Виды и методы тестирования (в том числе автоматизированные)

Тестирование информационных систем

Преподаватель: Сидиков И.Д.

Виды тестирования

Все виды тестирования программного обеспечения, в зависимости от преследуемых целей, можно условно разделить на следующие группы:

1. Функциональные
2. Нефункциональные
3. Связанные с изменениями

Функциональное тестирование

Функциональное тестирование - это тип тестирования программного обеспечения, которое проверяет программную систему на соответствие функциональным требованиям/спецификациям. Целью функциональных тестов является тестирование каждой функции программного приложения путем предоставления соответствующих входных данных и проверки выходных данных на соответствие функциональным требованиям.

Функциональное тестирование в основном включает в себя тестирование черного ящика и не касается исходного кода приложения. Это тестирование проверяет пользовательский интерфейс, API, базу данных, безопасность, связь между клиентом и сервером и другие функции тестируемого приложения. Тестирование может проводиться как вручную, так и с помощью автоматизации.

Что вы тестируете в функциональном тестировании?

Основной целью функционального тестирования является проверка функциональных возможностей программной системы. Он в основном концентрируется на:

1. Основные функции: тестирование основных функций приложения.
2. Базовое удобство использования: включает в себя базовое тестирование удобства использования системы. Он проверяет, может ли пользователь свободно перемещаться по экранам без каких-либо затруднений.
3. Доступность: Проверяет доступность системы для пользователя.
4. Условия ошибки: использование методов тестирования для проверки условий ошибки. Он проверяет, отображаются ли подходящие сообщения об ошибках.

Как проводить функциональное тестирование

Ниже приведен пошаговый процесс выполнения функционального тестирования:

1. Понимание функциональных требований
2. Идентификация тестовых входных данных или тестовых данных на основе требований
3. Вычислите ожидаемые результаты с выбранными тестовыми входными значениями
4. Выполнение тестовых случаев
5. Сравните фактические и вычисленные ожидаемые результаты

Функциональное и нефункциональное тестирование:

Функциональное тестирование	**Нефункциональное тестирование**
Функциональное тестирование выполняется с использованием функциональной спецификации, предоставленной клиентом, и проверяет систему на соответствие функциональным требованиям.	Нефункциональное тестирование проверяет производительность, надежность, масштабируемость и другие нефункциональные аспекты программной системы.
Функциональное тестирование выполняется в первую очередь	Нефункциональное тестирование следует проводить после функционального тестирования.
Для функционального тестирования можно использовать инструменты ручного тестирования или автоматизации.	Использование инструментов будет эффективным для этого тестирования
Бизнес-требования — это исходные данные для функционального тестирования.	Параметры производительности, такие как скорость и масштабируемость, являются входными данными для нефункционального тестирования.
Функциональное тестирование описывает, что делает продукт	Нефункциональное тестирование описывает, насколько хорошо работает продукт.
Легко проводить ручное тестирование	Трудно проводить ручное тестирование
Примеры функционального тестирования: •Модульное тестирование •Дымное тестирование •Санитарное тестирование •Интеграционное тестирование •Тестирование белого ящика •Тестирование черного ящика •Приемочное тестирование пользователей •Регрессионное тестирование	Примеры нефункционального тестирования: •Тестирование производительности •Нагрузочное тестирование •Объемное тестирование •Стресс тестирование •Тестирование безопасности •Тестирование установки •Проверка на проницаемость •Тестирование совместимости •Миграционное тестирование

Модульное тестирование (Unit Testing)

Модульное тестирование — это тип тестирования программного обеспечения, при котором тестируются отдельные модули или компоненты программного обеспечения. Цель состоит в том, чтобы подтвердить, что каждая единица программного кода работает так, как ожидалось. Модульное тестирование выполняется разработчиками на этапе разработки (этапа кодирования) приложения. Модульные тесты изолируют участок кода и проверяют его правильность. Единицей может быть отдельная функция, метод, процедура, модуль или объект.

Модульное тестирование — это метод тестирования WhiteBox, который обычно выполняется разработчиком. Хотя на практике из-за нехватки времени или нежелания разработчиков проводить тесты инженеры по контролю качества также проводят модульное тестирование.

Зачем выполнять модульное тестирование?

Модульное тестирование важно, потому что разработчики программного обеспечения иногда пытаются сэкономить время, выполняя минимальное модульное тестирование, и это миф, потому что несоответствующее модульное тестирование приводит к высокой стоимости исправления дефектов во время системного тестирования , интеграционного тестирования и даже бета-тестирования после создания приложения. Если надлежащее модульное тестирование проводится на ранней стадии разработки, то в конечном итоге это экономит время и деньги.

Вот основные причины для выполнения модульного тестирования в разработке программного обеспечения:

1. Модульные тесты помогают исправлять ошибки на ранних этапах цикла разработки и экономить средства.
2. Это помогает разработчикам понять кодовую базу тестирования и позволяет им быстро вносить изменения.
3. Хорошие модульные тесты служат проектной документацией
4. Модульные тесты помогают повторно использовать код. Перенесите код и тесты в новый проект. Изменяйте код, пока тесты не запустятся снова.

Как выполнить модульное тестирование

Чтобы выполнить модульные тесты, разработчики пишут раздел кода для тестирования определенной функции в программном приложении. Разработчики также могут изолировать эту функцию для более тщательного тестирования, что выявляет ненужные зависимости между тестируемой функцией и другими модулями, чтобы эти зависимости можно было устранить. Разработчики обычно используют платформу UnitTest для разработки автоматизированных тестов для модульного тестирования.

Модульное тестирование бывает двух типов

- Руководство
- Автоматизированный

Модульное тестирование обычно автоматизировано, но все же может выполняться вручную. Программная инженерия не отдает предпочтение одному перед другим, но автоматизация предпочтительнее. При ручном подходе к модульному тестированию может использоваться пошаговый учебный документ.

Как выполнить модульное тестирование

При автоматизированном подходе:

1. Разработчик пишет часть кода в приложении только для проверки функции. Позже они закомментируют и, наконец, удалят тестовый код при развертывании приложения.
2. Разработчик также может изолировать функцию, чтобы более тщательно протестировать ее. Это более тщательная практика модульного тестирования, которая включает копирование и вставку кода в собственную среду тестирования, а не в естественную среду. Изоляция кода помогает выявить ненужные зависимости между тестируемым кодом и другими единицами или пространствами данных в продукте. Затем эти зависимости могут быть устранены.
3. Кодировщик обычно использует UnitTest Framework для разработки автоматизированных тестовых случаев. Используя структуру автоматизации, разработчик вводит критерии в тест, чтобы проверить правильность кода. Во время выполнения тестовых случаев платформа регистрирует неудачные тестовые случаи. Многие фреймворки также автоматически помечают и сообщают об этих неудачных тестах . В зависимости от серьезности сбоя платформа может остановить последующее тестирование.
4. Рабочий процесс модульного тестирования: 1) Создание тестовых случаев 2) Обзор/переработка 3) Базовый план 4) Выполнение тестовых случаев.

Методы модульного тестирования

Методы модульного тестирования в основном подразделяются на три части: тестирование черного ящика, которое включает тестирование пользовательского интерфейса вместе с вводом и выводом, тестирование белого ящика, которое включает тестирование функционального поведения программного приложения, и тестирование серого ящика, которое используется для выполнения теста. наборы, методы тестирования, тестовые примеры и выполнение анализа рисков.

Методы покрытия кода, используемые в модульном тестировании, перечислены ниже:

1. Покрытие операторов - это метод тестирования белого ящика, при котором все исполняемые операторы в исходном коде выполняются хотя бы один раз.
2. Покрытие решений — это метод тестирования белого ящика, который сообщает об истинных или ложных результатах каждого логического выражения исходного кода.
3. Покрытие ветвей — это метод тестирования белого ящика, в котором проверяется каждый результат модуля кода (оператора или цикла).
4. Покрытие условий или покрытие выражений — это метод тестирования, используемый для проверки и оценки переменных или подвыражений в условном операторе.
5. Покрытие конечного автомата, безусловно, является наиболее сложным типом метода покрытия кода. Это потому, что он работает на поведении дизайна.

Преимущество модульного тестирования

1. Разработчики, желающие узнать, какие функции предоставляет модуль и как его использовать, могут просмотреть модульные тесты, чтобы получить общее представление об API модуля.
2. Модульное тестирование позволяет программисту позже провести рефакторинг кода и убедиться, что модуль по-прежнему работает правильно (например, регрессионное тестирование). Процедура заключается в написании тестовых примеров для всех функций и методов, чтобы всякий раз, когда изменение вызывает ошибку, ее можно было быстро идентифицировать и исправить.
3. Благодаря модульному характеру модульного тестирования мы можем тестировать части проекта, не дожидаясь завершения других.

Недостатки модульного тестирования

1. Нельзя ожидать, что модульное тестирование выявит каждую ошибку в программе. Невозможно оценить все пути выполнения даже в самых тривиальных программах.
2. Модульное тестирование по своей природе фокусируется на единице кода. Следовательно, он не может обнаруживать ошибки интеграции или общие ошибки системного уровня.

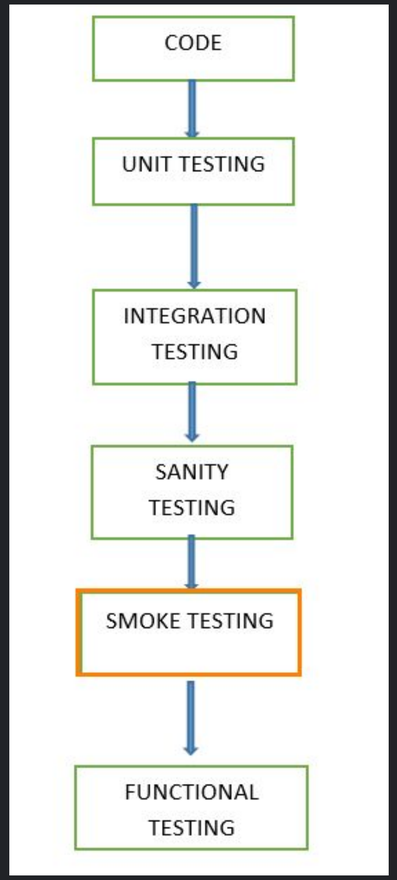
Модульное тестирование рекомендуется использовать в сочетании с другими действиями по тестированию.

Что такое дымовое тестирование

Smoke Testing — это процесс тестирования программного обеспечения, который определяет, является ли развернутая сборка программного обеспечения стабильной или нет. Дымовое тестирование является подтверждением того, что команда QA может приступить к дальнейшему тестированию программного обеспечения. Он состоит из минимального набора тестов, запускаемых в каждой сборке для проверки функциональных возможностей программного обеспечения. Дымовое тестирование также известно как «Тестирование проверки сборки» или «Тестирование доверия».

Когда мы проводим дымовые испытания

Дымовое тестирование выполняется всякий раз, когда разрабатываются новые функции программного обеспечения и интегрируются с существующей сборкой, которая развертывается в среде обеспечения качества/тестирования. Это гарантирует, что все критические функции работают правильно или нет.



Когда мы проводим дымовые испытания

В этом методе тестирования команда разработчиков развертывает сборку в QA. Берется подмножество тестовых случаев, а затем тестировщики запускают тестовые случаи в сборке. Команда контроля качества тестирует приложение на соответствие критическим функциям. Эти серии тестовых случаев предназначены для выявления ошибок в build. Если эти тесты пройдены, группа контроля качества продолжает функциональное тестирование .

Любой сбой указывает на необходимость вернуть систему команде разработчиков. Всякий раз, когда в сборку вносятся изменения, мы проводим тестирование с дымом, чтобы обеспечить стабильность.

Кто будет проводить дымовые испытания

После выпуска сборки в среду QA инженеры по обеспечению качества/руководители отдела контроля качества проводят тестирование с дымом. Всякий раз, когда появляется новая сборка, команда QA определяет основные функции приложения для проведения дымового тестирования. Команда контроля качества проверяет наличие отличий в тестируемом приложении.

Тестирование, выполняемое в среде разработки кода для проверки правильности приложения перед выпуском сборки в отдел контроля качества, называется тестированием работоспособности. Обычно это узкое и глубокое тестирование. Это процесс, который проверяет, соответствует ли разрабатываемое приложение его основным функциональным требованиям.

Почему мы проводим дымовые испытания?

Дымовое тестирование играет важную роль в разработке программного обеспечения, поскольку оно обеспечивает правильность системы на начальных этапах. Таким образом, мы можем сэкономить усилия по тестированию. В результате дымовые тесты доводят систему до исправного состояния. Как только мы завершим дымовое тестирование, мы приступим только к функциональному тестированию.

- Все ограничители шоу в сборке будут идентифицированы путем проведения дымового тестирования.
- Дымовое тестирование выполняется после того, как сборка передается в QA. С помощью дымового тестирования большинство дефектов выявляется на начальных этапах разработки программного обеспечения.
- С помощью дымового тестирования мы упрощаем обнаружение и исправление крупных дефектов.
- С помощью дымового тестирования команда QA может найти дефекты в функциональности приложения, которые могли появиться в новом коде.
- Дымовое тестирование выявляет основные серьезные дефекты.

Пример 1: Окно регистрации: можно перейти к следующему окну с действительным именем пользователя и паролем при нажатии кнопки отправки.

Преимущества дымового тестирования

Вот несколько преимуществ, перечисленных для Smoke Testing.

- Простота проведения тестирования
- Дефекты выявляются на ранних стадиях.
- Улучшает качество системы
- Снижает риск
- Доступ к прогрессу проще.
- Экономит усилия и время тестирования
- Легкое обнаружение критических ошибок и исправление ошибок.
- Он работает быстро
- Сводит к минимуму интеграционные риски

Пример примеров тестов дыма

ID	СЦЕНАРИИ ИСПЫТАНИЙ	ОПИСАНИЕ	ЭТАП ТЕСТА	ОЖИДАЕМЫЙ РЕЗУЛЬТАТ	ФАКТИЧЕСКИЙ РЕЗУЛЬТАТ	СТАТУС
1	Действительные учетные данные для входа	Проверьте функцию входа в веб-приложение, чтобы убедиться, что зарегистрированному пользователю разрешен вход с именем пользователя и паролем.	1.Запустите приложение 2.Перейдите на страницу входа 3.Введите правильное имя пользователя 4.Введите правильный пароль 5.Нажмите кнопку входа	Логин должен быть успешным	как и ожидалось	PASS
2	Добавление функциональности элемента	Возможность добавить товар в корзину	1.Выберите список категорий 2.Добавьте товар в корзину	Товар должен быть добавлен в корзину	Товар не добавляется в корзину	FAIL
3	Функциональность выхода	Проверьте функцию выхода	1. выберите кнопку выхода	Пользователь должен иметь возможность выйти.	Пользователь не может выйти	FAIL

Вывод

В программной инженерии дымовое тестирование должно выполняться в обязательном порядке для каждой сборки, поскольку оно помогает находить дефекты на ранних стадиях. Дымовое тестирование — это последний шаг перед тем, как сборка программного обеспечения перейдет в системную стадию. Смоук-тесты должны выполняться на каждой сборке, переданной на тестирование. Это относится к новым разработкам, а также к основным и второстепенным выпускам системы.

Перед выполнением дымового тестирования группа контроля качества должна убедиться в правильной версии сборки тестируемого приложения. Это простой процесс, который требует минимального времени для проверки стабильности приложения.

Дымовые тесты могут свести к минимуму усилия по тестированию и повысить качество приложения. Дымовое тестирование может проводиться вручную или с помощью автоматизации в зависимости от клиента и организации.

Санитарное тестирование

это вид тестирования программного обеспечения, выполняемого после получения сборки программного обеспечения с небольшими изменениями в коде или функциональности, чтобы убедиться, что ошибки были исправлены и что из-за этих изменений не возникло дополнительных проблем. Цель состоит в том, чтобы определить, работает ли предлагаемая функциональность примерно так, как ожидалось. Если тест работоспособности не проходит, сборка отклоняется, чтобы сэкономить время и деньги, связанные с более тщательным тестированием.

Цель состоит не в том, чтобы тщательно проверить новую функциональность, а в том, чтобы определить, что разработчик применил некоторую рациональность (здравомыслие) при создании программного обеспечения.

Разница между дымовым тестированием и санитарным тестированием

Дымное тестирование	Санитарное тестирование
Дымовое тестирование выполняется, чтобы убедиться, что критически важные функции программы работают нормально.	Санитарное тестирование проводится для проверки того, что новые функции/ошибки были исправлены.
Целью этого тестирования является проверка «стабильности» системы, чтобы приступить к более тщательному тестированию.	Цель тестирования — проверить «рациональность» системы, чтобы приступить к более строгому тестированию.
Это тестирование выполняется разработчиками или тестировщиками.	Проверка работоспособности при тестировании программного обеспечения обычно выполняется тестировщиками.
Тестирование дыма обычно документируется или записывается в сценарии.	Проверка работоспособности обычно не документируется и не записывается по сценарию.
Дымовое тестирование является частью приемочного тестирования.	Санитарное тестирование является подмножеством регрессионного тестирования .
Дымовое тестирование проверяет всю систему от начала до конца	Санитарное тестирование проверяет только определенный компонент всей системы.
Проверка дыма похожа на общую проверку здоровья	Санитарное тестирование похоже на специализированную проверку здоровья

Интеграционное тестирование

Системное интеграционное тестирование определяется как тип тестирования программного обеспечения, проводимого в интегрированной аппаратно-программной среде для проверки поведения всей системы. Это тестирование, проводимое на полной интегрированной системе для оценки соответствия системы заданным требованиям.

Системное интеграционное тестирование (SIT) выполняется для проверки взаимодействия между модулями программной системы. Он касается проверки требований к программному обеспечению высокого и низкого уровня, указанных в Спецификации/данных требований к программному обеспечению и Документе по проектированию программного обеспечения.

Он также проверяет сосуществование программной системы с другими и тестирует интерфейс между модулями программного приложения. В этом типе тестирования модули сначала тестируются по отдельности, а затем объединяются в систему.

Например, программные и/или аппаратные компоненты объединяются и тестируются постепенно, пока не будет интегрирована вся система.

Зачем проводить системное интеграционное тестирование

В программной инженерии тестирование системной интеграции проводится потому, что:

- Это помогает обнаружить дефект на ранней стадии
- Более ранняя обратная связь о приемлемости отдельного модуля будет доступна
- Планирование исправления дефектов является гибким, и его можно совмещать с разработкой.
- Правильный поток данных
- Правильный поток управления
- Правильное время
- Правильное использование памяти
- Исправьте требования к программному обеспечению

Как проводить системное интеграционное тестирование

Это систематический метод построения структуры программы при проведении тестов для выявления ошибок, связанных с интерфейсом.

Все модули интегрируются заранее, и вся программа тестируется как единое целое. Но во время этого процесса, скорее всего, возникнет множество ошибок.

Исправление таких ошибок затруднено, поскольку изоляция причин усложняется огромным расширением всей программы. Как только эти ошибки будут исправлены и исправлены, появится новая, и процесс продолжится в бесконечном цикле . Чтобы избежать этой ситуации, используется другой подход — добавочная интеграция.

Есть несколько дополнительных методов, таких как интеграционные тесты, которые проводятся в системе на основе целевого процессора. Используемая методология — тестирование черного ящика . Можно использовать как восходящую, так и нисходящую интеграцию.

Тестовые случаи определяются с использованием только высокоуровневых требований к программному обеспечению.

Как проводить системное интеграционное тестирование

Подтверждающие тесты на этом уровне выявляют проблемы, характерные для среды, такие как ошибки в выделении и освобождении памяти. Практичность интеграции программного обеспечения в хост-среду будет зависеть от того, сколько там целевого специфического функционала. Для некоторых встроенных систем связь с целевой средой будет очень сильной, что делает непрактичной интеграцию программного обеспечения в хост-среду.

Крупные разработки программного обеспечения разделят интеграцию программного обеспечения на несколько уровней. Более низкие уровни интеграции программного обеспечения могут быть основаны преимущественно на хост-среде, а более поздние уровни интеграции программного обеспечения становятся все более зависимыми от целевой среды.

Критерии входа и выхода для интеграционного тестирования

Обычно при выполнении интеграционного тестирования используется стратегия ETVX (критерии входа, задачи, проверки и критерии выхода).

Критерии входа:

- Завершение модульного тестирования (Юнит-тесты)

Входы:

- Данные о требованиях к программному обеспечению
- Документ по дизайну программного обеспечения
- План проверки программного обеспечения
- Документы по интеграции программного обеспечения

Критерии входа и выхода для интеграционного тестирования

Мероприятия:

- На основе требований высокого и низкого уровня создайте тестовые примеры и процедуры.
- Объединяйте низкоуровневые сборки модулей, реализующие общий функционал
- Разработайте тестовую обвязку
- Протестируйте сборку
- После прохождения теста сборка объединяется с другими сборками и тестируется до тех пор, пока система не будет интегрирована в целом.
- Повторно выполнить все тесты на платформе с целевым процессором и получить результаты.

Критерии выхода:

- Успешное завершение интеграции Программного модуля на целевом Аппаратном обеспечении
- Корректная работа программного обеспечения в соответствии с указанными требованиями

Выходы

- Отчеты об интеграционных испытаниях
- Примеры и процедуры тестирования программного обеспечения [SVCP].

Тестирование черного ящика

Тестирование « черного ящика» — это метод тестирования программного обеспечения, при котором функциональные возможности программных приложений проверяются без знания внутренней структуры кода, деталей реализации и внутренних путей. Тестирование черного ящика в основном фокусируется на вводе и выводе программных приложений и полностью основано на требованиях и спецификациях программного обеспечения. Он также известен как поведенческое тестирование.

Вышеупомянутый черный ящик может быть любой программной системой, которую вы хотите протестировать. Например, операционная система, такая как Windows, веб-сайт, такой как Google, база данных, такая как Oracle, или даже ваше собственное приложение. В разделе «Тестирование черного ящика» вы можете протестировать эти приложения, просто сосредоточившись на входных и выходных данных, не зная их внутренней реализации кода.

Как провести тестирование BlackBox

Вот общие шаги, необходимые для проведения любого типа тестирования черного ящика.

- Первоначально изучаются требования и спецификации системы.
- Тестер выбирает допустимые входные данные (положительный тестовый сценарий), чтобы проверить, правильно ли их обрабатывает SUT. Кроме того, выбираются некоторые недопустимые входные данные (сценарий отрицательного теста), чтобы убедиться, что SUT способна их обнаружить.
- Тестер определяет ожидаемые результаты для всех этих входов.
- Тестировщик программного обеспечения создает тестовые примеры с выбранными входными данными.
- Тестовые случаи выполняются.
- Тестировщик программного обеспечения сравнивает фактические результаты с ожидаемыми.
- Дефекты, если таковые имеются, устраняются и повторно тестируются.

Типы тестирования черного ящика

Существует много типов тестирования черного ящика, но наиболее известными из них являются следующие:

- Функциональное тестирование - Этот тип тестирования черного ящика связан с функциональными требованиями системы; это делают тестировщики программного обеспечения.
- Нефункциональное тестирование - Этот тип тестирования «черного ящика» связан не с тестированием конкретной функциональности, а с нефункциональными требованиями, такими как производительность, масштабируемость, удобство использования.
- Регрессионное тестирование .- Регрессионное тестирование проводится после исправления кода, обновлений или любого другого обслуживания системы, чтобы убедиться, что новый код не повлиял на существующий код.

Инструменты, используемые для тестирования черного ящика

Инструменты, используемые для тестирования «черного ящика», во многом зависят от типа тестирования «черного ящика», которое вы выполняете.

- Для функциональных/регрессионных тестов вы можете использовать QTP , Selenium.
- Для нефункциональных тестов вы можете использовать — LoadRunner , Jmeter

Методы тестирования черного ящика

Ниже приведены известные стратегии тестирования среди многих, используемых в тестировании черного ящика.

- Тестирование класса эквивалентности: оно используется для минимизации количества возможных тестовых случаев до оптимального уровня при сохранении разумного тестового покрытия.
- Тестирование граничных значений: тестирование граничных значений сосредоточено на значениях на границах. Этот метод определяет, является ли определенный диапазон значений приемлемым для системы или нет. Это очень полезно для сокращения количества тестовых случаев. Это наиболее подходит для систем, где вход находится в определенных диапазонах.
- Тестирование таблицы решений: таблица решений помещает причины и их последствия в матрицу. В каждом столбце есть уникальная комбинация.

Тестирования белого ящика

Тестирование « белого ящика» — это метод тестирования программного обеспечения, при котором внутренняя структура, дизайн и кодирование программного обеспечения тестируются для проверки потока ввода-вывода и улучшения дизайна, удобства использования и безопасности. При тестировании белого ящика код виден тестировщикам, поэтому его также называют тестированием в чистом ящике, тестированием в открытом ящике, тестированием в прозрачном ящике, тестированием на основе кода и тестированием в стеклянном ящике.

Это одна из двух частей подхода Box Testing к тестированию программного обеспечения. Его аналог, тестирование «черного ящика», включает тестирование с точки зрения внешнего или конечного пользователя. С другой стороны, тестирование «белого ящика» в разработке программного обеспечения основано на внутренней работе приложения и вращается вокруг внутреннего тестирования.

Что вы проверяете в тестировании белого ящика?

Тестирование «белого ящика» включает в себя тестирование программного кода на предмет следующего:

- Дыры внутренней безопасности
- Сломанные или плохо структурированные пути в процессах кодирования
- Поток определенных входных данных через код
- Ожидаемый результат
- Функциональность условных циклов
- Тестирование каждого оператора, объекта и функции на индивидуальной основе

Тестирование может проводиться на системном, интеграционном и модульном уровнях разработки программного обеспечения. Одной из основных целей тестирования белого ящика является проверка рабочего процесса приложения. Он включает в себя тестирование ряда предопределенных входных данных в сравнении с ожидаемыми или желаемыми выходными данными, поэтому, когда конкретный ввод не приводит к ожидаемому результату, вы столкнулись с ошибкой.

Как вы проводите тестирование белого ящика?

Чтобы дать вам упрощенное объяснение тестирования белого ящика, мы разделили его на два основных этапа . Вот что делают тестировщики при тестировании приложения с использованием метода тестирования белого ящика:

ШАГ 1) ПОНИМАЙТЕ ИСХОДНЫЙ КОД

Первое, что часто делает тестировщик, — это изучает и понимает исходный код приложения. Поскольку тестирование методом «белого ящика» включает тестирование внутренней работы приложения, тестировщик должен хорошо разбираться в языках программирования, используемых в приложениях, которые он тестирует. Кроме того, тестировщик должен хорошо разбираться в методах безопасного кодирования. Безопасность часто является одной из основных целей тестирования программного обеспечения. Тестировщик должен уметь обнаруживать проблемы с безопасностью и предотвращать атаки со стороны хакеров и наивных пользователей, которые могут сознательно или неосознанно внедрить вредоносный код в приложение.

Как вы проводите тестирование белого ящика?

ШАГ 2) СОЗДАЙТЕ ТЕСТ-КЕЙСЫ И ВЫПОЛНИТЕ

Второй основной шаг к тестированию белого ящика включает в себя проверку исходного кода приложения на правильность потока и структуры. Один из способов — написать дополнительный код для проверки исходного кода приложения. Тестировщик разработает небольшие тесты для каждого процесса или серии процессов в приложении. Этот метод требует от тестировщика глубоких знаний кода и часто выполняется разработчиком.

Методы тестирования белого ящика

Основным методом тестирования «белого ящика» является анализ покрытия кода. Анализ покрытия кода устраняет пробелы в наборе тестовых примеров. Он определяет области программы, которые не проверяются набором тестовых примеров. После выявления пробелов вы создаете тест-кейсы для проверки непроверенных частей кода, тем самым повышая качество программного продукта.

Типы тестирования белого ящика

Тестирование методом « белого ящика» включает в себя несколько типов тестирования, используемых для оценки удобства использования приложения, блока кода или конкретного программного пакета.

- Модульное тестирование: часто это первый тип тестирования приложения. Модульное тестирование выполняется для каждой единицы или блока кода по мере его разработки. Модульное тестирование в основном выполняется программистом. Как разработчик программного обеспечения, вы разрабатываете несколько строк кода, одну функцию или объект и тестируете их, чтобы убедиться, что они работают, прежде чем продолжить. Модульное тестирование помогает выявить большинство ошибок на ранних этапах жизненного цикла разработки программного обеспечения. Ошибки, обнаруженные на этом этапе, дешевле и их легко исправить.
- Тестирование на наличие утечек памяти: Утечки памяти являются основной причиной замедления работы приложений. Специалист по обеспечению качества, имеющий опыт обнаружения утечек памяти, необходим в тех случаях, когда у вас медленно работает программное приложение.

Преимущества тестирования белого ящика

- Оптимизация кода путем поиска скрытых ошибок.
- Случаи тестирования белого ящика можно легко автоматизировать.
- Тестирование более тщательное, так как обычно охватываются все пути кода.
- Тестирование можно начать раньше в SDLC, даже если графический интерфейс недоступен.

Недостатки тестирования WhiteBox

- Тестирование белого ящика может быть довольно сложным и дорогим.
- Разработчики, которые обычно выполняют тестовые случаи белого ящика, ненавидят его. Тестирование белого ящика разработчиками не является подробным, что может привести к производственным ошибкам.
- Для тестирования методом «белого ящика» требуются профессиональные ресурсы с подробным пониманием программирования и реализации.
- Тестирование методом «белого ящика» занимает много времени, для полного тестирования больших программных приложений требуется время.

Заключительные примечания:

- Тестирование белого ящика может быть довольно сложным. Сложность во многом связана с тестируемым приложением. Небольшое приложение, выполняющее одну простую операцию, можно протестировать в режиме «белого ящика» за несколько минут, в то время как для полного тестирования больших программных приложений требуются дни, недели и даже больше.
- Тестирование белого ящика при тестировании программного обеспечения должно выполняться для программного приложения по мере его разработки после его написания и снова после каждой модификации.

Тестирование серого ящика

Тестирование серого ящика — это метод тестирования программного обеспечения для тестирования программного продукта или приложения с частичным знанием внутренней структуры приложения. Целью тестирования серого ящика является поиск и выявление дефектов, связанных с неправильной структурой кода или неправильным использованием приложений.

В этом процессе обычно выявляются контекстно-зависимые ошибки, связанные с веб-системами. Это увеличивает охват тестированием, концентрируясь на всех уровнях любой сложной системы.

Тестирование серого ящика — это метод тестирования программного обеспечения, который представляет собой комбинацию методов тестирования белого ящика и метода тестирования черного ящика.

- При тестировании White Box известна внутренняя структура (код)
- При тестировании Black Box внутренняя структура (код) неизвестна
- В Gray Box Testing внутренняя структура (код) известна частично

Тестирование серого ящика

В программной инженерии тестирование серого ящика дает возможность тестировать обе стороны приложения, уровень представления, а также часть кода. В первую очередь это полезно при тестировании интеграции и тестировании на проникновение .

Пример тестирования серого ящика: при тестировании функций веб-сайтов, таких как ссылки или потерянные ссылки, если тестер сталкивается с какой-либо проблемой с этими ссылками, он может сразу внести изменения в HTML-код и проверить в режиме реального времени.

Почему тестирование серого ящика

Тестирование серого ящика выполняется по следующей причине:

- Он сочетает в себе преимущества как тестирования черного ящика, так и тестирования белого ящика.
- Он объединяет вклад разработчиков и тестировщиков и улучшает общее качество продукта.
- Это уменьшает накладные расходы на длительный процесс тестирования функциональных и нефункциональных типов.
- Это дает разработчику достаточно свободного времени для исправления дефектов.
- Тестирование выполняется с точки зрения пользователя, а не с точки зрения дизайнера.

Стратегия тестирования серого ящика

Для выполнения тестирования серого ящика не обязательно, чтобы тестер имел доступ к исходному коду. Тест разработан на основе знания алгоритма, архитектуры, внутренних состояний или других высокоуровневых описаний поведения программы.

Чтобы выполнить тестирование серого ящика:

- Он применяет простую технику тестирования черного ящика.
- Он основан на генерации тестовых случаев требований, поэтому он предварительно устанавливает все условия перед тем, как программа будет протестирована методом утверждений.

Методы, используемые для тестирования серого ящика:

- Матричное тестирование: этот метод тестирования включает в себя определение всех переменных, которые существуют в их программах.
- Регрессионное тестирование : чтобы проверить, не привели ли изменения в предыдущей версии к регрессу других аспектов программы в новой версии. Это будет сделано путем тестирования стратегий, таких как повторное тестирование всех, повторное тестирование рискованных вариантов использования, повторное тестирование в брандмауэре.
- Orthogonal Array Testing или OAT : он обеспечивает максимальное покрытие кода с минимальным количеством тестовых случаев.
- Тестирование шаблонов: это тестирование выполняется на исторических данных о предыдущих дефектах системы. В отличие от тестирования черного ящика, тестирование серого ящика копается в коде и определяет, почему произошел сбой.

Шаги для выполнения тестирования серого ящика:

- Шаг 1 : Определите входные данные
- Шаг 2 : Определите результаты
- Шаг 3 : Определите основные пути
- Шаг 4 : Определите подфункции
- Шаг 5 : Разработка входных данных для подфункций
- Шаг 6 : Разработка результатов для подфункций
- Шаг 7 : Выполните тестовый пример для подфункций
- Шаг 8. Проверьте правильный результат для подфункций.
- Шаг 9 : Повторите шаги 4 и 8 для других подфункций.
- Шаг 10 : Повторите шаги 7 и 8 для других подфункций.

Тестовые случаи для тестирования серого ящика могут включать в себя, связанные с графическим интерфейсом, связанные с безопасностью, связанные с базой данных, связанные с браузером, связанные с операционной системой и т. д.

Проблемы тестирования серого ящика

- Когда тестируемый компонент сталкивается с каким-либо сбоем, это может привести к прерыванию текущей операции.
- Когда тест выполняется полностью, но содержание результата неверно.

Итоги

- Общая стоимость системных дефектов может быть снижена и предотвращена от дальнейшего прохождения с помощью тестирования серого ящика.
- Тестирование серого ящика больше подходит для GUI, функционального тестирования , оценки безопасности, веб-приложений, веб-сервисов и т. д.
- Методы, используемые для тестирования серого ящика
  - Матричное тестирование
  - Регрессионное тестирование
  - OAT или тестирование ортогонального массива
  - Тестирование шаблонов

Что такое УАТ?

Пользовательское приемочное тестирование (UAT) — это тип тестирования, выполняемого конечным пользователем или клиентом для проверки/принятия программной системы перед перемещением программного приложения в производственную среду. UAT выполняется на заключительном этапе тестирования после функционального, интеграционного и системного тестирования.

Назначение УАТ

Основной целью UAT является проверка сквозного бизнес-потока. Он не фокусируется на косметических ошибках, орфографических ошибках или системном тестировании. Пользовательское приемочное тестирование проводится в отдельной среде тестирования с настройкой данных, аналогичной производственной. Это своего рода тестирование «черного ящика», в котором участвуют два или более конечных пользователя.

УАТ выполняется:

- Клиент
- Конечные пользователи

Необходимость пользовательского приемочного тестирования

Необходимость пользовательского приемочного тестирования возникает после того, как программное обеспечение прошло модульное, интеграционное и системное тестирование, потому что разработчики могли создать программное обеспечение на основе документа с требованиями по своему собственному пониманию, и дальнейшие необходимые изменения во время разработки могут быть им не доведены до сведения, поэтому для проверки того, является ли окончательный продукт принят клиентом/конечным пользователем, требуется приемочное тестирование пользователем.

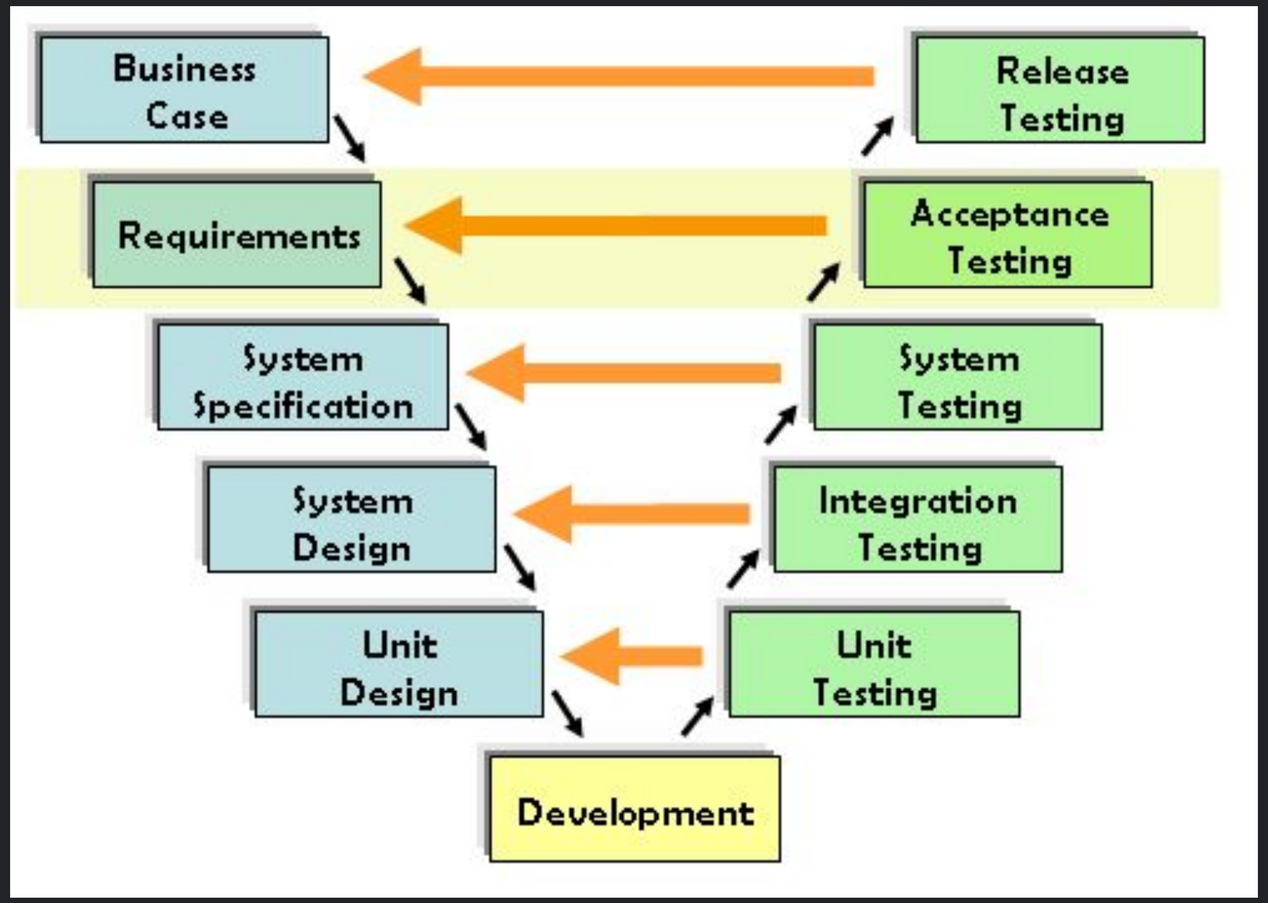
Необходимость пользовательского приемочного тестирования

Разработчики кодируют программное обеспечение на основе документа с требованиями, который является их «собственным» пониманием требований и может на самом деле не соответствовать тому, что клиенту нужно от программного обеспечения .

Изменения требований в ходе проекта могут быть неэффективно доведены до сведения разработчиков.

Приемочные испытания и V-модель

В VModel приемочное тестирование пользователей соответствует фазе требований жизненного цикла разработки программного обеспечения (SDLC).



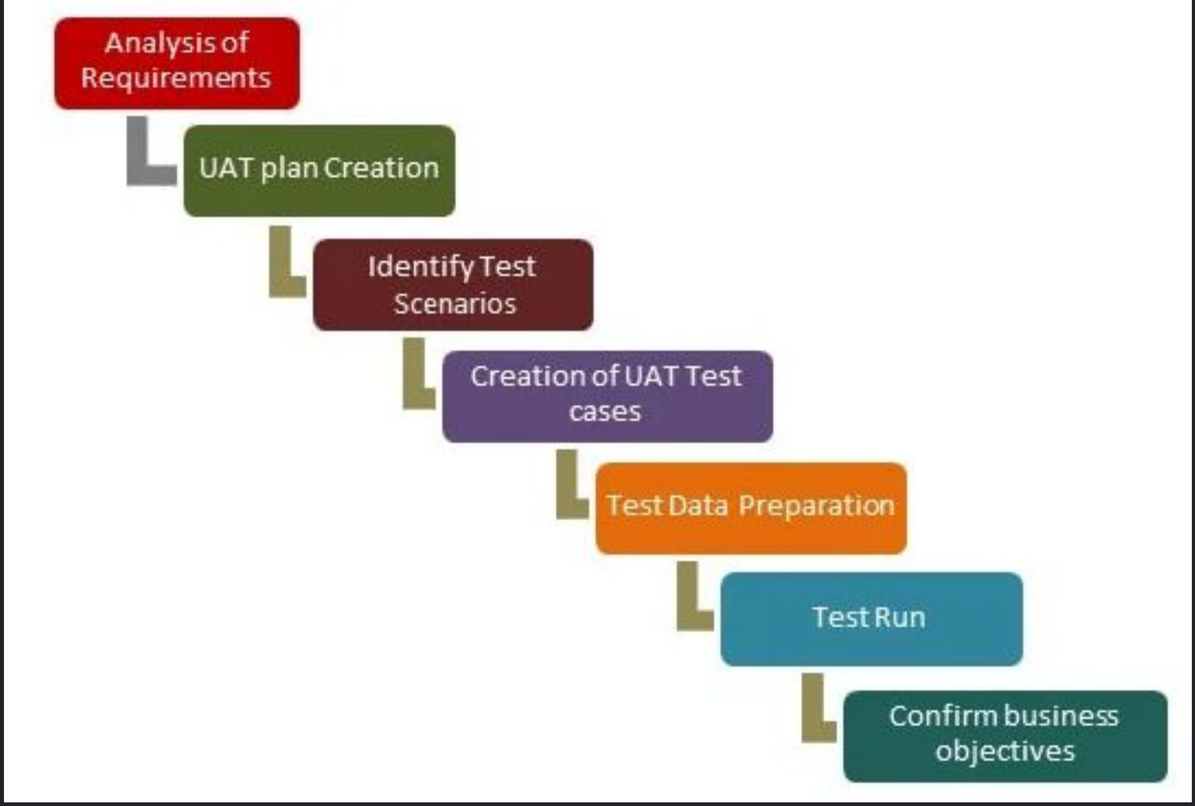
Предпосылки пользовательского приемочного тестирования:

Ниже приведены критерии входа для пользовательского приемочного тестирования:

- Должны быть доступны бизнес-требования.
- Код приложения должен быть полностью разработан
- Модульное тестирование, интеграционное тестирование и системное тестирование должны быть завершены
- Отсутствие отличных результатов, высокие и средние дефекты на этапе тестирования системной интеграции —
- До UAT допустима только косметическая ошибка.
- Регрессионное тестирование должно быть завершено без серьезных дефектов
- Все заявленные дефекты должны быть исправлены и протестированы до UAT.
- Матрица прослеживаемости для всех испытаний должна быть заполнена
- Среда UAT должна быть готова
- Подпишите почту или сообщение от группы тестирования системы о том, что система готова к выполнению UAT.

Как выполнять тесты UAT

UAT выполняется предполагаемыми пользователями системы или программного обеспечения. Этот тип тестирования программного обеспечения обычно происходит в месте расположения клиента и называется бета-тестированием. После того, как критерии входа для UAT удовлетворены, тестеры должны выполнить следующие задачи:



1. Анализ бизнес-требований
2. Создание плана тестирования UAT
3. Определите сценарии тестирования
4. Создание тестовых случаев UAT
5. Подготовка тестовых данных (производство как данные)
6. Запустите тестовые случаи
7. Запишите результаты
8. Подтвердить бизнес-цели

Шаг 1) Анализ бизнес-требований

Одним из наиболее важных действий в UAT является определение и разработка тестовых сценариев. Эти тестовые сценарии основаны на следующих документах:

- Устав проекта
- Варианты использования в бизнесе
- Диаграммы технологического процесса
- Документ бизнес-требований (BRD)
- Спецификация системных требований (SRS)

Шаг 2) Создание плана UAT:

План тестирования UAT описывает стратегию, которая будет использоваться для проверки и обеспечения соответствия приложения его бизнес-требованиям. Он документирует критерии входа и выхода для UAT, сценарии тестирования и подход к тестам, а также сроки тестирования .

Шаг 3) Определите тестовые сценарии и тестовые наборы:

Определите тестовые сценарии в отношении высокоуровневого бизнес-процесса и создайте тестовые сценарии с четкими этапами тестирования. Тестовые случаи должны в достаточной степени охватывать большинство сценариев UAT. Сценарии бизнес-использования являются входными данными для создания тестовых случаев.

Шаг 4) Подготовка тестовых данных:

Для UAT лучше всего использовать оперативные данные. Данные должны быть зашифрованы из соображений конфиденциальности и безопасности . Тестер должен быть знаком с потоком базы данных.

Шаг 5) Запустите и запишите результаты:

Выполняйте тестовые случаи и сообщайте об ошибках, если таковые имеются. Повторно протестируйте ошибки после исправления. Для выполнения можно использовать инструменты управления тестированием .

Шаг 6) Подтвердите достижение бизнес-целей:

Бизнес-аналитики или тестировщики UAT должны отправить электронное письмо с подписью после тестирования UAT. После согласования продукт можно отправлять в производство. Результатами тестирования UAT являются план тестирования, сценарии и примеры тестирования UAT, результаты тестирования и журнал дефектов.

Критерии выхода из UAT:

Прежде чем приступить к производству, необходимо принять во внимание следующее:

- Нет открытых критических дефектов
- Бизнес-процесс работает удовлетворительно
- UAT Подписание встречи со всеми заинтересованными сторонами

Качества тестировщиков UAT:

Тестировщик UAT должен хорошо разбираться в бизнесе. Он должен быть самостоятельным и мыслить как неизвестный системе пользователь . Тестировщик должен быть аналитическим и нестандартным мыслителем и объединять все виды данных, чтобы сделать UAT успешным.

Тестировщик, бизнес-аналитик или профильные эксперты, которые понимают бизнес-требования или потоки, могут подготовить тесты и данные, которые являются реалистичными для бизнеса.

Лучшие практики:

Для успеха UAT необходимо учитывать следующие моменты:

- Подготовьте план UAT на ранней стадии жизненного цикла проекта
- Подготовьте контрольный список до начала UAT
- Проведение сеанса Pre-UAT во время самого этапа тестирования системы
- Четко определите ожидания и область применения UAT
- Тестируйте сквозной бизнес-процесс и избегайте системных тестов
- Протестируйте систему или приложение с помощью реальных сценариев и данных.
- Думайте как Неизвестный пользователь системы
- Проведите юзабилити-тестирование
- Проведите сеанс обратной связи и совещание перед переходом к производству

Инструменты UAT

На рынке существует несколько инструментов, используемых для приемочного тестирования пользователей, и некоторые из них перечислены для справки:

Инструмент Java, используемый в качестве механизма тестирования. Легко создавать тесты и записывать результаты в таблицу. Пользователи инструмента вводят форматированный ввод, и тесты создаются автоматически. Затем выполняются тесты, и результат возвращается пользователю.

Watir : это инструментарий, используемый для автоматизации тестов на основе браузера во время приемочного тестирования пользователей. Ruby — это язык программирования, используемый для межпроцессного взаимодействия между ruby и Internet Explorer.

Пример руководства для UAT

- В большинстве случаев в обычных сценариях разработки программного обеспечения UAT выполняется в среде QA. Если нет промежуточной среды или среды UAT
- UAT подразделяется на бета- и альфа-тестирование, но это не так важно, когда программное обеспечение разрабатывается для сферы услуг.
- UAT имеет больше смысла, когда клиент вовлечен в большую степень

Вывод:

- В программной инженерии полной формой UAT является пользовательское приемочное тестирование.
- UAT — это одна из многих разновидностей тестирования, появившихся за последние двадцать пять лет.
- С UAT клиент может быть уверен, «чего ожидать» от продукта, а не строить предположения.
- Преимущество UAT в том, что не будет сюрпризов, когда продукт будет выпущен на рынок.

Регрессионное тестирование

Регрессионное тестирование определяется как тип тестирования программного обеспечения для подтверждения того, что недавнее изменение программы или кода не повлияло отрицательно на существующие функции. Регрессионное тестирование — это не что иное, как полный или частичный выбор уже выполненных тестовых случаев, которые выполняются повторно, чтобы убедиться, что существующие функции работают нормально.

Это тестирование проводится, чтобы убедиться, что новые изменения кода не будут иметь побочных эффектов для существующих функций. Это гарантирует, что старый код по-прежнему будет работать после внесения последних изменений в код.

Необходимость регрессионного тестирования

Необходимость регрессионного тестирования в основном возникает всякий раз, когда требуется изменить код, и нам нужно проверить, влияет ли измененный код на другую часть программного приложения или нет. Более того, регрессионное тестирование необходимо при добавлении новой функции в программное приложение и для устранения дефектов, а также устранения проблем с производительностью.

Как проводить регрессионное тестирование

Чтобы выполнить процесс регрессионного тестирования, нам нужно сначала отладить код, чтобы выявить ошибки. После выявления ошибок вносятся необходимые изменения для их исправления, затем выполняется регрессионное тестирование путем выбора соответствующих тестовых случаев из набора тестов, который охватывает как измененные, так и затронутые части кода.

Обслуживание программного обеспечения — это деятельность, которая включает улучшения, исправления ошибок, оптимизацию и удаление существующих функций. Эти модификации могут привести к неправильной работе системы. Поэтому регрессионное тестирование становится необходимым.

Выбор регрессионного теста

Выбор регрессионного теста — это метод, при котором выполняются некоторые выбранные тестовые примеры из набора тестов, чтобы проверить, влияет ли измененный код на программное приложение или нет. Тестовые наборы делятся на две части: повторно используемые тестовые наборы, которые можно использовать в дальнейших циклах регрессии, и устаревшие тестовые наборы, которые нельзя использовать в последующих циклах.

Приоритизация тестовых случаев

Расставьте приоритеты тестовых случаев в зависимости от влияния на бизнес, критических и часто используемых функций. Выбор тестовых случаев на основе приоритета значительно сократит набор регрессионных тестов.

Выбор тестовых случаев для регрессионного тестирования

Из отраслевых данных было обнаружено, что большое количество дефектов, о которых сообщили клиенты, были вызваны исправлениями ошибок в последнюю минуту, создающими побочные эффекты, и, следовательно, выбор тестового примера для регрессионного тестирования — это искусство, и это не так просто. Эффективные регрессионные тесты можно выполнить, выбрав следующие тестовые примеры:

- Тест-кейсы с частыми дефектами
- Функциональность, которая более заметна для пользователей
- Тестовые случаи, которые проверяют основные функции продукта
- Тестовые случаи функциональных возможностей, которые претерпели больше и недавние изменения
- Все интеграционные тестовые случаи
- Все сложные тестовые случаи
- Тестовые примеры граничных значений
- Образец успешных тестовых случаев
- Пример тестовых случаев отказа

Регрессионное тестирование и управление конфигурацией

Управление конфигурацией во время регрессионного тестирования становится обязательным в гибких средах, где код постоянно модифицируется. Чтобы обеспечить эффективность регрессионных тестов, обратите внимание на следующее:

- Код, подвергаемый регрессионному тестированию, должен находиться под инструментом управления конфигурацией.
- На этапе регрессионного тестирования нельзя допускать никаких изменений в коде. Код регрессионного тестирования должен быть защищен от изменений, вносимых разработчиком.
- База данных, используемая для регрессионного тестирования, должна быть изолирована. Никакие изменения базы данных не должны быть разрешены

Разница между повторным тестированием и регрессионным тестированием

Повторное тестирование означает повторное тестирование функциональности или ошибки, чтобы убедиться, что код исправлен. Если это не исправлено, необходимо повторно открыть дефект . Если исправлено, дефект закрывается.

Регрессионное тестирование означает тестирование вашего программного приложения, когда в нем происходит изменение кода, чтобы убедиться, что новый код не повлиял на другие части программного обеспечения.

Проблемы регрессионного тестирования

Ниже приведены основные проблемы тестирования при проведении регрессионного тестирования:

- При последовательных запусках регрессии наборы тестов становятся довольно большими. Из-за ограничений по времени и бюджету весь набор регрессионных тестов не может быть выполнен.
- Сведение к минимуму набора тестов при достижении максимального покрытия тестами остается сложной задачей.
- Определение частоты регрессионных тестов, т. е. после каждой модификации или каждого обновления сборки или после множества исправлений ошибок, является сложной задачей.

Вывод

Значение регрессионного тестирования. Регрессионное тестирование — это тип тестирования программного обеспечения, который гарантирует, что приложение по-прежнему работает должным образом после улучшений, любых изменений кода или обновлений.

Эффективная стратегия регрессии, позволяющая организациям сэкономить время и деньги. Согласно одному из тематических исследований в банковской сфере, регрессия экономит до 60% времени на исправлении ошибок (которые были бы обнаружены регрессионными тестами) и 40% на деньгах.