Урок №1

Введение в тестирование.

Содержание

1. Что такое тестирование программного обеспечения?
2. Необходимость тестирования
3. Цели и задачи тестирования
4. Базовая терминология тестирования
    a. Баг
    b. Тестовые данные
    c. Тестовая ситуация
    d. Отказ
5. Тестировщик и QA инженер
    a. Кто такой тестировщик?
    b. Цели и задачи тестировщика
    c. Кто такой QA инженера?
    d. Цели и задачи QA инженера
6. Цели и задачи тестировщика и QA инженера
7. Тестирование в контексте процесса разработки программного обеспечения
    a. Обзор стадий разработки
    b. Обзор моделей разработки
    c. Цели и задачи тестирования на различных стадиях процесса разработки в различных моделях
    d. Жизненный цикл тестирования (Testing Life-Cycle)
        1. Анализ требований (Requirements Analysis)
        2. Анализ дизайна проекта (Design Analysis)

3. Планирование тестирования (Test planning)
4. Разработка тестов (Test development)
5. Выполнение тестов (Test Execution)
6. Написание отчетов (Test Reporting)
7. Повторная проверка дефектов (Retesting the Defects)

e. График стоимости поиска дефекта на различных стадиях разработки проекта

8. Подходы к тестированию
   a. Exploratory (ознакомительное) и Scripted (по сценарию) тестирование
   b. Manual (ручное) и Automated (автоматическое) тестирование
   c. Тестирование Black Box(черный ящик) и White Box (белый ящик)
   d. Positive (позитивное) и Negative (негативное) тестирование
9. Виды ошибок
   a. Ошибки пользовательского интерфейса
      1. Функциональность
      2. Взаимодействие программы с пользователем
      3. Организация программы
      4. Пропущенные команды
      5. Производительность
      6. Выходные данные
      7. Обработка ошибок
   b. Ошибки обработки граничных условий
   c. Ошибки вычислений
   d. Ошибки управления потоком
   e. Ошибки передачи или интерпретации данных
   f. Ошибки в результате гонок
   g. Ошибки, связанные с перегрузками и аппаратным обеспечением
   h. Ошибки, связанные с контролем версий
   i. Ошибки документации
   j. Ошибки тестирования

1. Что такое тестирование программного обеспечения?

Начиная изучение курса «Тестирование программного обеспечения», необходимо вначале поговорить о тестировании в широком смысле, то есть о тестировании как таковом.

В узком понимании, тестирование – это процесс определения соответствия объекта тестирования заданным спецификациям (характеристикам). В контексте разработки, соответственно, тестирование понимается как процесс определения соответствия продукта изначальным спецификациям, которые были заданны техническим заданием.

Если рассматривать тестирование в более широком смысле, то его можно характеризовать как процесс опытного (экспериментального) анализа функциональности некоторой исследуемой системы. При этом постулируется, что всякий случай тестирования можно определить как исследование. Например, тестирование с целью выявление структурных дефектов (дефектов конструкции) можно определить, как исследование системы на предмет наличия дефектов конструкции.

Всякое тестирование подразумевает два действующих лица: субъект и объект тестирования. Субъектом тестирования (то есть источником активности) выступает «тестировщик» (англ. tester) – специально назначенное лицо, в обязанности которого входит выполнение тестирования. Под объектом тестирования понимается исследуемая система. При этом в роли тестируемой системы может выступать как весь продукт в его полноте, так и отдельные его модули, составные части. При тестировании программного обеспечения, в общем случае, объектом тестирования является версия разрабатываемого программного продукта. Разумеется, что под версией подразумевается не только «финальная» (конечная) версия, но и промежуточные версии, получаемые в ходе разработки.

При проведении тестирования всегда имеет место «внешнее воздействие»

на систему со стороны тестировщика, которое также называют «тестовым воздействием». Тестирование заключается в осуществлении тестировщиком направленного воздействия на тестируемую систему, предполагающего получение некоторой ожидаемой реакции, называемой «тестовой реакцией», появление которой должно подтвердить соответствие системы конкретной отдельно взятой спецификации. Получение обратной (по отношению к ожидаемой) или неспецифичной реакции позволяет говорить о несоответствии системы спецификации и принимать решение о возврате программного продукта на доработку с целью устранения дефекта.

Как и в моделировании, в тестировании понятие системы является ключевым, поскольку наше представление об исследуемом объекте является первичным по отношению к эксперименту. Другими словами характер эксперимента зависит от наших знаний о структурном составе и характере связей между компонентами тестируемой системы. Напомним, что под системой понимается некоторый образ, обобщённо или полно описывающий исследуемый нами объект. Таким образом, можно выделить два крайних случая: в первом – мы обладаем абсолютно полной информацией о тестируемой системе; во втором – тестируемая система выступает в роли «чёрного ящика», то есть мы обладаем лишь знанием о входах системы и можем фиксировать получаемые выходы (результаты внутренней активности системы).

Весь процесс тестирования можно условно поделить на тестовые задания, каждое из которых заключается в исследовании какой-то отдельной черты испытуемой системы. Тестовое задание, как правило, состоит из описания цели задания, перечисления и указания порядка тестовых воздействий и формализованной фиксации тестовой реакции, а так же из заключения о том, соответствует ли система тестируемой спецификации.

При наличии исчерпывающей информации о системе мы имеем возможность составлять тестовые задания, основываясь на знании о внутренней логике функционирования системы. Например, зная, как входной

сигнал преобразуется внутри системы, мы можем подавать на входы системы некорректные данные с целью тестирования её отказоустойчивости. В описанном случае, возможность формирования некорректных данных зависит от знания о том, какие данные ожидаемы системой.

В случае же, когда мы обладаем неполной информацией об устройстве системы, тестирование меняет свою форму. Как правило, мы всегда имеем основания что-либо предполагать о характере функционирования испытуемой системы. В таких условиях, тестовые задания будут формироваться на основе наших предположений.

В тех случаях, когда мы не обладаем никакой информацией о системе, тестирование принимает исследовательский характер. В таком случае наша задача состоит в том, чтобы определить – как выходы зависят от входов, или, другими словами, как реакции системы зависят от внешних воздействий; и, на основании полученной информации, определить, соответствует ли система заданным спецификациям.

Тестирование на производстве можно рассматривать в двух смыслах: во-первых, тестирование применяется для выявления дефектов конструкции на этапе разработки, а во-вторых – для проверки соответствия конечных экземпляров готовой продукции критериям качества, определённым техническими спецификациями. В обоих вышеуказанных смыслах тестирование применяется при производстве продукции любого вида. Само собой разумеется, что в контексте производства программной продукции в основном речь идёт о тестировании на этапе разработке, поскольку, все экземпляры готового программного продукта заведомо идентичны и способ изготовления конечной продукции исключает возникновение дефектов. Строго говоря, процесс производства программного продукта можно ограничить этапом разработки.

Тестирование программного обеспечения – это отдельный подвид тестирования, характеризующийся свойственными ему одному качествами, которые происходят из специфики объектов тестирования.

Во-первых, необходимо отметить, что программные продукты, в силу различных причин, имеют модульную структуру и уровневую организацию, и, как правило, функционируют не обособленно, а в рамках некоторой инфраструктуры (совместно с другими приложениями, программными комплексами, а также в рамках программных сред); всё вышеуказанное приводит к возникновению различных уровней тестирования:

- *компонентное или модульное тестирование* (component testing or unit testing) заключается в тестировании различных модулей приложения, которые могут быть протестированы по отдельности в силу своей автономности. Признаком, образующим данный уровень является то, что тестирование отдельных модулей можно производить на ранних этапах разработки, пока работа над остальными модулями продолжается;
- *интеграционное тестирование* (integration testing) представляет собой исследование связей между компонентами приложения, а также исследования взаимодействия приложения со средой, в рамках которой оно будет исполняться;
- *системное тестирование* (system testing) направлено на исследование функциональных и нефункциональных особенностей системы в целом;
- *приёмочное тестирование* (acceptance testing) проводиться на финальном этапе разработки и направлено на выяснение того, соответствует ли система приёмочно/сдаточным критериям.

Во-вторых, тестирование программного обеспечения имеет свойственные одному ему виды и методы тестирования. Таким образов, в зависимости от целей тестирования, выделяют три основных типа тестирования программного обеспечения:

Функциональные виды тестирования связанны с исследованием внешнего поведения системы, то есть выполняемых ею функций. Сюда относятся такие виды тестирования, как:

- *функциональное тестирование* (functional testing) направлено на

проверку корректности выполняемых системой функций и может присутствовать на всех уровнях тестирования;

- *тестирование безопасности* (security and access control testing) направлено на проверку безопасности системы, а также оценку целостности подхода к защите приложения от несанкционированного доступа и защите конфиденциальных данных;
- *тестирование взаимодействия* (interoperability testing) направлено на оценку возможности приложения взаимодействовать с внешними компонентами или системами, а также включает тестирование совместимости (compatibility testing) и интеграционное тестирование (integration testing).

Нефункциональные виды тестирования направлены на проверку всех нефункциональных особенностей системы. Сюда относятся тесты специфичных для программных продуктов характеристик:

- *тестирование установки* (installation testing) направлено на проверку процесса инсталляции системы, а также процессов настойки, удаления и обновления программного обеспечения;
- *тестирование удобства использования* (usability testing) связано с оценкой степени удобства использования, а также понятности и привлекательности пользовательского интерфейса приложения;
- *тестирование на отказ и восстановление* (failover and recovery testing) связанно с оценкой средств обеспечения отказоустойчивости и надёжности системы;
- *конфигурационное тестирование* (configuration testing) направлено на проверку функциональности системы при всех возможных, поддерживаемых системой, конфигурациях программного обеспечения и оборудования;
- *тестирования производительности* состоит из таких методов тестирования как: *тестирование нагрузки* (performance and load testing) состоит в исследовании реакции системы на

функционирование в условиях нагрузки (имеется в виду нагрузка в пределах нормы); *стрессовое тестирование* (stress testing) предполагает исследование поведения системы при функционировании в условия перегрузки (нагрузки превышающей штатную); *тестирование стабильности и надёжности* (stability/reliability testing) направлено на изучение поведения системы в условиях нормальной нагрузки при длительном функционировании, *объёмное тестирование* (volume testing) используется для оценки поведения системы при условии увеличения объёма данных, обрабатываемых приложением.

Виды тестирования **связанные с изменениями**. Тесты данной группы направлены на то, чтобы подтвердить, что обнаруженная на предыдущих этапах тестирования проблема была действительно решена. К данному виду можно отнести следующие тесты:

- *дымовое тестирование* (smoke testing) направлено на обзорную проверку всех компонентов приложения на предмет работоспособности, а также на выявление грубых дефектов, наличие которых можно определить, так сказать, «невооружённым глазом»;
- *регрессионное тестирование* (regression testing) предназначено для проверки осуществлённых в системе изменений, а так же на подтверждение того, что существовавшая до изменения функциональность, работает также как и прежде;
- *тестирование сборки* (build verification testing) направлено на проверку соответствия выпускаемой версии программного продукта критериям качества, необходимым для начала тестирования;
- *санитарное тестирование* или иначе *проверка согласованности/исправности* (sanitary testing) состоит в проведении теста достаточного для подтверждения того, что определённая отдельно взятая функция работает соответственно заявленным спецификациям.

2. Необходимость тестирования

На сегодняшний день, к программным продуктам предъявляются достаточно высокие требования в области качества, в силу большой конкуренции и широкого спектра предложений во многих сферах рынка программного обеспечения.

Качество определяют из двух составляющих: спецификаций производителя и спецификаций потребителя. Спецификации производителя выражают объективные требования к программному обеспечению, порождаемые свободной конкуренцией. Другими словами, выдвигая требования к конечному продукту, производитель исходит из того, что программное обеспечение должно быть конкурентоспособным, но приводить к минимально возможным затратам на разработку. Спецификации потребителя выражают общие ожидания конечных пользователей относительно разрабатываемого продукта. Соответствие обоим описанным спецификациям и понимается под качеством программного обеспечения.

Очевидно, что не все ожидания можно выразить в формальном виде, а значит не всегда возможно измерить, соответствует ли продукт ожиданиям, или нет. Поэтому принято говорить об измеряемых ожиданиях, которые находят выражение в спецификациях – заранее заданных формальных требованиях к конечному продукту.

Тестирование позволяет контролировать заданные спецификации на всех этапах разработки, и, следовательно, является частью обеспечения качества (quality assurance) конечного продукта.

Таким образом, тестирование следует рассматривать, как необходимый и обязательный этап разработки программного обеспечения.

3. Цели и задачи тестирования

Общая цель тестирования – выявление дефектов программного обеспечения, которая является одной из целей обеспечения качества в рамках унифицированного процесса разработки программного обеспечения. Однако тестирование не ограничивается одним выявлением дефекта, оно также должно контролировать исправление выявленного недостатка. Таким образом, можно определить следующие цели тестирования программного обеспечения:

1. выявление дефектов на различных этапах жизненного цикла приложения (в процессе разработки, во время сопровождения и т.д.);
2. проверка того, что выявленный дефект был успешно устранён;
3. выяснение того, что изменения, связанные с устранением выявленного дефекта, не привнесли новых дефектов в систему.

Исходя из целей, перед тестированием ставятся следующие задачи:

- обнаружение дефектов модели системы;
- обнаружение дефектов кодирования;
- выявление ошибок и недостатков взаимодействия системы с окружением, а также внешними компонентами и системами;
- выявление дефектов интеграции программного обеспечения;
- выявление недостатков производительности системы;
- выявление неустойчивости программного обеспечения к перегрузкам;
- выявление неустойчивости программного обеспечения к вводу ошибочных данных и отказа при увеличении объёмов обрабатываемых данных;
- выявление уязвимостей в системе безопасности приложения и возможностей несанкционированного доступа к конфиденциальным данным;
- контроль исправления, выявленных в процессе тестирования дефектов;
- выявление регрессии системы в процессе разработки.

4. Базовая терминология тестирования

Баг

Термин «баг» (англ. bug - жук) – это современное сленговое выражение, означающее ошибку проектирования или разработки, происходящую во время выполнения приложения. Этимология термина «баг» не достаточно изучена, поскольку это выражение укрепилось в IT-сленге стихийно, однако, существует несколько версий истории о том, кто первым использовал выражение «баг» в том смысле, в котором оно сейчас употребляется IT-сообществом.

Наиболее устоявшейся легендой о происхождении слова баг считается история о том, как в 1947 году учёными Гарвардского университета во время тестирования вычислительной машины Mark II Aiken Relay Calculator был найден мотылёк, который застрял между контактами электромеханического реле №70 на панели F. Найденное насекомое было вклеено посредством скотча в технический журнал с сопроводительной надписью: «Первый фактический случай обнаружения жука (бага)» (англ. «First actual case of bug being found»). И именно это выражение принято считать первым употреблением слова баг применительно к компьютерной технике.

В тестировании программного обеспечения для обозначения ошибки принято пользоваться термином «дефект», однако сегодня слово баг повсеместно используется для обозначения любого рода ошибок приложения на всех этапах разработки.

Происхождение большинства ошибок очевидно, и, соответственно, их исправление не вызывает сложностей. Но существуют, также, ошибки, обнаружить которые не так уж и просто (в силу различных причин). Выделяют следующие породы «жуков», которых сложно поймать:

- *«гейзинбаг» (англ. heisenbug)* – это программная ошибка, которая исчезает или изменяет свои характеристики при попытке её обнаружить. В качестве примера гейзинбага можно привести ошибки,

которые имеют место при обычной компиляции, но пропадают в режиме отладки (при компиляции с помощь оптимизирующего компилятора; другими словами, при создании дебаг-версии программы).

- *«борбаг» (англ. bohrbug)* – это такая программная ошибка, поведение которой определено некоторым количеством жёстко определённых (но, возможно, неизвестных) условий. Баги такого типа не изменяют своего поведения и не исчезают при попытке их обнаружить, а также являются наиболее часто встречаемыми багами среди сложно-устранимых. Однако они могут появляться только при определённых условиях (например, если были введены некоторые специфические данные, или при определённых настройках приложения, или даже, возможно, что при определённом состоянии окружения). Поэтому опасность данного типа ошибок состоит в том, что они могут быть пропущены тестировщиком, и об их наличии станет известно только спустя некоторое время после начала эксплуатации приложения конечным пользователем.
- *«мандельбаг» (англ. mandelbug)* - программная ошибка, чьё поведение настолько сложно (определяется большим количеством факторов), что может показаться хаотичным, или даже абсолютно недетерминированным (беспричинным, не предопределённым). Также мандельбагом называют ошибку, которая скорее кажется тестировщику борбагом, нежели гейзинбагом. Примером мандельбага может служить ситуация, при которой существует задержка во времени (между ошибочным действием и её проявлением) достаточная, чтобы ошибка показалась беспричинной. Также бывают ситуации, при которых концептуальные ошибки устройства системы приводят к появлению багов, на первый взгляд, кажущихся беспричинными.
- *«шрёдинбаг» (англ. schrodinbug)* – это программная ошибка, которая

никак не проявляется (или является неизвестной), до тех пор, пока кто-нибудь не обнаружит её, прочитав исходный код или использовав программы в необычных (не предусмотренных) условиях. После обнаружения шрёдинбага, как правило, становиться непонятно, как программа функционировала до этого момента (или просто казалось, что она функционирует).

- *«статистический баг» (англ. statistical bug)* – это такая программная ошибка, которая может быть обнаружена только при агрегации достаточно большого количества результатов тестов. Другими словами, отдельные испытания (и даже их небольшое количество) не выявляют ошибки – она становиться видна, только если рассматривать большое количество результатов одновременно.

  Данный тип ошибок специфичен для программ, которые производят случайный или псевдослучайный вывод. Примером может служить алгоритм случайности, производящий неравномерный вывод (то есть, большая часть выходных значений сосредоточена в каком-то отдельном диапазоне). Дефект алгоритма будет не виден при малом количестве испытаний, но если провести достаточное количество испытаний и рассмотреть все выходы вместе, то ошибочность алгоритма станет очевидна.

Тестовые данные

Всякое программное обеспечение, так или иначе, обрабатывает некоторые данные. Данные можно определить, как сведения о некотором событии, факте, происшествии и т.д., представленные в формальном виде и подлежащие дальнейшему анализу. Анализ и преобразование данных и представляют собой основные задачи программного обеспечения (важно помнить разницу между данными и информацией, которую можно получить на основании данных).

Таким образом, для проведения тестирования программного обеспечения необходимы данные, которые бы симулировали реальный поток, обрабатываемы программой.

Существует два типа тестовых данных: реальные и синтезированные. Реальные данные можно получить двумя способами: первый состоит в том, чтобы тестировать программное обеспечение в реальных условиях (что практически не возможно); второй – получать реальные данные у конечного пользователя (что возможно достаточно редко) или из существующей системы (в этом случае могут возникнуть проблемы с конфиденциальностью). Поэтому, наиболее часто для тестирования используют синтезированные данные.

Основной условием синтеза данные является их правдоподобность. Синтезировать данные можно на основании образца (небольшого количества реальных данных), статистической информации о том, какие данные являются типичными для конкретного тестового случая, а также другого набора условий, ограничивающих спектр допустимых вариаций входных данных. На базе имеющейся информации строиться генератор тестовых данных, который формирует необходимый для тестирования поток.

Тестовая ситуация

Тестовая ситуация – это базисное понятие тестирования. Под тестовой ситуацией понимается некоторое определённое состояние тестируемой системы, которому соответствует строго определённый набор её параметров; также тестовую ситуацию можно определить как совокупность внешних и внутренних факторов, позволяющих различить условия выполнения программы. Если один из параметров меняется – изменится и тестовая ситуация. Исходя из этого, полный набор всех возможных тестовых ситуаций определяет тестовое покрытие.

Отказ

Отказ – это нарушение работоспособности системы; это такое изменение рабочих параметров системы (состояние), при котором снижается эффективность её функционирования ниже допустимого уровня, либо полностью прекращается выполнение функций.

Отказ может возникать вследствие внутренних изменений в системе (изменение параметров всей системы или её компонентов) или под влиянием внешней, по отношению к системе, среды. Отказ может быть внезапным или постепенным. При внезапном отказе характеристики системы изменяются скачкообразно, тогда как, постепенный отказ характеризуется медленным, поступательным изменением параметров системы, что создаёт трудности в смысле выявления его причины.

В смысле допустимости отказов, все системы можно разделить на две группы: системы, которые предполагают возможность отказа при некоторых предусмотренных условиях; и системы, к которым предъявляются высокие требования надёжности, поскольку их отказ может привести к необратимым последствиям. Например, если откажет текстовый редактор, то текст можно набрать заново, или восстановить, тогда как, если откажет система навигации самолёта, то это может привести к аварии. Отсюда возникает понятие надёжности, как свойства системы сохранять значения установленных параметров в заданных пределах, соответствующих режимам и условиям функционирования.

В рамках обеспечения надёжности применяют ряд мероприятий связанных с обеспечением отказоустойчивости системы. Отказоустойчивость определяется, как свойство системы сохранять способность корректно функционировать после отказа.

5. Тестировщик и QA инженер

Качество – это сложное и неоднозначное понятие, которое определяется многими условиями. С одной стороны, качественный продукт – это продукт, отвечающий требованиям (meeting requirements). Под требованиями обычно понимают требования заказчика, которые находят своё выражение в «спецификациях продукта». С другой стороны, качественный продукт – это тот, который оправдывает ожидания пользователей (fitness for use), обычно отражённых в «спецификациях потребителя». Но для обеспечения качества конечного продукта, необходимо, чтобы требования были измеряемыми, поскольку единственная объективная форма контроля качества – это измерение продукта на соответствие спецификациям. При этом следует заметить (как метко выразился Хемфри Уотс), что «качество – это не что-то, что возникает само по себе, и не что-то, что может быть добавлено постфактум».

В условиях высокой конкуренции, разработка программных продуктов перестаёт быть стихийным процессом и переходит на промышленный уровень, где всё подчиняется технологии производства, обеспечивающей высокий уровень качества конечного продукта. В контексте разработки программного обеспечения вводится понятие «унифицированного процесса разработки» (далее, для обозначения унифицированного процесса разработки будет употребляться термин процесс), как гаранта качества. В рамках процесса, как одно из направлений деятельности, выделается проблема управления качеством (quality management), которую условно можно разделить на два основных направления: контроль качества (quality control) и обеспечение качества (quality assurance).

Контроль качества ставит основной акцент на поиске дефектов, которые уже были внесены в продукт, а значит, носит характер реакции. Обеспечение же качества, принимает проактивный характер, имея основной функцией

предотвращение появления дефектов.

Рассматривая управление качеством как комплексный процесс, следует понимать, что мероприятия по обеспечению качества должны проводиться на всех этапах жизненного цикла программного обеспечения, а так же, в контексте процесса, выходить за рамки одного проекта, а значит проводиться постоянно.

Кто такой тестировщик?

Как уже упоминалось, в рамках процесса предусмотрено строгое разделение труда. Так, моделированием и дизайном занимается архитектор, разработкой и реализацией – программист, а тестированием – тестировщик. Исходя из вышесказанного, в обязанности тестировщика входит поиск дефектов разрабатываемого программного обеспечения, а также, отчасти, выявление их этимологии с формальным указанием условий возникновения каждой ошибки.

Вопреки распространённому заблуждению, тестировщик не занимается поиском путей устранения выявленных дефектов – это входит в обязанности разработчиков. Тестировщик лишь только обязан отправить программное обеспечение на доработку или исправление, если дефект будет иметь место.

Также, тестировщик участвует в приёмке-сдаче готового продукта, поскольку в процессе приёмки необходимо произвести проверочное тестирование, для подтверждения того, что разработанное программное обеспечение соответствует заявленным спецификациям.

Существовало мнение, что тестировщиков вербуют из числа программистов, исходя из того, что, якобы, для качественного тестирования программного обеспечения необходимы знания о том, как оно разрабатывается. Это мнение, во многом, подкреплялось популярностью объектно-ориентированного тестирования.

Действительно, неплохо, если тестировщик будет иметь некоторый опыт разработки программного обеспечения, однако – совершенно не обязательно.

Тестирование и разработка – суть два различных процесса. А тестировщик обязан «проверять» работу софта по-другому, нежели это делает программист. Иначе он рискует пропустить те же ошибки, которые были пропущены программистами. Поскольку, программисты склонны взаимодействовать с программным обеспечением так, как это было ими предусмотрено. А реальный пользователь, в противоположность программистам, желает поступать с программным обеспечением так «как ему нравиться». Эта тенденция, также усугубляется тем фактом, что созданный интерфейс, зачастую, не столь очевиден, как это кажется разработчикам. Другими словами, тестировщик должен мыслить иначе программиста.

С одной стороны тестировщик должен чувствовать пользователя, то есть, предполагать (видеть) возможные, неадекватные с точки зрения интерфейса, варианты использования программного обеспечения. С другой стороны, тестировщик должен методично и полно проверять корректность функционирования всех модулей и компонентов программы. Системно и «дотошно» продумывать и проверять все возможные тестовые ситуации.

Таким образом, разработчик по своей сути – созидатель (от англ. creative – созидательный, плодотворный), тогда как тестировщик – исследователь.

Цели и задачи тестировщика

Мы уже говорили о целях и задачах тестирования программного обеспечения, как направления деятельности в рамках процесса. В общем смысле тестировщик является лицом, реализующим эти цели и задачи.

Задача тестировщика состоит в том, чтобы поставлять разработчикам информацию о соответствии продукта заданным спецификациям. Как выразился Алексей Баранцев, главный редактор портала software-testing.ru, в статье «Основные положения тестирования: «главная деятельность тестировщиков заключается в том, что они предоставляют участникам проекта по разработке программного обеспечения отрицательную обратную связь о качестве программного продукта».

Кто такой QA инженера?

Обеспечение качества занимает в процессе не менее важное место, чем контроль. Контроль позволяет отыскивать дефекты и устранять их, но куда эффективнее не допускать появления дефектов, чем тратить время и средства на их устранение. Именно разработка и внедрение превентивных мер по недопущению появления дефектов и составляет основную задачу обеспечения качества.

В рамках процесса имеет место должность инженера по качеству или QA инженера (Quality Assurance Engineer), задачей которого является изучение процесса на предмет оптимизации с точки зрения повышения качества процесса, с одной стороны, и продукта – с другой.

Инженеры по качеству (QA инженеры) составляют группу SQE (Software Quality Engineering – Инженерия по качеству программного обеспечения). Группа SQE имеет два основных направления деятельности:

- разработка процессов (Process Engineering - PE), которая предполагает постоянную деятельность, направленную ан поиск путей модернизации процесса, с целью повышения качества производимой продукции;
- обеспечение качества (Software Quality Assurance - SQA), в основном состоящее в удостоверении того, что разработка продукта соответствует определённым на предприятии процессам (process consistency).

Можно сказать, что AQ инженер занимается обеспечением качества. И одна из его основных функций – сбор и анализ метрии процесса (формальных показателей качества процесса, примерами которых могут быть: количество дефектов, найденных после релиза; несоответствие запланированных сроков сдачи проекта реальным срокам; показатели производительности и качества труда). На основании собранных данных инженер по качеству может принимать решения по изменению характера процесса для обеспечения

качесвта.

Цели и задачи QA инженера

К функциям обеспечения качества (QA) относятся такие виды деятельности как:

- постоянное совершенствование процессов производства (process improvement), которое, как правило, выражается в предотвращении дефектов (defects prevention), носящего форму изучения дефектов с целью их предотвращения, а также опыта прошлых проектов с целью поиска путей усовершенствования процессов разработки;
- удостоверение соответствия разработки продукта определённым процессам (process consistency), другими словами, проведение аудитов, целью которых являются: определение соответствия хода проекта определённому процессу, обеспечение источником информации для усовершенствования процесса, обеспечение входных данных для проведения периодического тестирования процесса.
- Разработка процессов (process engineering) – изучение опыта организации процесса на производстве с целью его совершенствования и разработки новых процессов, обеспечивающих производство качественного продукта.

Исходя из функций обеспечения качества, инженерами по качеству строятся соответствующие этим функциям цели и задачи. Примерами QA-целей могут служить:

- разработка эффективных процессов для проектов;
- контроль и обеспечение корректного использования процессов и адаптация процессов к специфичным потребностям проектов;
- выбор соответствующей модели жизненного цикла проекта;
- выбор эффективных методов и способов тестирования;
- организация конфигурационного контроля;

- подбор и использование эффективных методов предотвращения дефектов (примерами могут служить такие методы, как начальные совещания (kick-off), совещания по вопросам контроля законченных фаз проекта (postmortems), причинно-следственный анализ (causal analysis));
- и многое другое.

6. Цели и задачи тестировщика и QA инженера

Хотя основная цель тестировщиков и QA инженеров одна – обеспечение качества конечного продукта, задачи их деятельности (как можно судить из предыдущей главы) отличаются. Но, при всех отличиях, их деятельность неразрывно связана, поскольку решения QA инженера влияют на работу тестировщика, поскольку определяют характер и форму последней, а тестировщик, в свою очередь, обеспечивает QA инженера данными, необходимыми для дальнейшего совершенствования процесса.

QA инженер и тестировщик работают на различных уровнях процесса, а также в различных зонах ответственности. И важно, чтобы каждый выполнял отведённую ему роль, поскольку качество не возможно как без эффективной организации, так и без осуществления конечного контроля.

Из сказанного ранее следует, что QA инженер отвечает за обеспечение качества, а тестировщик – за его контроль. В отдельных аспектах цели QA-инженера могут принимать характер контроля, но всегда в QA-контексте.

7. Тестирование в контексте процесса разработки программного обеспечения

Вы уже знакомы с разновидностями жизненных циклов разработки программного продукта, наверное, вы обратили внимание, что во всех циклах имеется этап, на котором выполняется тестирование части или целиком всего продукта. Рассмотрим несколько основных жизненных циклов разработки программного обеспечения, и отметим в них этапы, где производиться тестирование разрабатываемого продукта.

Каскадная модель:

- Исследование концепции
- Исследование системы
- Анализ требований
- Разработка проекта
- **Внедрение**, на этой фазе начинается процесс тестирования, разрабатываются тесты, которые тестируют части продукта.
- **Установка**, производиться тестирование готового продукта, в операционной среде заказчика.
- **Эксплуатация и поддержка**, этап, когда пользователи продукта впервые используют его по назначению, именно сейчас они могут обнаружить новые ошибки, поэтому тестировщики должны быть готовы принять отчет об ошибке, и правильно его оформить и передать разработчикам для исправления.
- **Сопровождение**, программный продукт редко остается таким, каким он был изначально сделан, как правило, постоянно к программе выходят разнообразные дополнения, патчи, которые нуждаются в тестировании.
- Вывод из эксплуатации

V-образная модель:

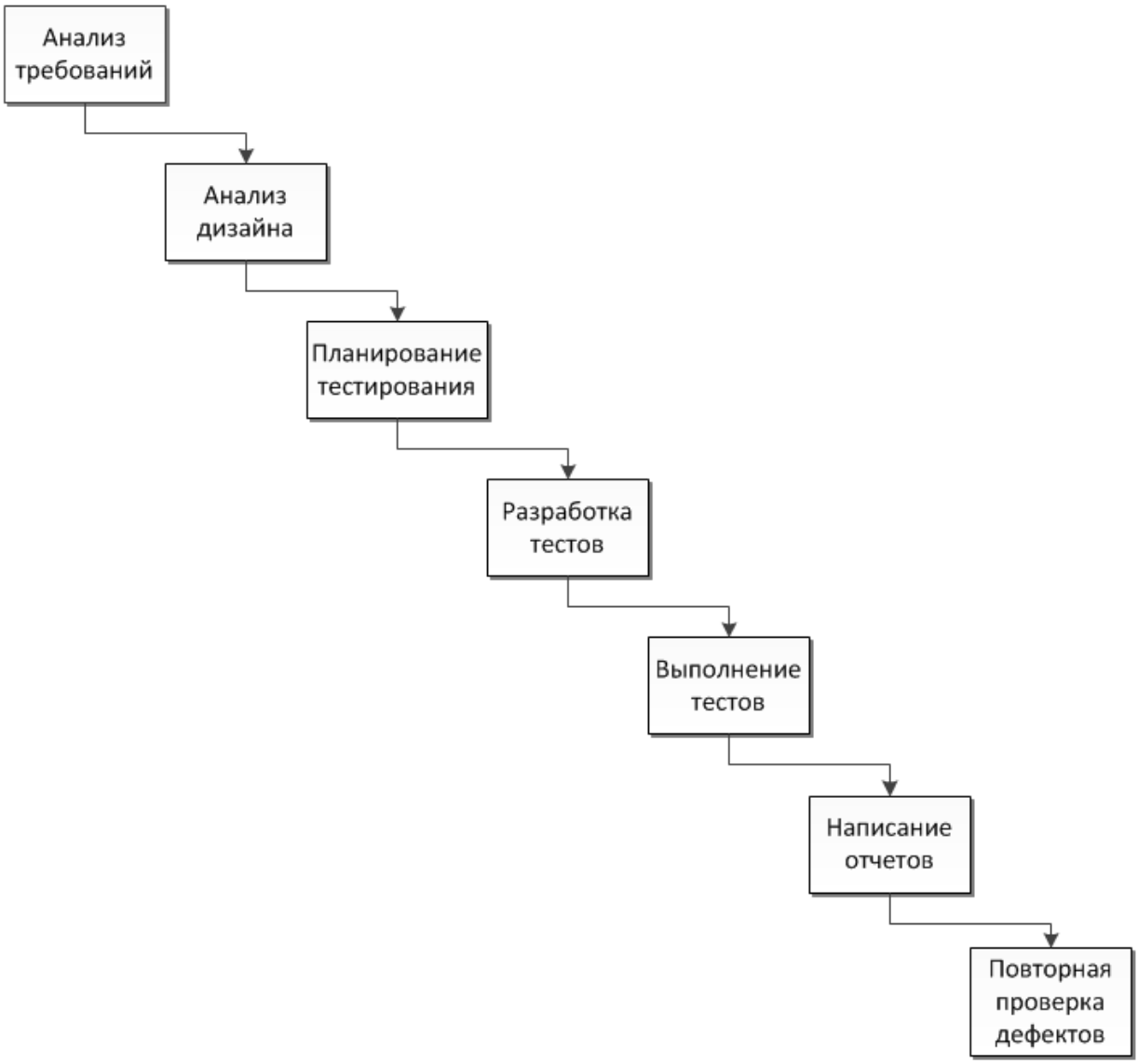
- Планирование проекта и требований
- Анализ требований
- Проектирование архитектуры проекта
- Детализированная разработка
- Разработка программного кода
- **Модульное тестирование** – тестирование небольших участков программного кода, например классов.
- **Интеграция и тестирование** – выполняется интеграционное тестирование, это тестирование так называемых функциональных точек, например: тестирование части программы, которая работает с базой данных.
- **Системное тестирование** – этап жизненного цикла проекта, когда выполняется полное тестирование готового продукта.
- **Сопровождение** – продукт тестируется в операционной среде клиента, вносятся дополнения.
- **Приемочные испытания** – функциональное тестирование, на этой фазе приложение тестируется на выполнение основных своих функций.

Из выше написанного можно сделать вывод, что тестированию программного обеспечения уделяется достаточно много ресурсов. Это обоснованно тем, что компании, которые занимаются разработкой программного обеспечения, хотят делать качественные продукты, это позволит разрабатываемым программам «держать» конкуренцию.

Жизненный цикл тестирования программного обеспечения.

Для структуризации процесса тестирования был разработан жизненный

цикл тестирования программного обеспечения.



- **Анализ требований**, для человека, занимающегося тестированием программного обеспечения, информация о том чего хочет клиент, очень важна, клиентов будет больше, если они будут получать именно то, что хотят. На фазе анализа требований тестировщик должен получить требования к тестируемому продукту и сформировать из них «матрицу требований», это позволит в будущем следить, чтобы программа оставалась такой, какой хочет видеть ее клиент.
- **Анализ дизайна проекта**, на этапе жизненного цикла, тестировщики

будут решать, какой подход к тестированию будет использоваться, решение зависит от «матрицы требований» создавшейся на прошлой фазе жизненного цикла. Например, компания разрабатывает продукт видео редактор, основными требованиями оказались скорость конвертации видео, и дружественный интерфейс. На стадии анализа дизайна проекта было принято решение разработать автоматический тест для измерения скорости конвертации видео, и найти человека, который будет смотреть за тем, чтобы интерфейс был дружественным.

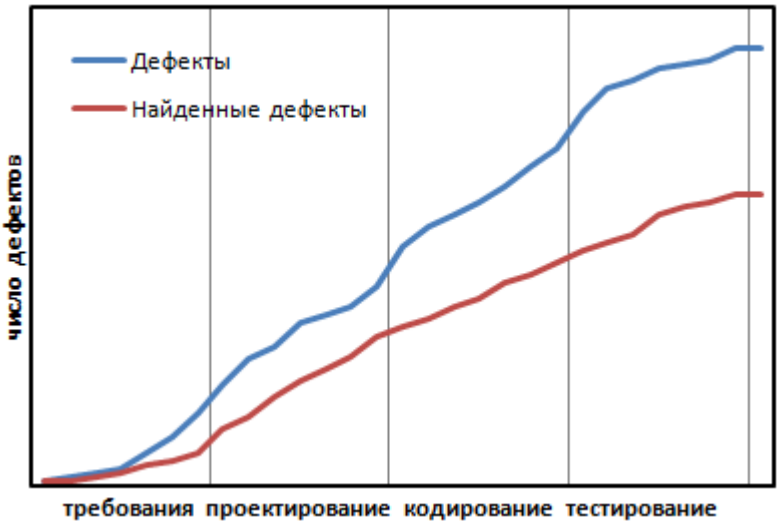
- **Планирование тестирования**, это этап, на котором производиться планирование тестов, тестировщики планируют как они будут проверять требования предъявленные к продукту. Например: для выше упомянутой программы будут подобранны тестовые данные (видео материал, и эталонное время работы с ним). Так же на этом этапе производиться распределение задач между тестировщиками.
- **Разработка тестов**, разработка тестов ведется параллельно разработке программы, как только разрабатывается часть программы, к ней разрабатывают тест, который будет производить тестирование этой части программы.
- **Выполнение тестов**, выполнение тестов производиться постоянно в процессе разработки продукта, для более раннего отслеживания дефектов.
- **Написание отчетов**, тестировщики после выполнения тестирования должны написать отчеты о найденных дефектах, и частях приложения которые работают правильно. В любом случае должна иметься документация о, всех частях продукта, не зависимо от найденных дефектов.
- **Повторная проверка дефектов**, когда отчеты были написаны и переданы разработчикам, их задача исправить найденные дефекты приложения, после исправления все приложение должно пройти повторную проверку тестировщиками.

График стоимости поиска дефекта на различных стадиях разработки проекта

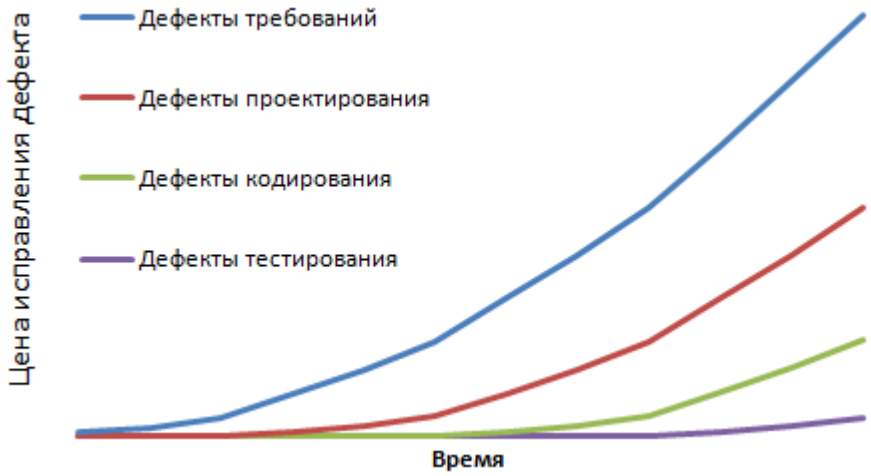
Чем дальше по жизненному циклу идет проект, тем больше ресурсов необходимо затратить на поиск дефектов в разрабатываемом продукте.

Такая ситуация возникает потому что при написании малой части продукта имеется мало дефектов, но практически все, найти невозможно. Чем больше частей, из которых состоит продукт тем больше в нем дефектов. Однако наибольшим источником дефектов, как правило, выступает «связка» между частями, например: программист разработал часть приложения, которая работает с базой данных, другой программист разработал часть программы которая занимается математическими вычислениями, в итоге может получиться ситуация когда часть с математикой предоставляет данные в не верном представлении для части работы с базой данных. Поэтому и получается что график количества найденных дефектов растет экспоненциально.

Если постоянно прилагать усилия к исправлению дефектов, немалое количество багов будет найдено, рассмотрим следующую диаграмму.



Дефекты нужно устранять по мере их поступления, так как чем дольше дефект находится в не исправленном состоянии, тем дороже потом обойдется его исправление. Например, если был найден дефект в коде одной из частей продукта, то такое исправление обойдется гораздо дешевле, чем исправление ошибки архитектуры системы, когда все программисты строили свою логику вокруг разработанной не правильно архитектуры, однако дороже всего обходятся не найденные дефекты на этапе сбора требований.



Именно по выше перечисленным причинам, компании занимающиеся разработкой программного обеспечения так скрупулезно относятся к поиску дефектов и отбирают на вакации тестировщиков только лучших кандидатов.

8. Подходы к тестированию

В контексте тестирования программного обеспечения намного более результативно будет производить поиск дефектов, которые имеют общий характер. Например, сначала проверить все «окна» приложения на «проверку граничных условий», после чего протестировать приложение на всех платформах, и так далее. Поэтому используются разные подходы к тестированию программного обеспечения, это позволяет сделать поиск дефектов более результативным.

Стоит отметить что при тестировании программного обеспечения вы не ограничиваетесь одним подходом, а одновременно используете несколько подходов.

Подходы к тестированию бывают:

- **Exploratory (ознакомительное) и Scripted (по сценарию)** – ознакомительное тестирование это тестирование приложения в свободном для тестировщика режиме. Ознакомительное тестирование обеспечит начальную информацию о том возможно ли вообще тестировать это приложение, если например приложение не запускается, то продолжать его тестирование не представляется возможным. Также ознакомительное тестирование производиться для частей приложения не затронутых в сценариях для тестирования по сценарию. Большим минусом ознакомительного тестирования является то, что тестировщик должен хорошо знать предметную область приложения. Например: тестировщик тестирует математическую часть приложения, производит операцию, но результат операции его не устраивает, ему «кажется» что он не правильный, но на самом деле он может быть правильным. В тестировании по сценарию такой ситуации не возникнет, так как сценарий обеспечивает достаточно ясными тестовыми данными. Например: при тестировании математической части приложения тестировщик будет видеть тестовые данные, информацию о том какие данные нужно передавать программе, для получения определенного

результата, который тоже описан в тестовых данных, если результат работы программы такой же какой описан в тестовых данных, значит, приложение работает правильно. Минусом тестирования по сценарию является то, что не все приложение покрывается такими сценариями.

- **Manual (ручное) и Automated (автоматическое)** - Ручное тестирование производиться человеком, а автоматическое тестирование производиться машиной. Разница между подходами очевидна, ручное тестирование обеспечивает больше тестовых отчетов. Например, человек тестируя часть приложения, может заметить, что поле ввода не достаточно большое для ввода нужного количества данных. Однако человек может допускать ошибки при тестировании. Когда компьютер тестирует приложение, не допускаются ошибки, машина тестирует приложение намного быстрее, чем человек, и денег за это не просит. Как правило, автоматическое тестирование применяется для постоянного тестирования какой-то части приложения, ведь после добавления и подгонки части программы для общей системы часть может перестать работать, и автоматическое тестирование сразу же укажет на часть, которая работает не правильно. Также автоматическое тестирование применяется, когда нужно тестировать большой диапазон данных по одному и тому же алгоритму. Автоматическое тестирование незаменимо, когда речь идет о проверках на «прочность», например, если вы разрабатываете сервер для онлайн игры и вам нужно проверить, как сервер будет работать с 1000-й подключенных клиентов, данный тест возможен, только если для этого подключить весь офис компании, или запустить автоматический тест который выполнит 1000 подключений автоматически. Во всех случаях, где невозможно производить автоматическое тестирование, приходится использовать ручной труд тестировщиков.
- **Black Box(черный ящик) и White Box (белый ящик)** – тестирование черного и белого ящика подразумевает, будет или не будет тестировщик

иметь доступ к исходному коду программы. Когда производиться тестирование «черного ящика» подразумевается то, что тестировщик не имеет доступа к исходному коду, а тестирование «белого ящика» позволяет тестировщику использовать исходный код приложения, на практике иногда используется термин тестирование прозрачного ящика, означает что, будет производиться тестирование с использованием исходного кода приложения. Белый и прозрачный ящик это одно и то же. На первый взгляд может показаться нет смысла в тестировании черного ящика, если есть возможность тестировать «белый ящик» однако это не так. Тестирование черного ящика происходит тогда необходимо подтвердить, что приложение отвечает заранее предъявленным требованиям, соответствие документации. Плюсом тестирования черного ящика является то, что большая часть приложения за короткое время будет оттестирована, вить не надо, углубляться в код, достаточно просто пользоваться интерфейсом, так же как заказчик. Тестирование белого ящика подразумевает активное использование исходного кода, например создание unit-test когда пишется специальное приложение которое использует часть исходного кода разрабатываемого продукта, для того чтобы в автоматическом режиме производить его тестирование.

- **Positive (Позитивное) и Negative (негативное)** – Данные подходы к тестированию отличаются тем что при позитивном тестировании тестировщик играет роль «правильного» пользователя, который читает документацию, и исходя из указаний в документации пользуется программой. При позитивном тестировании тестировщик работает с программой, так как она ему «говорит», например, если приложение требует от пользователя пароль, размером от 4 до 8 символов, то тестировщик введет пароль, который будет состоять из необходимого программе количества символов. Негативное тестирование заставляет тестировщика тестировать программу наоборот, тестировщик должен играть роль хакера, который пытается поломать программу, стоит

приложить все возможные усилия для проверки программы в самых не благоприятных условиях, начиная от неправильного использования заканчивая целенаправленным взломом. Тестировщики часто недовольны тем что существует такое разделение и им приходиться играть роли а не заниматься результативным тестированием. Однако стоит разобраться в плюсах позитивного и негативного тестирования, при позитивном тестировании приложение тестируется намного быстрее, и исправляются самые важные дефекты, в итоге получается, что вся программа тестируется быстрее, но покрытие такого тестирования предусматривает только использование программы «по документации» а большинство пользователей не читают документацию к программе. Для «плохих» пользователей предусмотрено негативное тестирование, для проверки на дефекты всего приложения даже если ему будут преподносить неправильные данные, например, приложение требует ввести путь к файлу на диске а пользователь вводит что-то типа «Hello World», при не правильной обработке таких данных пользователь увидит как программа принудительно завершает свою работу, а при правильной обработке таких данных он увидит информацию о том какие данные необходимо передавать в это поле ввода. Как правило, компании, которые разрабатывают программные продукты, в первую очередь производят позитивное тестирование, и только по завершении позитивного тестирования производят негативное, это связанно с тем, что программа как минимум должна адекватно работать для «хороших» пользователей, а как максимум отлично работать для «плохих» пользователей. Так как часто на негативное тестирование не хватает времени быстро производиться позитивное тестирование для обеспечения минимального качества программы, ведь лучше минимум, чем ничего?

9. Виды ошибок

Дефекты приложения делятся на типы, это сделано для того чтоб их было проще искать, ведь когда конкретно знаешь что искать поиск идет намного более результативней нежели искать просто что-то абстрактное. Часто в компаниях разрабатывающих программные продукты тестирование производиться по видам ошибок, другими словами сначала ищутся дефекты, связанные с пользовательским интерфейсом, потом с тестированием на разных платформах и так далее.

Ошибки функциональности

такой вид ошибок наиболее опасен, так как ошибка функциональности это когда программа не выполняет какую либо функцию, либо выполняет ее не правильно. Например, представьте, компания разрабатывает программное обеспечение для бортового компьютера автомобиля, к продукту предъявлено требование: реализовать систему «круиз контроль» при активации этой функции автомобиль должен держать скорость, которая была перед активацией этого режима. После разработки выяснилось что разработанный «круиз контроль» не держит нужную скорость на подъеме вверх на холм, а пользователи ждут что автомобиль будет ехать с одной и той же скоростью в любых условиях, такой дефект был бы ошибкой функциональности.

Ошибки пользовательского интерфейса

Поиску такого вида дефектов выделяется много ресурсов, так как пользовательский интерфейс это то, что видит пользователь. Очень важно, чтобы программой было удобно пользоваться, чтобы она радовала глаз, и вообще производила хорошее впечатление.

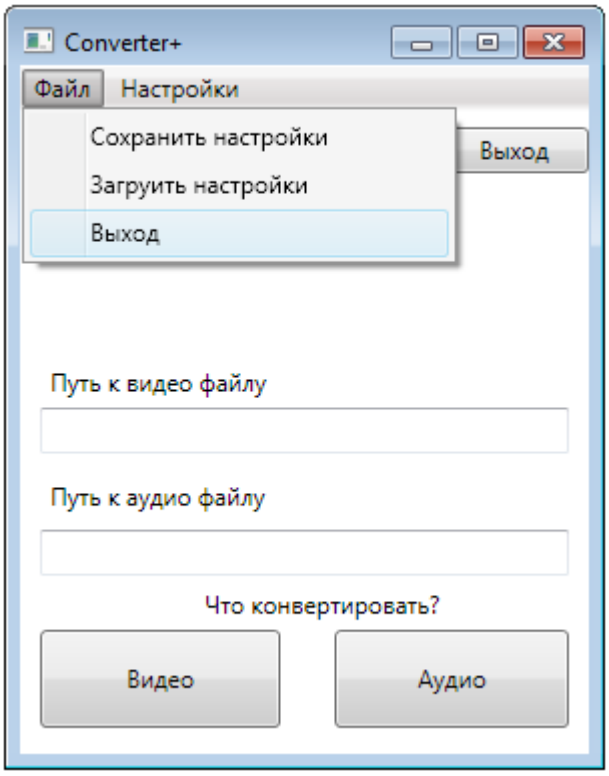
Взаимодействие программы с пользователем – пользовательский интерфейс это, прежде всего способ взаимодействия программы с пользователем, а клиент требует, чтобы программой было удобно пользоваться, задача разработчиков сделать интерфейс «дружественным» (user-friendly interface), а тестировщики должны проверить действительно ли

дружественный этот интерфейс.

Чуть позже я напишу, каким должен быть интерфейс, чтобы стать дружественным, а пока давайте рассмотрим остальные виды ошибок пользовательского интерфейса.

Организация интерфейса – стоит следить за тем, чтобы интерфейс логически был разделен на группы, например элементы управления для конвертации видео в программе должны находиться в одной части интерфейса, а элементы управления, отвечающие за конвертацию аудио в другой части интерфейса, должно быть привычное логическое разделение. Также нужно следить за тем, чтобы в интерфейс не был построен так, что сможет ввести пользователя в замешательство, чтобы не было дублирующих команд, или очень похожих.

Рассмотрим пример плохо организованного интерфейса:



Как видите, первое, что вводит в заблуждение это 3 кнопки для выключения программы, (пункт меню, кнопка на клиентской области, и стандартная кнопка windows). Пользователь не знает, чем отличаются функции этих трех кнопок, и это раздражает.

Также остальная организация интерфейса является не приемлемой, так как элементы управления разных функциональных точек приложения находятся в куче. Данный интерфейс является плохо организованным, и это считается дефектом, который необходимо исправить. Попробуйте представить, как надо переделать этот интерфейс, чтобы он стал хорошо организованным.

Пропущенные команды - это дефект интерфейса, когда в приложении явно не хватает какой-то функции, или выполняется эта функция странным для пользователя способом, или просто не понятно как пользоваться этой функцией. При поиске таких дефектов нужно иметь опыт работы с большим количеством разных приложений, и знать, как обычно выполняются разные действия. Например, обычно в текстовом редакторе для того чтобы сохранить файл нужно нажать на кнопку с изображением флоппи диска, обычно она находиться ближе к началу меню, в левом верхнем углу окна программы. Если вы увидите текстовый редактор без таких кнопок, к которым все привыкли, можно считать что в этой программе имеется дефект пропущенных команд, хотя часто отсутствие таких кнопок это задумка разработчиков, ведь никто не говорит о том, что клик на кнопку с флоппи диском это самый удобный способ сохранения.

Производительность – очень плохо, когда пользователь как программа с задержкой реагирует на его действия. Пользовательский интерфейс не сможет заставить работать программу быстрее, но очень легко заставить интерфейс показывать, что программа работает, занимается делом так сказать. Пользователи любят смотреть, как программа работает, но видят они это посредством пользовательского интерфейса. Вы пишете программу, которая конвертирует видео? Добавьте ProgressBar для отображения процесса конвертации, и будет видно, что программа сейчас занимается делом, она конвертирует видео файл и позволяет пользователю ознакомиться с информацией о том, сколько уже сконвертировали, и сколько еще осталось. Хочу отметить, что отображение информации о ходе выполнения задачи может замедлить выполнение поставленной задачи. Однако когда пользователь видит работу программы он готов ждать часами, а если будет видеть окно программы, которое не реагирует на его действия, пользовательское терпение очень быстро закончиться и будет вызван «диспетчер задач» для принудительного завершения работы программы. Поэтому программный интерфейс, который н показывает работу программы, считается дефектным.

Выходные данные – выходными данными программы называют все результаты работы программы, будь то информация, экспортированная в PDF файл, или просто вывод в label на клиентскую область окна. Дефектом выходных данных является любая не точность, не разборчивость, неудобочитаемость таких данных. Рассмотрим пример интерфейса, где имеется дефект выходных данных.

Как видите, в данном случае самая важная информация спрятана за кнопкой. Также дефектом выходных данных в данном приложении можно считать то, что не достаточно ясно указаны единицы измерения величин отображенных на экране мобильного устройства.

Обработка ошибок – хорошо, когда в программе не возникает ошибок или когда они возникают, программа сама в силах их исправить, но всех ошибок не предусмотреть, поэтому очень важно показывать подробный отчет об ошибке, желательно выводить на экран информацию о том, как исправить возникшую ошибку. Например, ошибка, не найден файл базы данных, отчет об ошибке должен выглядеть приблизительно следующим образом, «Файл с базой данных не был найден, по пути “C:\DataBase\Hello.mdf”, укажите новый путь к базе

данных, это можно сделать в меню настроек, а если база данных была утеряна, вы можете скачать ее у нас на сайте».

На этом стандартные ошибки пользовательского интерфейса заканчиваются, но не забывайте, возможно, у вас получиться найти дефекты интерфейса которые не попадают в эти категории.

Ошибки граничных условий

Ошибками граничных условий считаются части приложения, где нет проверки на формат входящих данных, например, приложение требует от пользователя числовое значение, а пользователь вводит текст hello world если приложение попробует интерпретировать введенный тест как число то это будет считаться ошибкой граничных условий. Обязательно любые входные данные должны проверятся множеством условий, для того чтобы программа не потеряла работоспособность. Если возле поля ввода написано что в него можно от 4 до 10 символов, то должен иметься код, который проверит, действительно ли пользователь ввел нужное количество символов. Раньше 60 процентов исходного кода приложения составляли различные проверки, сейчас все изменилось, множество элементов управления которые не позволяют пользователю «хулиганить».

Ошибки вычислений

Ошибками вычислений называют ошибки, когда программа выполняет неверные математические действия и в результате этого пользователь видит не правильный результат. Например, приложение выполняет подсчет зарплаты работникам и изымает налог из жалования, очень плохо, если налог вместо 3% будет 30%, поэтому очень важно находить и исправлять такие дефекты.

Ошибки управления потоком

Под эту категорию ошибок, попадают все дефекты, связанные с не правильной работой потоков. Это может быть вызвано не правильной синхронизацией, или в силу того что важному потоку был присвоен низкий приоритет а потоку которые выполняет второстепенные задачи присвоили приоритет реального времени.

Ошибки передачи или интерпретации данных

Подключение к другим компьютерам позволяет вашему приложению взаимодействовать с ними, с чем повышается полезность приложений. Однако это также и является хорошим источником дефектов для вашего приложения. Ошибкой передачи или интерпретации данных может быть не правильный формат передаваемых через сеть данных, или ошибка чтения принятых данных, также ошибкой передачи данных называют ситуацию при, когда диалоге между компьютерами наступает момент, когда оба компьютера отправляют данные (что приводит к их потере). Программные продукты должны быть готовы к тому, что через сетевое подключение придут данные, которые будет невозможно разобрать. Для проверки таких дефектов существуют различные программные продукты, которые позволяют «перехватывать» данные в сети, и позже отправлять в измененном виде.

Ошибки, связанные с перегрузками и аппаратным обеспечением

Аппаратное обеспечение это фундамент для работы приложений, к сожалению, фундамент не всегда качественный, а ваше приложение должно быть готово к сюрпризам, например к внезапной перезагрузке компьютера, или к тому, что компьютер будет так нагружен задачами, что не сможет выполнить нужную задачу за отведенный отрезок времени. Приложение должно время от времени создавать резервные копии важных данных. Если речь идет о передаче данных по сети, приложение должно быть готово к обрывам сети. Если тестируется веб сайт, тестировщик должен узнать, сколько времени будет потрачено на загрузку сайта с медленным подключением к интернету, возможно понадобиться заменить тяжелые элементы более легкими, или создать «легкую» версию сайта.

Ошибки, связанные с контролем версий

Сейчас операционная среда очень развита, программисты пишут приложения, которые используют массу внешних библиотек. Например, когда вы программируете приложение на языке C#, оно использует .net framework который тоже использует операционную систему windows для работы.

Ошибками контроля версий являются ошибки связанные с версиями внешних библиотек. Например, было написано игровое приложение которое использует DirecX для вывода графики на экран, у программистов все работает а у клиентов нет, все оказалось потому что у разработчиков стояла более поздняя версия, в итоге это привело к тому что компании которые занимаются разработкой программных продуктов, составляют «инсталляторы» в которые помимо собственного продукта вкладывают еще и библиотеки которые необходимы для работы приложения. А разработчики сторонних библиотек в свою очередь со всех сил стараются сделать так чтобы приложение, написанное на .net framework 3,0 могло нормально работать в среде, где установлен .net framework 4,0.

Ошибки документации

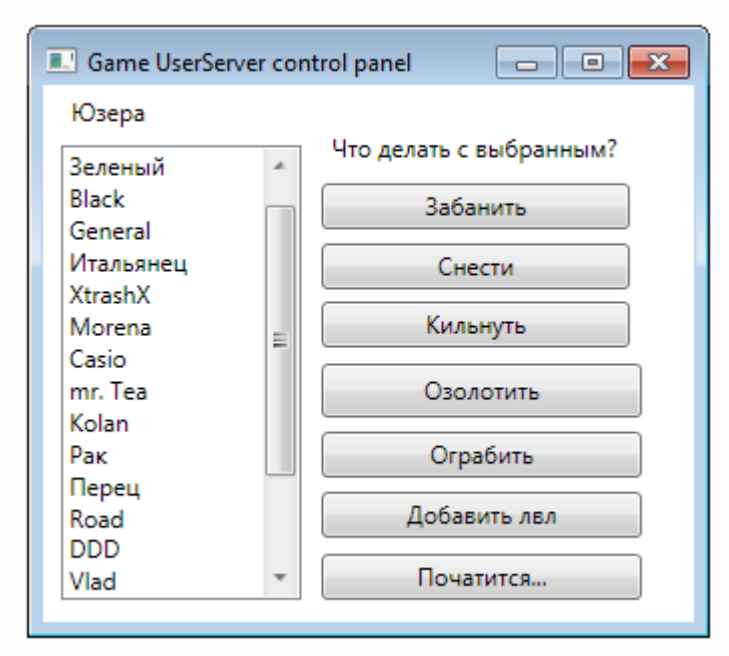
В документации к приложению должна иметься разносторонняя информация о том, что это за приложение, как следует использовать его, как правильно инсталлировать программу, как ее удалить, как активировать и многое другое.
Иногда случается так, что документация пишется с ошибками, это могут быть просто грамматические ошибки, или важные ошибки, когда неверно указаны системные требования к программе, или неправильно написана инструкция использования. Кроме того документация должна быть хорошо оформлена, удобная для чтения, и содержать всю необходимую информацию.

Ошибки тестирования

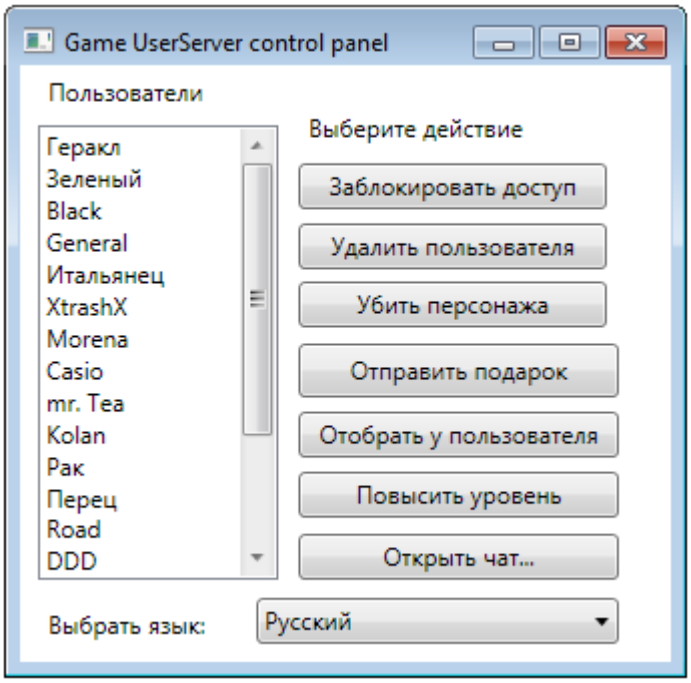
Ошибки тестирования возникают, когда тестировщики допускают ошибки в своей работе, после рабочая часть программы отправляется на исправление, как будто она работает не правильно. После такой ситуации программисты стараются исправить то, что работает правильно и делают, что часть программы начинает работать не правильно, после опять производиться тестирование и, оказывается, была допущена ошибка тестирования. В итоге все теряют много времени.

Дружественный пользовательский интерфейс должен быть

Доступность, под доступностью интерфейса подразумевается, что все элементы управления будут понятны максимальному количеству пользователей. Желательно не использовать профессиональный сленг в интерфейсе и документации к приложению. Давайте рассмотрим пример приложения, которое является не доступным для многих пользователей. Пользовательский интерфейс составлен для сервера онлайн игры.



Как вы наверное заметили интерфейс является не доступным для большинства пользователей, хотя возможно вам как IT-шиннику с большим опытом общения с себе подобными такой интерфейс будет казаться вполне доступным. Но на самом деле это не так.



Далее представлен тот же интерфейс, Однако этот является доступнее ранее показанного. Сленг заменен обычными всем понятными словами, также была добавлена часть интерфейса, где пользователь может выбрать язык, на котором будет, отображается интерфейс.

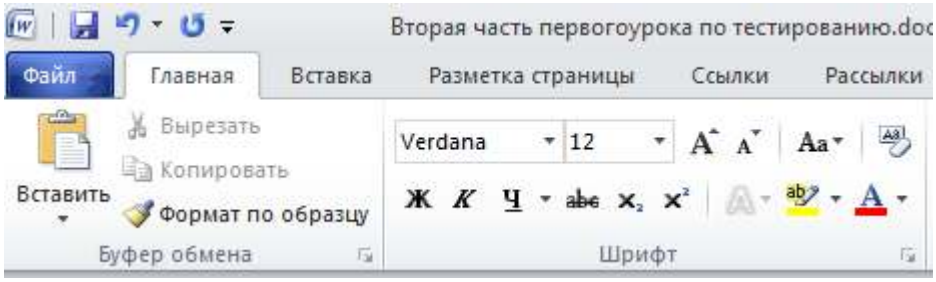
Минимализм, в интерфейсе не должно быть ничего лишнего, отличным примером может служить регулятор громкости в windows. Однако стоит отметить что часто пользователи хотят более тонких настроек, это как правило достигается созданием кнопки которая активирует «профессиональный» режим, в котором можно произвести тонкую настройку.

Еще одним хорошим примером может служить приложение для проигрывания мультимедиа Winamp:

Минимальный режим:

Максимальный режим:

Уверенность, Очень важно чтобы все элементы управления просто «кричали» о том что они делают, в этом деле важно знать «штуки» к которым привыкли пользователи, например, все пользователи привыкли что кнопка на которой изображена флоппи дискета выполняет сохранение документа. Хорошим примером такого интерфейса является приложение из пакета Microsoft Office под называнием Microsoft Word.



Все элементы привычны, большинство пользователей понимают, что означает каждая из кнопок.

Отзывчивость, помните, выше я писал про ошибки производительности, сейчас еще раз напомню вам, что очень важно, чтобы пользователь видел, как работает программа, если она работает долго, однако если ваше приложение работает быстро и не выполняет длительных операций не стоит засорять интерфейс полосками загрузки.

Соответствие контексту, у приложения почти всегда есть основная цель, а

есть дополнительные возможности, необходимо чтобы элементы управления для решения основных задач были хорошо видны, а дополнительные функции можно спрятать. Например, вы разрабатываете интерфейс приложения, которое позволит пользователю работать с изображениями, и вы решили дополнить свое творение возможностью слушать онлайн радио, не стоит выводить элементы управления радио на основное окно, лучше сделать пункт меню, в котором будет запускаться окно, и в нем может играть музыка.

Привлекательность, очень спорный пункт дружественного интерфейса, многие скажут, «на вкус или цвет все фломастеры разные» имеется виду, что всем ваш самый красивый интерфейс все равно не понравиться. Однако есть такое понятие как целевая аудитория, этим словом, можно назвать категорию предполагаемых пользователей. Например, вы разрабатываете дизайн сайта для афиши городских готических концертов, значит, интерфейс сайта должен быть в готическом стиле. Хотя когда речь идет об офисных работниках лучше оставить стиль стандартным, так как операционная система Windows сама украшает интерфейс исходя из настроек системы. Ну а если вы разрабатываете сайт, для максимального количества пользователей, используйте мягкие цвета, так называемые «постельные тона».

Эффективность, под эффективностью подразумевается то, что пользователю для решения задачи необходимо выполнить минимальное количество действий. Например, большинство программ для конвертации видео файлов умеют сохранять настройки, для того чтобы пользователь мог единожды настроить приложение и потом выполнять конвертирование в несколько кликов мышью.

Снисходительность, все вы, наверное, когда-то заполняли на сайте регистрационную форму, вводили логин, пароль, еще раз пароль, почту, имя фамилию, и много другого, и после нажатия кнопки оправить у вас очищались все поля, и сверху красовалась надпись, пароли не совпадают. Относитесь к пользователям с уважением, предусматривайте их ошибки, создавайте резервные копии данных, предотвращайте случайное удаление данных с помощью предупреждений типа «Вы действительно хотите удалить файл?»