МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

ГЛАВНОЕ УПРАВЛЕНИЕ ПО ОБРАЗОВАНИЮ
МОГИЛЕВСКОГО ОБЛАСТНОГО ИСПОЛНИТЕЛЬНОГО КОМИТЕТА

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«МОГИЛЕВСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ КОЛЛЕДЖ»

УТВЕРЖДАЮ
Директор колледжа
_____ С.Н.Козлов
22.04.2021

ТЕСТИРОВАНИЕ 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»

2021

Автор: Васюкевич Е.В., преподаватель учреждения образования «Могилевский государственный политехнический колледж»

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

Разработано на основе учебной программы по учебной дисциплине «Тестирование программного обеспечения», утвержденной директором УО «Могилевский государственный политехнический колледж», 25.08.2020

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

Протокол № _____ от _____

Пояснительная записка

Учебная программа по учебной дисциплине «Тестирование программного обеспечения» предусматривает изучение видов, назначения и применения методов тестирования программного обеспечения.

В процессе изучения учебной дисциплины у учащихся формируются профессиональные компетенции, благодаря которым они, используя современные программные продукты, смогут осуществлять организацию, планирование, подготовку и проведение тестирования программного обеспечения, а также управление процессом тестирования, а также организовывать тестирование в рамках производственных процессов разработки программного обеспечения.

Цель преподавания учебной дисциплины – развитие профессиональной компетентности в области тестирования сложных программных средств.

Изучение программного учебного материала базируется на знаниях, полученных учащимися в ходе изучения таких учебных дисциплин, как «Основы алгоритмизации и программирование», «Конструирование программ и языки программирования», «Технология разработки программного обеспечения».

Учебная дисциплина «Тестирование программного обеспечения» является практикоориентированной. Для закрепления теоретического материала и формирования у учащихся необходимых умений и навыков учебной программой предусматривается проведение лабораторных и практических занятий. Форма проведения лабораторных занятий определяется преподавателем исходя из цели обучения и содержания учебного материала.

В целях контроля усвоения программного учебного материала предусматривается проведение двух обязательных контрольных работ (ОКР) и экзамена. Содержание и конкретные сроки проведения ОКР определяются преподавателем, обсуждаются на заседании цикловой комиссии и утверждаются в установленном порядке.

Программой определены цели изучения каждой темы, спрогнозированы результаты их достижения в соответствии с уровнями усвоения учебного материала.

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 требований, свойства хороших требований;

особенности документирования дефектов с использованием систем отслеживания ошибок;

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 веб-приложений и мобильных приложений;

основы тестирования безопасности, производительности, регрессионного тестирования;

особенности выполнения автоматизированного тестирования;

уметь:

проводить тестирование структуры программных модулей и их взаимодействия;

проводить тестирование требований к программному обеспечению;

выполнять разработку тестовых сценариев; выполнять разработку use-cases;

выполнять разработку чек-листа и тест-кейсов;

составлять отчеты об ошибках;

проводить отладку и функциональное тестирование веб-ориентированных приложений;

проводить функциональное тестирование мобильных приложений;

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

В учебной программе приведены примерные критерии оценки результатов учебной деятельности учащихся по учебной дисциплине, которые разработаны на основе десятибалльной шкалы и показателей оценки результатов учебной деятельности учащихся в учреждениях среднего специального образования.

Учебная программа содержит примерный перечень оснащения лаборатории техническими и программными средствами, а также список использованных источников, необходимых для обеспечения образовательного процесса.

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 1.

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 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. Направления тестирования. Уровни тестирования

Эволюция методов тестирования ПО. Качество ПО.
Модели жизненного цикла ПО и роль тестирования в них.
Направления тестирования.
Уровни тестирования: модульное, интеграционное, приемочное.
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.

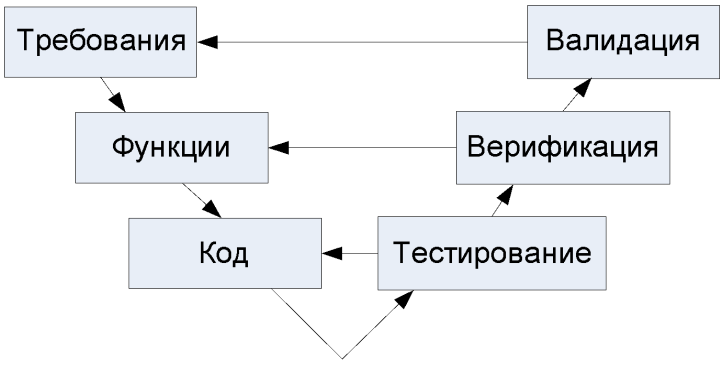

Рисунок 1 – Тестирование, верификация и валидация

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.

Тема 1.2 Виды и методы тестирования. Критерии выбора тестов и оценки качества ПО

Методы тестирования: «черный ящик», «белый ящик», тестирование моделей, анализ программного кода.

Виды тестирования.

Критерии выбора тестов. Классы критериев.

Оценка качества ПО.

Литература: [3]; [4]

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.

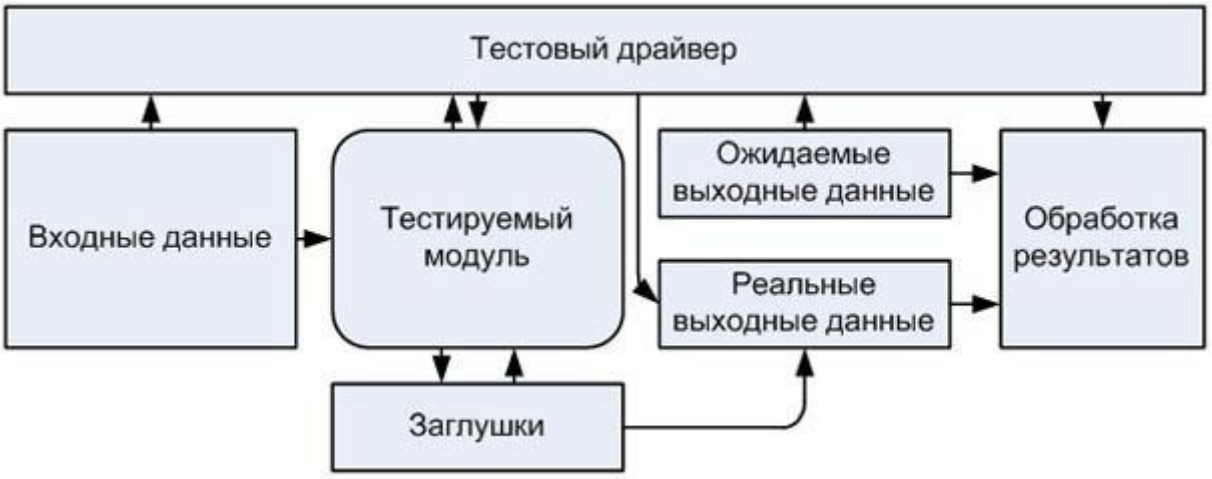

Рисунок 2 – Обобщенная схема среды тестирования

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.

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 1.3 Типы и уровни требований. Выявление требований

Требования, предъявляемые к ПО.
Типы и уровни требований.
Выявление требований
Литература: [1], с. 27-61

Методические рекомендации

Требование — описание того, какие функции и с соблюдением каких условий должно выполнять приложение в процессе решения полезной для пользователя задачи.

Требования являются отправной точкой для определения того, что проектная команда будет проектировать, реализовывать и тестировать. Элементарная логика говорит нам, что если в требованиях что-то «не то», то и реализовано будет «не то», т.е. колоссальная работа множества людей будет выполнена впустую.

Бизнес-требования выражают цель, ради которой разрабатывается продукт (зачем вообще он нужен, какая от него ожидается польза). Результатом выявления требований на этом уровне является общее видение — документ, который, как правило, представлен простым текстом и таблицами. Здесь нет детализации поведения системы и иных технических характеристик, но вполне могут быть определены приоритеты решаемых бизнес-задач, риски и т.п.

Пользовательские требования описывают задачи, которые пользователь может выполнять с помощью разрабатываемой системы (реакцию системы на действия пользователя, сценарии работы пользователя). Поскольку здесь уже появляется описание поведения системы, требования этого уровня могут быть использованы для оценки объёма работ, стоимости проекта, времени разработки и т.д. Пользовательские требования оформляются в виде вариантов использования, пользовательских историй, пользовательских сценариев.

Бизнес-правила описывают особенности принятых в предметной области (и/или непосредственно у заказчика) процессов, ограничений и иных правил. Эти правила могут относиться к бизнес-процессам, правилам работы сотрудников, нюансам работы ПО и т.д.

Атрибуты качества расширяют собой нефункциональные требования и на уровне пользовательских требований могут быть представлены в виде описания ключевых для проекта показателей качества (свойств продукта, не связанных с функциональностью, но являющихся важными для достижения целей создания продукта — производительность, масштабируемость, восстанавливаемость). Атрибутов качества очень много76, но для любого проекта реально важными является лишь некоторое их подмножество.

Ограничения представляют собой факторы, ограничивающие выбор способов и средств (в том числе инструментов) реализации продукта.

Требования к интерфейсам описывают особенности взаимодействия разрабатываемой системы с другими системами и операционной средой.

Требования к данным описывают структуры данных (и сами данные), являющиеся неотъемлемой частью разрабатываемой системы. Часто сюда относят описание базы данных и особенностей её использования.

Требования начинают свою жизнь на стороне заказчика. Их сбор и выявление осуществляются с помощью основных техник, представленных на рисунке 3.

Рисунок 3 – Основные техники сбора и выявления требований

Тема 1.4 Тестирование требований, техники работы с требованиями

Требования к программному продукту.
Тестирование требований, техники работы с требованиями
Литература: [1], с. 27-61

Методические рекомендации

Тестирование документации и требований относится к разряду нефункционального тестирования. Основные техники такого тестирования в контексте требований таковы.

Взаимный просмотр. Взаимный просмотр («рецензирование») является одной из наиболее активно используемых техник тестирования требований и может быть представлен в одной из трёх следующих форм (по мере нарастания его сложности и цены):

− Беглый просмотр может выражаться как в показе автором своей работы коллегам с целью создания общего понимания и получения обратной связи, так и в простом обмене результатами работы между двумя и более авторами с тем, чтобы коллега высказал свои вопросы и замечания. Это самый быстрый, самый дешёвый и наиболее широко используемый вид просмотра. Для запоминания: аналог беглого просмотра — это ситуация, когда вы в школе с одноклассниками проверяли перед сдачей сочинения друг друга, чтобы найти описки и ошибки;

− Технический просмотр выполняется группой специалистов. В идеальной ситуации каждый специалист должен представлять свою область знаний. Просматриваемый продукт не может считаться достаточно качественным, пока хотя бы у одного просматривающего остаются замечания. Для запоминания: аналог технического просмотра — это ситуация, когда некий договор визирует юридический отдел, бухгалтерия и т.д;

− Формальная инспекция представляет собой структурированный, систематизированный и документируемый подход к анализу документации. Для

его выполнения привлекается большое количество специалистов, само выполнение занимает достаточно много времени, и потому этот вариант просмотра используется достаточно редко (как правило, при получении на сопровождение и доработку проекта, созданием которого ранее занималась другая компания). Для запоминания: аналог формальной инспекции — это ситуация генеральной уборки квартиры (включая содержимое всех шкафов, холодильника, кладовки и т.д.).

Вопросы. Следующей очевидной техникой тестирования и повышения качества требований является (повторное) использование техник выявления требований, а также (как отдельный вид деятельности) — задавание вопросов. Если хоть что-то в требованиях вызывает у вас непонимание или подозрение — задавайте вопросы. Можно спросить представителей заказчика, можно обратиться к справочной информации. По многим вопросам можно обратиться к более опытным коллегам при условии, что у них имеется соответствующая информация, ранее полученная от заказчика. Главное, чтобы ваш вопрос был сформулирован таким образом, чтобы полученный ответ позволил улучшить требования.

Тест-кейсы и чек-листы. Мы помним, что хорошее требование является проверяемым, а значит, должны существовать объективные способы определения того, верно ли реализовано требование. Продумывание чек-листов или даже полноценных тест-кейсов в процессе анализа требований позволяет нам определить, насколько требование проверяемо. Если вы можете быстро придумать несколько пунктов чек-листа, это ещё не признак того, что с требованием всё хорошо (например, оно может противоречить каким-то другим требованиям). Но если никаких идей по тестированию требования в голову не приходит — это тревожный знак.

Исследование поведения системы. Эта техника логически вытекает из предыдущей (продумывания тест-кейсов и чек-листов), но отличается тем, что здесь тестированию подвергается, как правило, не одно требование, а целый набор. Тестировщик мысленно моделирует процесс работы пользователя с системой, созданной по тестируемым требованиям, и ищет неоднозначные или вовсе неописанные варианты поведения системы. Этот подход сложен, требует достаточной квалификации тестировщика, но способен выявить нетривиальные недоработки, которые почти невозможно заметить, тестируя требования по отдельности.

Вопросы для самоконтроля

1Дайте определение понятиям отладка, тестирование.
2Опишите взаимосвязь тестирования, валидации и верификации.
3Дайте понятие термину верификация.
4Дайте понятие термину валидация.
5Перечислите требования к идеальному критерию тестирования.
6Перечислите классы критериев.
7Перечислите и опишите методы тестирования.
8Для чего осуществляется тестирование тестового окружения?
9Перечислите виды требований и способы их тестирования.

Раздел 2 Структурное и функциональное тестирование ПО

Тема 2.1 Планирование процесса тестирования. Тест-план. Риски и стратегия тестирования

Тест-план.
Планирование процесса тестирования.
Риски и стратегия тестирования
Литература: [3]

Методические рекомендации

Планирование тестирования включает действия, направленные на определение основных целей тестирования и задач, выполнение которых необходимо для достижения этих целей.

В процессе планирования мы убеждаемся в том, что мы правильно поняли цели и пожелания заказчика и объективно оценили уровень риска для проекта, после чего ставим цели и задачи для, собственно, тестирования.

Для более ясного описания целей и задач тестирования составляются такие документы как тест-политика, тест-стратегия и тест-план.

Тест-политика – высокоуровневый документ, описывающий принципы, подходы и основные цели компании в сфере тестирования.

Тест-стратегия – высокоуровневый документ, содержащий описание уровней тестирования и подходов к тестированию в пределах этих уровней. Действует на уровне компании или программы (одного или больше проектов).

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.

Риск — это существующий или развивающийся фактор процесса, который обладает потенциально негативным воздействием на процесс.

Проще говоря, чтобы чётко разграничить риск и проблему: риск — это то, что может случиться и привести к негативным последствиям, а проблема, это то, что уже случилось и мешает работать. И риск, и проблема мешают или могут мешать работать, но способы работы с рисками и проблемами несколько разные: первые надо пытаться понять, найти и по возможности минимизировать их последствия до того, как они «выстрелят», а с проблемами надо работать по факту — чинить или «тушить». Если отделить риски и проблемы, область управления рисками становится намного проще и понятнее.

Простым примером, который не является риском связанным с тестированием ПО, но часто к таковым относится, является использование одного окружения для тестеров и девелоперов. Неудобная ситуация, которая порождает или может породить кучу проблем, но это источник проблемы, а не риск.

Тема 2.2 Тест-дизайн. Принципы разработки тестов

Тест-дизайн.
Принципы разработки тестов
Литература: [9]

Методические рекомендации

При создании IT-продукта большую роль играет обеспечение качества – Quality Assurance (QA). Для того, чтобы устранить ошибки и «баги», QA-инженеры в числе прочих инструментов применяют техники тест-дизайна.

Тест-дизайн – это разработка, создание тестов. Каждый тест направлен на проверку конкретного предположения. Например, «что будет, если пользователь по ошибке кликнет здесь не левой, а правой кнопкой мыши».

QA моделирует набор тестовых случаев (тест-кейсов), чтобы проверить, как приложение ведет себя в разных условиях. Задача специалиста – найти баланс и выявить максимальное количество ошибок при необходимом минимуме тестовых сценариев. При этом нужно проверить все наиболее важные кейсы, поскольку время тестирования ограничено.

Основные техники тест дизайна:

- Верификация – это процесс оценки системы или её компонентов с целью определения удовлетворяют ли результаты текущего этапа разработки условиям, сформированным в начале этого этапа.
- Валидация – это определение соответствия, разрабатываемого ПО ожиданиям и потребностям пользователя, требованиям к системе. Верификация и валидация являются одними из техник тест дизайна.
- Positive\ negative testing. Заключается в тестировании позитивных и негативных сценариев. Порядок тестирования такой: сначала проверяем позитивные сценарии, и только потом негативные. Позитивный тестовый случай использует только корректные данные и проверяет, что программа работает так, как и полагается, при условии, что пользователь вносит корректные данные и не выходит за рамки предусмотренного сценария поведения.
- Эквивалентное разделение (Equivalence Partitioning - EP), классы эквивалентности (equivalent classes-EC), например, у вас есть диапазон допустимых значений от 1 до 10, вы должны выбрать одно верное значение внутри интервала, скажем, 5, и одно неверное значение вне интервала - 0.
- Анализ граничных Значений (Boundary Value Analysis – BVA) Если взять пример выше, в качестве значений для позитивного тестирования выберем минимальную и максимальную границы (1 и 10), и значения больше и меньше границ (0 и 11). Анализ Граничный значений может быть применен к полям, записям, файлам и другим параметрам, имеющим ограничения. Чтобы удостовериться в правильности поведения программы при различных входных данных, в идеале следует протестировать все возможные значения для каждого элемента этих данных.
- Причина/ Следствие (Cause/Effect - CE). Это ввод комбинаций условий (причин), для получения ответа от системы (следствие). Например, вы проверяете возможность добавлять клиента, используя определенную экранную форму. Для этого вам необходимо будет ввести несколько полей, таких как "Имя",

"Адрес", "Номер Телефона" а затем, нажать кнопку "Добавить" - эта "Причина". После нажатия кнопки "Добавить", система добавляет клиента в базу данных и показывает его номер на экране - это "Следствие".

- Предугадывание ошибки (Error Guessing - EG) Это когда тест аналитик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы "предугадать" при каких входных условиях система может выдать ошибку. Например, спецификация говорит: "пользователь должен ввести код". Тест аналитик, будет думать: "Что, если я не введу код?", "Что, если я введу неправильный код? ", и так далее. Это и есть предугадывание ошибки.
- Исчерпывающее тестирование (Exhaustive Testing - ET). Используется крайне редких случаях. В пределах этой техники вы должны проверить все возможные комбинации входных значений, и в результате, это должно найти все проблемы. На практике применение этого метода не представляется возможным, из-за огромного количества входных значений.
- Попарное тестирование (Pairwise testing). Метод классов эквивалентности применяется для тестирования каждого входного параметра по отдельности. Пусть наша программа принимает на вход десяток параметров. Дефекты, возникающие при определенном сочетании всех десяти параметров, довольно редки. Взаимное влияние параметров, о котором пользователь не знает - это дефект интерфейса (интерфейс интуитивно не понятен). Чаще всего будут встречаться ситуации, в которых один параметр влияет на один из оставшихся, т.е. самыми частыми будут дефекты, возникающие при определенном сочетании двух каких-то параметров. Таким образом, можно упростить себе задачу и протестировать все возможные значения для каждой из пар параметров. Такой подход называется попарным тестированием (pairwise testing).
- ADHOC testing. Это тестирование без подробных спецификаций, сопроводительных документов, тест плана и т.д. Другими словами, тестирование в полном хаосе. Преимущество такой техники в том, что нет необходимости в планировании и документации, наиболее важные ошибки находятся быстро, нет задержек со стартом проекта.
- Дымовое (Smoke testing). Понятие дымовое тестирование пошло из инженерной среды: "При вводе в эксплуатацию нового оборудования ("железа") считалось, что тестирование прошло удачно, если из установки не пошел дым." В области же программного обеспечения, дымовое тестирование рассматривается как короткий цикл тестов, выполняемый для подтверждения того, что после сборки кода (нового или исправленного) устанавливаемое приложение, стартует и выполняет основные функции.

Тема 2.3 Разработка и документирование тестов

Разработка и документирование тестов.
Тестовая документация
Литература: [3]

Методические рекомендации

Рабочая тестовая документация значительно улучшает качество последующего тестирования за счет анализа и детального планирования тестов. После завершения тестирования наличие тестовой документации позволяет оценить, насколько успешно были проведены все этапы тестирования, а для заказчика является подтверждением реального объема работ.

Рабочую тестовую документацию тестировщик может разрабатывать исключительно на основе спецификации еще до поставки программного обеспечения. В этом случае после поставки на тестирование версии программного продукта специалист по тестированию может сразу приступить к поиску дефектов.

Существуют следующие виды рабочей тестовой документации (рисунок 5):

1. Check List.
2. Acceptance Sheet.
3. Test Survey.
4. Test Cases.

Основные факторы выбора тестовой документации – сложность бизнес логики проекта, сроки проекта, размер команды и объем проекта.

На одном проекте могут комбинироваться несколько типов тестовой документации. Например, для всего проекта составлен Acceptance Sheet, но для наиболее сложных частей составлены Test Cases. Если какие-либо модули программного продукта будут подвергаться автоматизированному тестированию, то для таких модулей в обязательном порядке составляются Test Cases.

Тип документации	Что описывают	Когда используют
Checklist	Вспомогательный тип документации, содержащий список основных проверок.	Для типовой функциональности.
Acceptance Sheet	Перечень всех модулей и функций приложения, подлежащих проверке.	Небольшие (до 3 месяцев), простые по бизнес-логике проекты.
Test Survey	Перечень всех модулей и функций, а также конкретные проверки для них. Может содержать ожидаемый результат.	Средние или большие проекты с понятной бизнес-логикой.
Test Cases	Набор входных значений, предусловий, пошаговое описание и постусловия для каждой проверки. Всегда содержит ожидаемый результат.	Большие и долгосрочные проекты, проекты со сложной бизнес-логикой, проекты с большой командой.

Рисунок 4 – Виды рабочей тестовой документации

Примеры фрагментов рабочей тестовой документации приведены на рисунке 5.

При составлении рабочей тестовой документации необходимо указать номер тестируемой сборки, тип выполняемой тестовой активности, период времени тестирования, ФИО тестировщика, тестовое окружение (операционная система, браузер, др.).

Рабочая тестовая документация представляет собой перечень всех проверок для модулей/подмодулей приложения. В качестве одного модуля как правило выступает рабочее окно приложения, в качестве подмодулей – логически завершенные блоки этого окна.

Для каждого модуля в обязательном порядке выполняется тестирование GUI, а также общие функциональные проверки (General). Далее в рамках модуля в качестве функциональных проверок выступают действия над активными элементами пользовательского интерфейса (полями, кнопками, чекбоксами и т.д.). Степень детализации каждой из таких функциональных проверок зависит от выбранного типа тестовой документации (Acceptance Sheet, Test Survey, Test Cases). В частности, для Acceptance Sheet все активные элементы пользовательского интерфейса только перечисляют. Для Test Survey для каждого элемента приводят позитивные и негативные проверки, источником которых являются базовые проверки (в виде чеклиста) для соответствующих элементов GUI. Для Test Cases каждую из позитивных и негативных проверок описывают в виде последовательности шагов с указанием ожидаемого результата.

Для Test Survey, Test Cases напротив каждой проверки указывается глубина тестирования: Smoke, MAT, AT. Для Acceptance Sheet в качестве глубины тестирования всегда указывается AT.

Checklist	Acceptance Sheet	Test Survey	Test Cases
Протестировать форму авторизации	Форма авторизации: 1. GUI. 2. General. 3. Поле «Эл.адрес или телефон». 4. Поле «Пароль». 5. Кнопка «Войти». 6. Чекбокс «Не выходить из системы». 7. Ссылка «Забыли пароль».	Форма авторизации: 1. GUI. 2. General. 3. Валидный эл.адрес + валидный пароль. 4. Валидный телефон + валидный пароль. 5. Валидный эл.адрес + невалидный пароль. 6. Валидный телефон + невалидный пароль. 7. Невалидный эл.адрес или телефон + валидный пароль. 8. Невалидный эл.адрес или телефон + невалидный пароль. 9. Запомнить данные: выйти из системы и зайти обратно. 10. Ссылка «Забыли пароль».	Авторизация с помощью e-mail: 1. Открыть страницу abc.com. 2. Ввести в поле «Эл.адрес или телефон» e-mail abc@mail.ru. 3. Ввести в поле «Пароль» пароль qwerty. 4. Нажать на кнопку «Войти». Ожидаемый результат: пользователь переходит на свою домашнюю страницу.

Рисунок 5 – Примеры тестовой документации

Тема 2.4 Описание дефектов. Жизненный цикл дефектов

Дефект. Описание дефектов. Жизненный цикл дефектов
Литература: [2]

Методические рекомендации

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 того, 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 6 представлены уровни отслеживания дефектов.

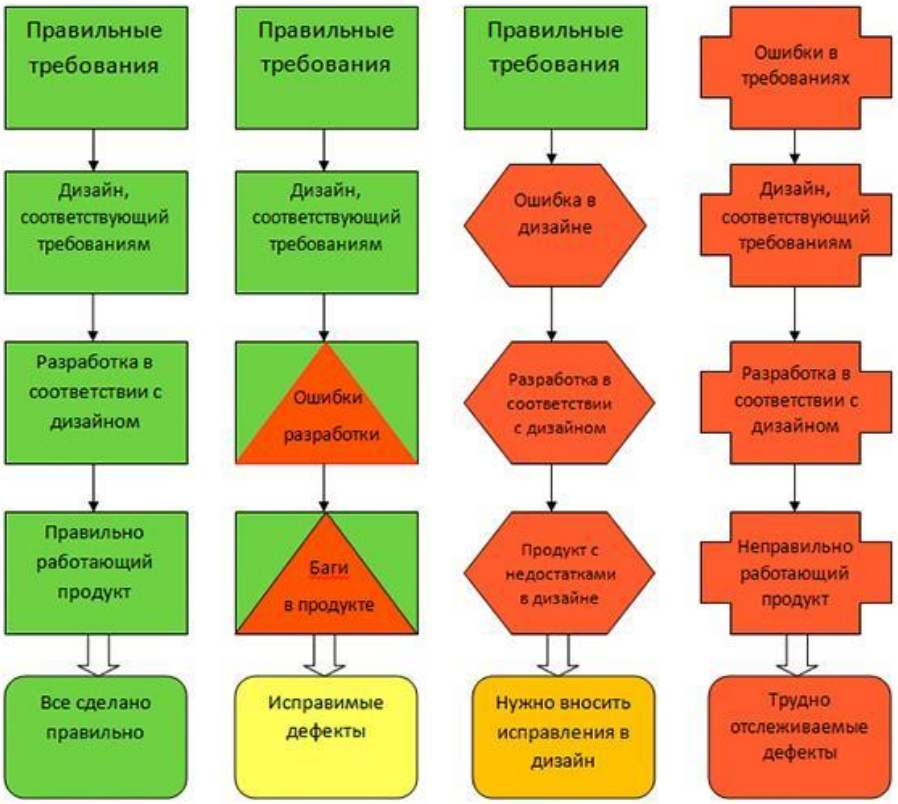

Рисунок 6 – Уровни отслеживания дефектов

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 2.5 Документирование результатов тестирования

Документация, создаваемая на различных этапах жизненного цикла.
Документация, сопровождающая процессы верификации и тестирования.
Отчеты о прохождении тестов.
Отчеты о покрытии программного кода.
Отчеты о проблемах.
Литература: [9]

Методические рекомендации

Итоговый отчет можно разделить на части с соответствующей информацией:

- Приветствие.
- Общая информация (Common Information).
- Тестовое окружение (Test Platform).
- Рекомендации QA (QA Recommendations).
- Детализированная информация (Detailed Information).
- Окончание содержимого.

Приветствие. Свое письмо с отчетом необходимо начать с приветствия всех адресатов. Если по каким-либо причинам произошла задержка данных отчета, либо не весь запланированный функционал был проверен, то эту информацию необходимо предоставить в начале письма. Следует извиниться за задержку и указать адекватные причины произошедшего. Также в самом начале письма следует указывать, если были какие-то внешние факторы, препятствующие проверке какой-то части функционала. Если во время тестирования не произошло никаких форс-мажорных обстоятельств, то достаточно обычного вежливого приветствия и далее уже переход к следующим пунктам.

Общая информация (Common Information). В данной части отчета описывается, какие виды тестов проводились. Зачастую указываются модули, которые тестировались или функционал. Стоит удостовериться, не забыта ли какая-то часть функционала, особенно это актуально, когда нужно собрать итоговый отчет, соединив в себе данные о разных видах тестов и функционале.

Тестовое окружение (Test Platform). Как правило, в этой части указываются:

- Название проекта;
- Номер сборки;
- Ссылка на проект (сборку). Необходимо убедиться, что зайдя по этой ссылке вы действительно попадаете на проект или можете установить приложение.

При указании данных в этой части отчета нужно быть очень внимательным, т.к. неправильная ссылка на сборку или неверный номер сборки не дают достоверной информации всем заинтересованным людям, а также затрудняют работу человеку, собирающему финальный отчет.

Рекомендации QA (QA Recommendations). Данная часть отчета является наиболее важной, т.к. здесь отражается общее состояние сборки. Здесь

показывается аналитическая работа тестировщика, его рекомендации по улучшению функционала, наиболее слабые места и наиболее критичные дефекты, динамика изменения качества проекта. В этом разделе должна быть информация о следующем:

- Указан функционал (часть функционала), который заблокирован для проверки. Даны пояснения почему этот функционал не проверен (указаны наиболее критичные дефекты).
- Произведен анализ качества проверенного функционала. Следует указать, улучшилось оно или ухудшилось по сравнению с предыдущей версией, какое качество на сегодняшний момент, какие факторы повлияли на выставление именно такого качества сборки.
- Если качество сборки ухудшилось, то обязательно должны быть указаны регрессионные места.
- Наиболее нестабильные части функционала следует выделить и указать причину, по которой они таковыми являются.
- Даны рекомендации по тому функционалу и дефектам, скорейшее исправление которых является наиболее приоритетным.
- Список наиболее критичных для сборки дефектов, с указанием названия и их критичности.
- Для отчета уровня Smoke обязательно указать весь нестабильный функционал.

Если сборка является релизной или предрелизной, то любое ухудшение качества является критичным и важно об этом сообщить менеджеру как можно раньше. Помимо всего вышеуказанного для релизных и предрелизных сборок в отчете о качестве продукта важно указывать следующее:

Дана информация о всех проблемах, характерных сборке. Проведен анализ, насколько оставшиеся проблемы являются критичными для конечного пользователя.

Указаны дефекты, которые следует исправить, чтобы качество конечной сборки было выше.

Детализированная информация (Detailed Information). В данной части отчета описывается более подробная информация о проверенных частях функционала, устанавливается качество каждой проверенной части функционала(модуля) в отдельности. В зависимости от типа проводимых тестов, эта часть отчета будет отличаться.

Smoke

При оценке качества функционала на уровне Smoke теста, оно может быть либо Приемлемым, либо Неприемлемым. Качество сборки зависит от нескольких факторов:

- Если это релизная или предрелизная сборка, то для выставления Приемлемого качества на уровне Smoke не должно быть найдено функциональных дефектов.
- Наличие нового функционала. Новый функционал, который впервые поставляется на тестирование, не должен содержать дефектов уровня Smoke для выставления Приемлемого качества всей сборки.

− Чтобы установить сборке Приемлемое качество, не должно быть дефектов уровня Smoke у того функционала, по которому планируется проводить полные тесты.

− Все наиболее важные части функционала отрабатывают корректно, тогда качество всего функционала на уровне Smoke может быть оценено, как Примлемое.

В части о детализированной информации качества сборки следует более подробно описать проблемы, которые были найдены во время теста.

Окончание содержимого. В завершении содержимое отчета должно включать в себя информацию следующего характера:

− Ссылка на тест-план.
− Ссылка на документ feature matrix (если таковой имеется).
− Ссылка на документ со статистикой (если таковой имеется).
− Общее количество всех новых дефектов.
− Подпись высылающего отчет.

Данные ссылки должны быть корректными, необходимо проверить достоверную ли информацию получает пользователь, открывший ссылку. Следует обращать особое внимание на подпись, удостоверьтесь, что указана именно ваша подпись либо какая-то универсальная для определенного проекта подпись.

Вопросы для самоконтроля

1 Дайте определение понятию тест-плана, приведите порядок планирования процесса тестирования.

2 Дайте определение понятию тест-дизайна.

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

4 Для чего необходима тестовая документация и на какие виды она подразделяется?

5 Перечислите основные причины появления дефектов в программном коде.

6 На какие пункты можно разделить итоговый отчет?

Раздел 3 Организация тестирования ПО
Тема 3.1 Модульное тестирование

Цели и задачи модульного тестирования.
Подходы к проектированию тестового окружения.
Организация модульного 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 помощью 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.

Тема 3.2 Интеграционное тестирование и системное тестирование

Интеграционное тестирование. Цели и задачи интеграционного тестирования. Классификация методов интеграционного тестирования. Планирование и организация интеграционного тестирования.

Системное тестирование. Цели и задачи системного тестирования. Виды системного тестирования. Планирование и организация системного тестирования

Литература: [3], [4]

Методические рекомендации

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.

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).

Тема 3.3 Отладка ПО, её виды

Отладка ПО. Виды отладки. Методы и средства получения дополнительной информации.

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 7.

Рисунок 7 – 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 обнаруживаться 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 8 представлена классификация ошибок этапа выполнения по возможным причинам

Рисунок 8 – 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 9 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 9 – 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 10).

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 10 – 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 3.4 Тестирование пользовательского интерфейса

Цели, задачи и особенности тестирования пользовательского интерфейса.

Функциональное тестирование пользовательских интерфейсов. Полнота покрытия. Методы проведения тестирования пользовательского интерфейса. Тестирование удобства использования пользовательского интерфейса.

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 3.5 Тестирование объектно-ориентированных программных средств (ПС)

Тестирование объектно-ориентированных программных средств.

Проектирование тестовых вариантов. Тестирование содержания и тестирование взаимодействия классов.

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 характеристики 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 3.6 Регрессионное тестирование

Регрессионное тестирование.

Цели, задачи регрессионного тестирования. Виды регрессионного тестирования. Методы отбора тестов

Особенности проведения регрессионного тестирования

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 уменьшение 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 существующими 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 3.7 Особенности тестирования Web-приложений

Тестирование web-приложений. Функциональное тестирование web-приложений, его особенности.

Составление отчета

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 своей 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 3.8 Тестирование безопасности и производительности

Тестирование безопасности. Тестирование производительности
Литература: [2]

Методические рекомендации

Тестирование производительности – это комплекс типов тестирования, целью которого является определение работоспособности, стабильности, потребления ресурсов и других атрибутов качества приложения в условиях различных сценариев использования и нагрузок. Тестирование производительности позволяет находить возможные уязвимости и недостатки в системе с целью предотвратить их пагубное влияние на работу программы в условиях использования. Необходимые параметры работы системы в определенной среде можно тестировать с помощью:

- Определения рабочего количества пользователей приложения.

- Измерение времени выполнения различных операций системы.
- Определения производительности приложения при различных степенях нагрузки.
- Определения допустимых границ производительности программы при разных уровнях нагрузки.

В зависимости от характеристик, которые нам нужно протестировать, тестирование производительности делится на типы:

- Нагрузочное тестирование (Loadtesting) – тестирование времени отклика приложения на запросы различных типов, с целью удостовериться, что приложение работает в соответствии с требованиями при обычной пользовательской нагрузке.
- Стресс-тестирование (Stresstesting) – тестирование работоспособности приложения при нагрузках, превышающих пользовательские в несколько раз. При стресс-тестировании (зачастую, только при нем) мы можем получить реальные данные границ производительности приложения, исследовать способность программы обрабатывать исключения, ее стабильность и устойчивость. Именно в значительно увеличенной нагрузке на приложение и заключается разница между тестированием производительности и стресс тестированием.
- Тестирование стабильности или наработка на отказ (Stability/Reliabilitytesting) исследует работоспособность приложения при длительной работе во времени, при нормальной для программы нагрузке.
- Объемное тестирование (VolumeTesting) – тестирование проводится с увеличением не нагрузки и времени работы, а количества используемых данных, которые хранятся и используются в приложении.

Очень часто при определении тестирования производительности и его типах приходят к ошибочному пониманию и путанице данных терминов. Чтобы избежать этого и закрепить полученные знания, подведем итог. Итак, тестирование производительности – это проверка таких нефункциональных требований, как производительность и работоспособность приложения при различных нагрузках и условиях. В зависимости от исследуемой характеристики программы, мы можем выделить такие типы тестирования как:

- Нагрузочное тестирование (производительность при нормальных условиях).
- Стресс-тестирование (работоспособность, производительность и характеристики приложения при экстремальных нагрузках).
- Тестирование стабильности (при длительной работе).
- Объемное тестирование (при увеличенных объемах обрабатываемых данных).

Тема 3.9 Особенности тестирования мобильных приложений

Тестирование мобильных приложений. Особенности проведения
Литература: [2]

Методические рекомендации

Тестирование мобильных приложений – это процесс, с помощью которого прикладное ПО, разработанное для портативных мобильных устройств, проверяется на его функциональность, удобство использования и совместимость. Тестирование может быть мануальным или автоматизированным.

1 Функциональное тестирование является самым базовым тестом для любого приложения, для проверки соответствия требованиям. Подобно другим приложениям, основанным на пользовательском интерфейсе, мобильные приложения требуют ряда взаимодействий человека в пользовательских сценариях.

2 Тестирование совместимости имеет самый высокую важность, когда дело доходит до тестирования мобильных приложений. Цель теста на совместимость мобильного приложения, как правило, состоит в том, чтобы ключевые функции приложения работали должным образом на конкретном устройстве. Сама совместимость должна занимать всего несколько минут и может быть спланирована заранее. Решить, какие тесты на совместимость мобильных устройств следует выполнить не легкая задача (поскольку тестирование со всеми существующими устройствами просто невозможно). Поэтому необходимо подготовить тестовую матрицу с каждой возможной комбинацией и расставить приоритеты для клиента.

3 Localization Testing. В настоящее время большинство приложений предназначены для глобального использования, и очень важно заботиться о региональных особенностях, таких как языки, часовые пояса и т.д. Важно проверить функциональность приложения, когда кто-то меняет часовой пояс. Необходимо учитывать, что иногда западные дизайны могут не работать с аудиторией из восточных стран или наоборот.

4 Laboratory testing, обычно проводимые сетевыми операторами, выполняются путем моделирования всей беспроводной сети. Этот тест выполняется для обнаружения каких-либо сбоев, когда мобильное приложение использует передачу голоса и / или данных для выполнения некоторых функций.

5 Performance Testing охватывает производительность клиентских приложений, сервера и сети. Благодаря Performance Testing можно идентифицировать существующие сети, серверы и узкие места серверных приложений, учитывая предопределенную нагрузку и сочетание транзакций.

6 Stress Testing является обязательным тестированием на пути обнаружения исключений, зависаний и взаимоблокировок, что может остаться незамеченными во время тестирования функциональности и пользовательского интерфейса.

Вот список некоторых критериев:

− Загрузите в свое приложение как можно больше данных, чтобы попытаться достичь его предела.

− Выполняйте одни и те же операции снова и снова.

− Выполняйте повторные операции на разных скоростях, очень быстро или очень медленно.

− Оставьте ваше приложение работающим в течение длительного периода времени, одновременно взаимодействуя с устройством и просто оставляя его бездействующим, или выполняя некоторую автоматическую задачу, которая занимает много времени, например, слайд-шоу.

− Случайно отправлять экранные нажатия и нажатия клавиш в вашем приложении.

− На вашем устройстве должно быть запущено несколько приложений, чтобы вы могли часто переключаться между приложением и другими приложениями на устройстве.

7 Security Testing помогает выявить все возможные уязвимости в отношении политик взлома, аутентификации и авторизации, безопасности данных, управления сеансами и других стандартов безопасности. Приложения должны шифровать имя пользователя и пароли при аутентификации пользователя по сети.

Один из способов тестирования сценариев, связанных с безопасностью, заключается в маршрутизации данных вашего мобильного устройства через прокси-сервер, такой, как OWASP Zed Attack Proxy, и поиске уязвимости.

8 Usability Testing оценивает приложение на основе следующих трех критериев для целевой аудитории: эффективность; точность и полнота; удовлетворенность. Очень важно провести юзабилити-тестирование с самого раннего этапа разработки приложения. Этот вид тестирования требует активного участия пользователей, и результаты могут повлиять на дизайн приложения, что очень трудно изменить на более поздних этапах проекта.

Существует еще множество тестов, которые необходимо провести при тестировании мобильного приложения:

− Installation/Uninstallation testing
− Updates Testing
− Certification Testing
− Screen Orientation / Resolution
− Memory Leakage Testing
− Available Tools
− Touch Screens
− Soft & Hard Keys

Вопросы для самоконтроля

1 Опишите принцип модульного тестирования.

2 Опишите принципы интеграционного и системного тестирования.

3 Из каких фаз состоит функциональное тестирование пользовательского интерфейса?

4 Назовите особенности интеграционного тестирования объектно-ориентированных ПС.

5 Перечислите виды системного тестирования.

6 Опишите классификацию ошибок.

7 Перечислите тесты, которые необходимо провести при тестировании мобильных приложений.

8 Назовите особенности тестирования web-приложений.

Раздел 4 Тестирование при промышленной разработке ПО
Тема 4.1 Автоматизация тестирования. Введение в Selenium

Автоматизация тестирования. Цели и подходы автоматизации. Инструменты.

Введение в Selenium. Принцип работы Selenium

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.

Selenium Webdriver – инструмент для автоматизации реального браузера, как локально, так и удаленно, наиболее близко имитирующий действия пользователя.

Selenium 2 (или Webdriver) - последнее пополнение в пакете инструментов Selenium и является основным вектором развития проекта. Это абсолютно новый инструмент автоматизации. По сравнению с Selenium RC Webdriver использует совершенно иной способ взаимодействия с браузерами. Он напрямую вызывает команды браузера, используя родной для каждого конкретного браузера API. Как совершаются эти вызовы и какие функции они выполняют зависит от конкретного браузера. В то же время Selenium RC внедрял javascript код в браузер при запуске и использовал его для управления веб приложением. Таким образом, Webdriver

использует способ взаимодействия с браузером более близкий к действиям реального пользователя.

Самое главное изменение новой версии Selenium - это Webdriver API.

По сравнению с более старым интерфейсом он обладает рядом преимуществ:

− Интерфейс Webdriver был спроектирован более простым и выразительным;

− Webdriver обладает более компактным и объектно-ориентированным API;

− Webdriver управляет браузером более эффективно, а также справляется с некоторыми ограничениями, характерными для Selenium RC, как загрузка и отправление файлов, попапы и дилоги.

Для работы с Webdriver необходимо 3 основных программных компонента:

− Браузер, работу которого пользователь хочет автоматизировать. Это реальный браузер определенной версии, установленный на определенной ОС и имеющий свои настройки (по умолчанию или кастомные). На самом деле Webdriver может работать и с "ненастоящими" браузерами, но подробно о них позже.

− Для управления браузером совершенно необходим driver браузера. Driver на самом деле является веб сервером, который запускает браузер и отправляет ему команды, а также закрывает его. У каждого браузера свой driver. Связано это с тем, что у каждого браузера свои отличные команды управления и реализованы они по-своему. Найти список доступных драйверов и ссылки для скачивания можно на официальном сайте Selenium проекта.

− Скрипт/тест, который содержит набор команд на определенном языке программирования для драйвера браузера. Такие скрипты используют Selenium Webdriver bindings (готовые библиотеки), которые доступны ользователям на различных языках.

Тема 4.2 WebDriver и JUnit

Основные методы библиотеки Selenium webdriver и Selenium JUnit
Литература: [10]

Методические рекомендации

Основными понятиями в Selenium Webdriver являются:

− Webdriver — самая важная сущность, ответственная за управление браузером. Основной ход скрипта/теста строится именно вокруг экземпляра этой сущности.

− Webelement — вторая важная сущность, представляющая собой абстракцию над веб элементом (кнопки, ссылки, инпута и др.). Webelement инкапсулирует методы для взаимодействия пользователя с элементами и получения их текщего статуса.

− By — абстракция над локатором веб элемента. Этот класс инкапсулирует информацию о селекторе(например, CSS), а также сам локатор

элемента, то есть всю информацию, необходимую для нахождения нужного элемента на странице.

Сам процесс взаимодействия с браузером через Webdriver API довольно прост:

1 Нужно создать Webdriver:

```
IWebDriver driver = new ChromeDriver();
```

При выполнении этой команды будет запущен Chrome, при условии, что он установлен в директорию по умолчанию и путь к ChromeDriver сохранен в системной переменной PATH.

2 Необходимо открыть тестируемое приложение (AUT), перейдя по url:

```
driver.Navigate().GoToUrl("https://google.com");
```

Теоритически в хроме при этом должен открыться сайт компании.

3 Далее следует серия действий по нахождению элементов на странице и взаимодействию с ними:

```
By elementLocator = By.Id("#element_id");
IWebElement element = driver.FindElement(elementLocator));
```

Или более кратко:

```
IWebElement element = driver.FindElement(By.Id("#element_id")));
```

После нахождения элемента, кликнем по нему:

```
element.Click();
```

Далее следует совокупность похожих действий, как того требует сценарий.

4 В конце теста (часто также и в середине) должна быть какая-то проверка, которая и определит в конечном счете результат выполнения теста:

```
assertEquals("Webpage expected title", driver.getTitle());
```

Проверки может и не быть, если цель вашего скрипта — не тест, а выполнение какой-то рутины.

5 После теста надо закрыть браузер:

```
driver.Quit();
```

Следует отметить, что для поиска элементов доступно два метода:

1 Первый — найдет только первый элемент, удовлетворяющий локатору:

```
IWebElement element = driver.FindElement(By.Id("#element_id")));
```

2 Второй — вернет весь список элементов, удовлетворяющих запросу:

```
List<IWebElement> elements = driver.FindElements(By.Name("elements_name"))
```

JUnit — библиотека для модульного тестирования программ Java. Созданный Кентом Беком и Эриком Гаммой, JUnit принадлежит семье фреймворков xUnit для разных языков программирования, берущей начало в SUnit Кента Бека для Smalltalk. JUnit породил экосистему расширений — JMock, EasyMock, DbUnit, HttpUnit и т. д.

Библиотека JUnit была портирована на другие языки, включая PHP (PHPUnit), C# (NUnit), Python (PyUnit), Fortran (fUnit), Delphi (DUnit), Free Pascal (FPCUnit), Perl (Test::Unit), C++ (CPPUnit), Flex (FlexUnit), JavaScript (JSUnit).

JUnit – это Java фреймворк для тестирования, т. е. тестирования отдельных участков кода, например, методов или классов. Опыт, полученный при работе с JUnit, важен в разработке концепций тестирования программного обеспечения.

Пример теста JUnit:

```
import org.junit.Test;
import junit.framework.Assert;

public class MathTest {
   @Test
   public void testEquals() {
      Assert.assertEquals(4, 2 + 2);
      Assert.assertTrue(4 == 2 + 2);
   }

   @Test
   public void testNotEquals() {
      Assert.assertFalse(5 == 2 + 2);
   }
}
```

JUnit позволяет в любой момент быстро убедиться в работоспособности кода. Если программа не является совсем простой и включает множество классов и методов, то для её проверки может потребоваться значительное время. Естественно, что данный процесс лучше автоматизировать. Использование JUnit позволяет проверить код программы без значительных усилий и не занимает много времени.

Юнит тесты классов и функций являются своего рода документацией к тому, что ожидается в результате их выполнения. И не просто документацией, а документацией, которая может автоматически проверять код на соответствие предъявленным функциям. Это удобно, и часто тесты разрабатывают как вместе, так и до реализации классов. Разработка через тестирование — крайне популярная технология создания серьезного программного обеспечения.

Классы JUnit являются важными классами, используемыми при написании и тестировании JUnits. Некоторые из важных классов:

- Assert — Содержит набор методов assert.
- TestCase — содержит тестовый набор, который определяет устройство для запуска нескольких тестов.
- TestResult — содержит методы для сбора результатов выполнения контрольного примера.

Вопросы для самоконтроля

1 Перечислите особенности организации автоматизированного тестирования.
2 Для чего необходим Selenium WebDriver
3 Перечислите основные методы WebDriver
4 Перечислите основные классы JUnit

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

9 https://www.simbirsoft.com/blog/tekhniki-test-dizayna-i-ikh-prednaznachenie/

10 https://artstroy.net/selenium-webdriver-vvedenie/

Задания на домашнюю контрольную работу по учебной дисциплине «Тестирование 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 Тестирование мобильных приложений. Особенности проведения.
36 Основные методы библиотеки Selenium webdriver и Selenium JUnit.

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

Предпоследняя 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