не подлежащий тестированию. Например, если контекст модуля ModF3: (ModF31, ModF32) – помечен как базовый, то в результате рекурсивный спуск затронет лишь модули ModF1, ModF2, ModF3 и вышележащий модуль ModF. Описанный способ организации тестовых наборов с успехом применяется в системах автоматизации тестирования.

Собственно использование эффективной системы автоматизации тестирования сокращает до минимума (например, до одной ночи) время пропуска тестов, без которого невозможно подтвердить факт роста качества (уменьшения числа оставшихся ошибок) продукта. Системное тестирование осуществляется в рамках циклов тестирования (периодов пропуска разработанного тестового набора над build разрабатываемого приложения). Перед каждым циклом фиксируется разработанный или исправленный build, на который заносятся обнаруженные

в результате тестового прогона ошибки. Затем ошибки исправляются, и на очередной цикл тестирования предъявляется новый build. Окончание тестирования совпадает с экспериментально подтвержденным заключением о достигнутом уровне качества относительно выбранного критерия тестирования или о снижении плотности не обнаруженных ошибок до некоторой заранее оговоренной величины. Возможность ограничить цикл тестирования пределом в одни сутки или несколько часов поддерживается исключительно за счет средств автоматизации тестирования.

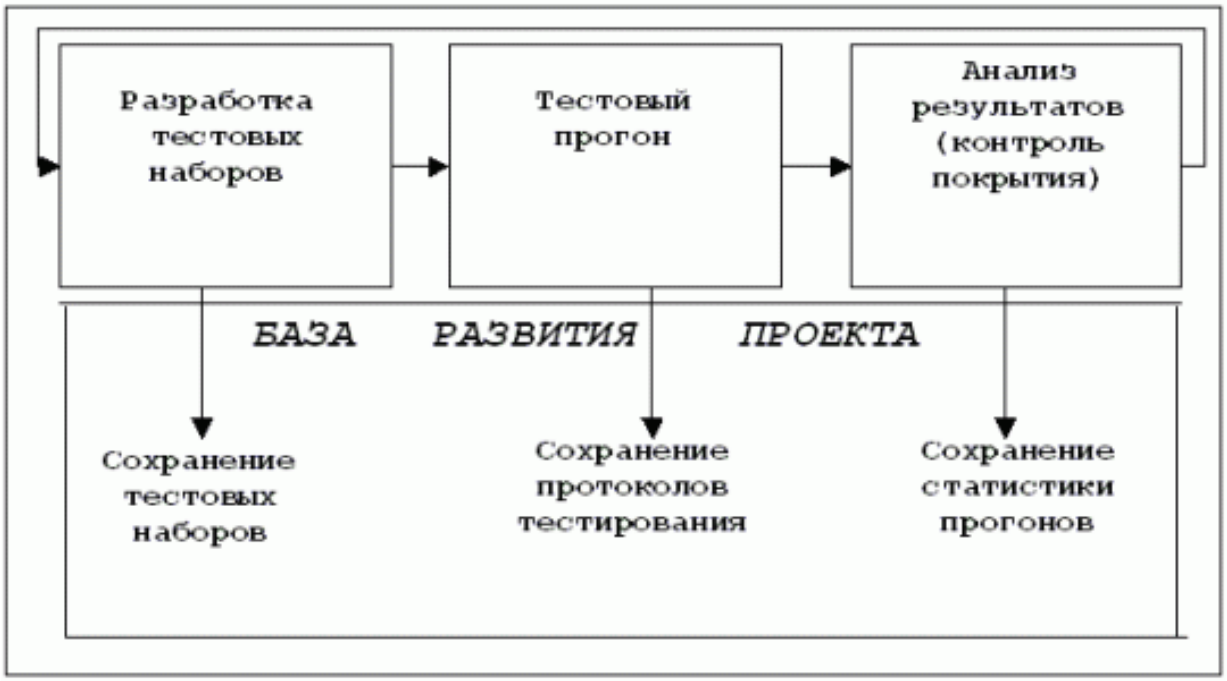

Рисунок 17 – Структура инструментальной системы автоматизации тестирования

На рисунке 17 представлена обобщенная структура системы автоматизации тестирования, в которой создается и сохраняется следующая информация:

1 Набор тестов, достаточный для покрытия тестируемого приложения в соответствии с выбранным критерием тестирования – как результат ручной или автоматической разработки (генерации) тестовых наборов и драйвер/монитор пропуска тестового набора.

2 Результаты прогона тестового набора, зафиксированные в Log-файле. Log-файл содержит трассы ("протоколы"), представляющие собой реализованные при тестовом прогоне последовательности некоторых событий (значений отдельных переменных или их совокупностей) и точки реализации этих событий на графе программы. В составе трасс могут присутствовать последовательности явно и неявно заданных ме-

ток, задающих пути реализации трасс на управляющем графе программы, совокупности значений переменных на этих метках, величины промежуточных результатов, достигнутых на некоторых метках и т.п.

3 Статистика тестового цикла, содержащая: 1) результаты пропуска каждого теста из тестового набора и их сравнения с эталонными величинами; 2) факты, послужившие основанием для принятия решения о продолжении или окончании тестирования; 3) критерий покрытия и степень его удовлетворения, достигнутая в цикле тестирования.

Результатом анализа каждого прогона является список проблем, в виде ошибок и дефектов, который заносится в базу развития проекта. Далее происходит работа над ошибками, где каждая поднятая проблема идентифицируется, относится к соответствующему модулю и разработчику, приоритезируется и отслеживается, что обеспечивает гарантию ее решения (исправления или отнесения к списку известных проблем, решение которых по тем или иным причинам откладывается) в последующих build. Исправленный и собранный для тестирования build поступает на следующий цикл тестирования, и цикл повторяется, пока нужное качество программного комплекса не будет достигнуто. В этом итерационном процессе средства автоматизации тестирования обеспечивают быстрый контроль результатов исправления ошибок и проверку уровня качества, достигнутого в продукте. Некачественный продукт зрелая организация не производит.

Тема 4.2 Особенности индустриального тестирования

Индустриальный подход, его особенности

Качество программного продукта и его тестирование

Процесс индустриального тестирования. Планирование тестирования

Подходы к разработке тестов

Литература: [3]

Методические рекомендации

Качество программного продукта можно оценить некоторым набором характеристик, определяющих, насколько продукт "хорош" с точки зрения всех потенциально заинтересованных в нем сторон. Такими сторонами являются:

– заказчик продукта

– спонсор
– конечный пользователь
– разработчики продукта
– тестировщики продукта
– инженеры поддержки
– отдел обучения
– отдел продаж и т.п.

Каждый из участников может иметь различное представление о продукте и по-разному судить о том, насколько он хорош или плох, то есть насколько высоко качество продукта. С точки зрения разработчика, продукт может быть настолько хорош, насколько хороши заложенные в нем алгоритмы и технологии. Пользователю продукта, скорее всего, безразличны детали внутренней реализации, его в первую очередь волнуют вопросы функциональности и надежности. Спонсора интересует цена и совместимость с будущими технологиями. Таким образом, задача обеспечения качества продукта выливается в задачу определения заинтересованных лиц, согласования их критериев качества и нахождения оптимального решения, удовлетворяющего этим критериям.

В рамках подобной задачи группа тестирования рассматривается не просто как еще одна заинтересованная сторона, но и как сторона, способная оценить удовлетворение выбранных критериев и сделать вывод о качестве продукта с точки зрения других участников. К сожалению, далеко не все критерии могут быть оценены группой тестирования. Поэтому ее внимание в основном сосредоточено на критериях, определяющих качество программного продукта с точки зрения конечного пользователя.

Тестирование как способ обеспечения качества. Тестирование, с технической точки зрения, есть процесс выполнения приложения на некоторых входных данных и проверка получаемых результатов с целью подтвердить их корректность по отношению к результату.

Тестирование не позиционируется в качестве единственного способа обеспечения качества. Оно является частью общей системы обеспечения качества продукта, элементы которой выбираются по критерию наибольшей эффективности применения в конкретном проекте.

В каждом конкретном проекте элементы системы должны быть выбраны так, чтобы обеспечить приемлемое качество, исходя из приоритетов и имеющихся ресурсов. Выбирая элементы для системы обеспечения качества конкретного продукта, можно применить комбини-

рованное тестирование, обзоры кода, аудит. При подобном выборе некоторые качества, например легкость модификации и исправления дефектов, не будут оценены и, возможно, выполнены. Задачей тестирования в рассматриваемом случае будет обнаружение дефектов и оценка удобства использования продукта, включая полноту функциональности. Исходя из задач, поставленных перед группой тестирования в конкретном проекте, выбирается соответствующая стратегия тестирования. Так, в данном примере, ввиду необходимости оценить удобство использования и полноту функциональности, преимущественный подход к разработке тестов следует планировать на основе использования сценариев.

Итак, основная последовательность действий при выборе и оценке критериев качества программного продукта включает:

1 Определение всех лиц, так или иначе заинтересованных в исполнении и результатах данного проекта.

2 Определение критериев, формирующих представление о качестве для каждого из участников.

3 Приоритезацию критериев, с учетом важности конкретного участника для компании, выполняющей проект, и важности каждого из критериев для данного участника.

4 Определение набора критериев, которые будут отслежены и выполнены в рамках проекта, исходя из приоритетов и возможностей проектной команды. Постановка целей по каждому из критериев.

5 Определение способов и механизмов достижения каждого критерия.

6 Определение стратегии тестирования исходя из набора критериев, попадающих под ответственность группы тестирования, выбранных приоритетов и целей.

Вопросы для самоконтроля

1 Постройте структуру инструментальной системы автоматизации тестирования.

2 Перечислите особенности организации автоматизированного тестирования.

3 Опишите процесс индустриального тестирования.

4 Перечислите основную последовательность действий при выборе и оценке критериев качества программного продукта.

Список используемых источников

1 Котляров, В.П. Основы тестирования программного обеспечения / В.П.Котляров. – М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2006.

2 https://qalight.com.ua/baza-znaniy/otkuda-berutsya-oshibki-v-po/

3 https://www.intuit.ru/studies/courses/1040/209/lecture/5380

4 http://www.protesting.ru/testing/types/regression.html

5 Липаев, В.В. Тестирование компонентов и комплексов программ / В.В.Липаев. – Москва: СИНТЕГ, 2010.

6 http://textarchive.ru/c-1144105-p16.html

7http://lib.kstu.kz:8300/tb/books/2015/ITB/Sovremennye%20metody%20i%20sredstva%20sozdaniya%20PO/teory/7.htm

8 http://netnado.ru/dokumenti-vhfg1/addfile-12/index.html

Задания на домашнюю контрольную работу по учебной дисциплине «Тестирование и отладка программного обеспечения»

1 Эволюция методов тестирования ПО. Качество ПО.

2 Модели жизненного цикла ПО и роль тестирования в них.

3 Тестирование, верификация и валидация, их различия.

4 Свойства тестирования.

5 Взаимосвязь уровней и целей тестирования ПО.

6 Типы дефектов и ошибок ПО, их жизненный цикл.

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

8 Модульное тестирование.

9 Интеграционное тестирование.

10 Системное тестирование.

11 Нагрузочное тестирование.

12 Формальные инспекции.

13 Регрессионное тестирование.

14 Комбинирование уровней тестирования.

15 Требования к идеальному критерию.

16 Классы критериев.

17 Структурные критерии.

18 Функциональные критерии.

19 Стохастические критерии.

20 Мутационный критерий.

21 Оценка покрытия программы и проекта.

22 Методика интегральной оценки тестированности.

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

24 Анализ функциональных требований к ПО и определение процедуры тестирования.

25 Анализ эксплуатационных требований к ПО и определение процедуры тестирования.

26 Методы тестирования. «Черный ящик».

27 Методы тестирования. «Белый (стеклянный) ящик».

28 Методы тестирования. Тестирование моделей.

29 Методы тестирования. Анализ программного кода (инспекции).

30 Текстовое окружение: драйверы и заглушки, генераторы сигналов (событийно-управляемый код).

31 Цели и задачи тестирования программного кода.

32 Инспекция кода и прогон.

33 Операторное покрытие и покрытие ветвлений.

34 Покрытие условий и путей.

35 Граф управления потоками.

36 Метрика МакКейба.

37 Базовый метод построения независимых путей для структурного тестирования.

38 Тест-примеры, их типы.

39 Проверка на граничных значениях и робастности.

40 Классы эквивалентности.

41 Оценка покрытия программного кода тестами.

42 Тест-план, его типовая структура.

43 Свойства потоков данных программных модулей.

44 Тестирование потоков данных программных модулей.

45 Тестирование графов модулей программ с учетом значений переменных и констант.

46 Тестирование циклов.

47 Цели и задачи обеспечения повторяемости тестирования.

48 Предусловия для выполнения теста.

49 Настройка тестового окружения, оптимизация последовательностей тестовых примеров.

50 Зависимость между тестовыми примерами.

51 Настройки по умолчанию для тестовых примеров и их групп.

52 Функциональное тестирование ПО (тестирование методом «черного ящика»).

53 Разбиение на классы эквивалентности.

54 Анализ граничных значений.

55 Функциональные диаграммы.

56 Тестирование с помощью функциональных диаграмм.

57 Организация тестирования. Три фазы тестирования.

58 Методика тестирования программных систем (ПС).

59 Нисходящая и восходящая стратегии тестирования.

60 Особенности тестового окружения для нисходящей и восходящей стратегии тестирования.

61 Цели и задачи модульного тестирования.

62 Понятие о модуле и его границах.

63 Подходы к проектированию тестового окружения.

64 Организация модульного тестирования.

65 Разработка плана и проведение модульного тестирования ПО.

66 Цели и задачи интеграционного тестирования.

67 Организация интеграционного тестирования.

68 Структурная классификация методов интеграционного тестирования.

69 Временная классификация методов интеграционного тестирования.

70 Планирование интеграционного тестирования.

71 Разработка плана и проведение интеграционного тестирования ПО.

72 Цели и задачи системного тестирования.

73 Виды системного тестирования.

74 Системное тестирование, приемо-сдаточные и сертификационные испытания при разработке сертифицируемого программного обеспечения.

75 Разработка плана и проведение системного тестирования ПО.

76 Отладка ПО. Классификация ошибок.

77 Отладка ПО. Методы отладки ПО.

78 Отладка ПО. Методы и средства получения дополнительной информации.

79 Отладка ПО. Общая методика отладки ПО.

80 Цели и задачи тестирования пользовательского интерфейса.

81 Особенности тестирования пользовательского интерфейса.

82 Функциональное тестирование пользовательских интерфейсов.

83 Проверка требований к пользовательскому интерфейсу. Полнота покрытия.

84 Методы проведения, повторяемость тестирования пользовательского интерфейса.

85 Тестирование удобства использования пользовательских интерфейсов.

86 Разработка плана и проведение тестирования пользовательского интерфейса.

87 Объектно-ориентированное тестирование. Проектирование объектно-ориентированных тестовых вариантов.

88 Объектно-ориентированное тестирование. Тестирование содержания классов. Тестирование взаимодействия классов.

89 Объектно-ориентированное тестирование. Предваряющее тестирование при экстремальной разработке.

90 Особенности интеграционного тестирования объектно-ориентированных ПС.

91 Разработка плана и проведение тестирования объектно-ориентированных ПС.

92 Цели и задачи тестированияWeb-приложений.

93 Особенности тестированияWeb-приложений

94 Нагрузочное тестирование.

95 Тестирование безопасности.

96 Подходы к проектированию тестового окружения.

97 Организация тестирования Web-приложений.

98 Цели и задачи регрессионного тестирования.

99 Особенности регрессионного тестирования

100 Виды регрессионного тестирования.

101 Управляемое регрессионное тестирование.

102 Методы отбора тестов.

103 Документация, создаваемая на различных этапах жизненного цикла.

104 Документация, сопровождающая процессы верификации и тестирования.

105 Стратегия и планы верификации.

106 Тест-требования.

107 Тест-планы.

108 Отчеты о прохождении тестов.

109 Отчеты о покрытии программного кода. Отчеты о проблемах.

110 Трассировочные таблицы.

111 Понятие решения об автоматизации тестирования.

112 Особенности организации автоматизированного тестирования.

113 Инструментальные средства автоматизированного тестирования.

114 Планирование тестирования.

115 Разработка тестов. Проведение тестирования. Анализ результатов.

116 Индустриальный подход, его особенности.

117 Качество программного продукта и его тестирование.

118 Процесс индустриального тестирования.

119 Планирование тестирования.

120 Подходы к разработке тестов.

Таблица 3 – Варианты заданий на домашнюю контрольную работу по учебной дисциплине «Тестирование и отладка программного обеспечения»

Предпослед-няя цифра шифра	Последняя цифра шифра									
	0	1	2	3	4	5	6	7	8	9
0	1,61, 81,101	2,62, 82,102	3,63, 83,103	4,104, 84,64	5,105, 85,66	6,106, 86,67	7,107, 87,65	8,108, 88,70	9,109, 89,71	10,110, 90,72
1	11,73, 91,111	12,66, 92,112	13,67, 93,113	14,114, 94,68	15,115, 95,74	16,116, 96,77	17,117, 97,76	18,118, 98,79	19,119, 99,83	20,120, 100,84
2	1,21, 80,102	2,22, 81,103	3,23, 82,104	4,21, 85,101	5,25, 88,106	6,26, 90,107	7,27 91,108	8,28, 92,109	9,29, 93,110	10,30, 94,111
3	11,31, 95,112	12,31, 96,113	13,33, 97,114	14,34, 98,115	15,35, 99,116	16,36, 100,117	17,37, 101,118	18,38, 102,119	19,39, 103,120	1,20, 40,104
4	2, 21, 41,120	3,22, 42,119	4,23, 43,118	5,24, 44,117	6,25, 45,116	7,26, 46,115	8,27, 47,114	9,28, 48,119	10,29, 49,112	11,30, 50,111
5	12,31, 51, 110	13,32, 52,109	14,33, 53,108	15,34, 54,107	16,35, 55,106	17,36, 56,105	18,37, 57,104	19,38, 58,104	20,39, 59,103	1,21, 40,60
6	2, 22, 41, 61	3,23, 42,62	7,24, 43,63	8,25, 44,64	9,26, 45,65	10,27, 46,66	11,28, 47,67	12,29, 48,68	13,30, 49,69	14,31, 50,70
7	15,32, 51,71	16,33, 52,72	17,34, 53,73	18,35, 54,74	19,36, 55,75	37,56, 76,120	21,38, 57,77	23,39, 58,78	24,40, 59,79	20,41, 60,80
8	21,42, 61,81	22,43, 62,82	23,44, 63,83	24,45, 64,84	25,46, 65,85	26,47, 66,86	27,48, 67,87	28,49, 68,88	29,50, 69,89	30,51, 70,90
9	31,52, 71,91	32,53, 72,92	33,54, 73,93	34,55, 74,94	35,56, 75,95	36,57, 76,96	37,58, 77,97	38,59, 78,98	39,59, 79,99	40,60, 80,100