Экзамен представляет собой тест, готовиться нужно по этим темам:
1. V&V. Валидация(аттестация) и верификация.
2. Общие вопросы верификации ПО. Цели и задачи верификации ПО.
3. Статические и динамические методы верификации ПО.
4. Факторы и атрибуты внешнего и внутреннего качества ПО.
5. Виды и методы тестирования. Смоук-тестирование, регрессионное тестирование, тестирование белого и черного ящиков.
6. Тестирование нефункциональных требований.
7. Взаимосвязь разработки и тестирования. V-модель разработки ПО.
8. Уровни тестирования. Модульное (unit), интеграционное (integration), системное (system), приемочное (acceptance) тестирование.
9. Техники тест дизайна. Разбиение на классы эквивалентности и тестирование граничных значений.
10. Понятие дефекта. Основные определения и классификация дефектов. Описание дефектов.
11. Атрибуты дефектов. Приоритет(priority) и серьезность(severity) дефектов.
12. Определение серьезности дефекта по его описанию — практическое задание.
13. Версионирование ПО на разных стадиях разработки.
14. Инструментальные средства поддержки тестирования. Системы отслеживания ошибок (Bug Tracking Systems).
15. Артефакты разработки ПО, относящиеся к тестированию. Тест-кейсы (test cases).
16. Артефакты разработки ПО, относящиеся к тестированию. План тестирования (test plan).

1. Валидация (аттестация) и верификация (ТРО_01)

Тестирование (верификация, валидация) – процесс выявления фактов расхождений с требованиями (ошибок).

Отладка (debug, debugging) – процесс поиска, локализации и исправления ошибок в программе [IEEE Std.610-12.1990]

Как правило, на ***фазе тестирования*** осуществляется и исправление идентифицированных ошибок, включающее:

- локализацию ошибок
- нахождение причин ошибок
- корректировку программы.

Судить о правильности результатов выполнения программы можно только сравнивая спецификацию функции с результатами ее вычисления.

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

Определения тестирования по стандарту

- Процесс *выполнения* ПО системы или компонента при заданных условиях с анализом или записью результатов и оценкой некоторых свойств тестируемого объекта.
- Процесс *анализа* ПО с целью фиксации различий между существующим состоянием ПО и требуемым (что свидетельствует о проявлении ошибки) и оценки свойств тестируемого ПО.
- Контролируемое выполнение программы на конечном множестве тестовых данных и анализ результатов этого выполнения для поиска ошибок [IEEE Std 829-1983].

2. Общие вопросы по верификации ПО. Цели и задачи верификации ПО (ТРО_01, wiki)

Методики повышения качества ПО

- Контроль качества – планомерная и систематичная программа действий, призванная гарантировать, что система обладает желательными характеристиками
- Явное определение целевых характеристик (внутренних и внешних) – эффективная методика

- Разработка стратегии тестирования. Выполнить задачи оценки и повышения качества только путем тестирования невозможно.
- Неформальные и формальные технические обзоры
- инспекция
- обзор
- аудит
- Контроль изменений
- Оценка результатов выполнения плана контроля качества
- Прототипирование

Цели тестирования

- продемонстрировать разработчикам и заказчикам, что программа соответствует требованиям;
- выявить ситуации, в которых поведение программы является неправильным, нежелательным или не соответствующим спецификации

3. Статические и динамические методы верификации ПО (ТРО_01)

Статическое тестирование выявляет неверные конструкции или неверные отношения объектов программы (ошибки формального задания) формальными методами анализа без выполнения тестируемой программы:

- С помощью специальных инструментов контроля кода
- Обзоры (Reviews)
- Инспекции (Inspections)
- Сквозные просмотры (Walkthroughs)
- Аудиты (Audits)
- Тестирование требований, спецификаций, документации.

Динамическое тестирование осуществляет выявление ошибок на выполняющейся программе.

Тестирование заканчивается, когда выполнилось или "прошло" (pass) успешно достаточное количество тестов в соответствии с выбранным критерием тестирования.

4. Факторы и атрибуты внешнего и внутреннего качества ПО (ТРО_01)

Критерии качества ПО

Качество программного продукта характеризуется набором свойств, определяющих, насколько продукт "хорош" с точки зрения заинтересованных сторон, таких как заказчик продукта, спонсор, конечный пользователь, разработчики и тестировщики продукта, инженеры поддержки, сотрудники отделов маркетинга, обучения и продаж. Каждый из участников может иметь различное представление о продукте и о том, насколько он хорош или плох, то есть о том, насколько высоко качество продукта.

Таким образом, постановка задачи обеспечения *качества продукта* выливается в задачу определения заинтересованных лиц, их критериев качества и затем нахождения оптимального решения, удовлетворяющего этим критериям. *Тестирование* является одним из наиболее устоявшихся способов обеспечения качества разработки программного обеспечения и входит в набор эффективных средств современной системы обеспечения *качества программного продукта*.

Внешние характеристики

- корректность
  - наличие/отсутствие дефектов в спецификации, проекте и реализации

- практичность
  - легкость изучения и использования
- эффективность
  - степень использования системных ресурсов
- надежность
  - способность системы выполнять необходимые функции; интервал между отказами
- целостность
  - способность предотвращать неавторизованный или некорректный доступ
- адаптируемость
  - возможность использования в других областях и средах
- правильность
  - степень безошибочности данных, выдаваемых системой
- живучесть
  - способность продолжать работу при недопустимых данных или в напряженных условиях

Внутренние характеристики

- удобство сопровождения
- тестируемость
- удобочитаемость
- гибкость
- портируемость
- возможность повторного использования
- понятность

5. Виды и методы тестирования. Смоук-тестирование, регрессионное тестирование, тестирование белого и черного ящиков (TPO_06, TPO_03, wiki)

Smoke Test (англ. *Smoke testing*, дымовое тестирование) в тестировании программного обеспечения означает минимальный набор тестов на явные ошибки. Дымовой тест обычно выполняется самим программистом; не проходящую этот тест программу не имеет смысла отдавать на более глубокое тестирование.

Примеры:

- Ошибки инсталляции: если программный продукт не устанавливается, его тестирование, скорее всего, окажется невозможным.
- Ошибки при соединении с базой данных, актуально для архитектуры клиент-сервер.

Регрессионное тестирование

Регрессионное тестирование - цикл тестирования, который производится при внесении изменений на фазе системного тестирования или сопровождения продукта.

Главная проблема регрессионного тестирования - выбор между полным и частичным перетестированием и пополнение тестовых наборов. При частичном перетестировании контролируются только те части проекта, которые связаны с измененными компонентами.

«Белый ящик»

- Модель программы в виде "белого ящика" предполагает знание исходного текста программы или спецификации программы в виде потокового графа управления.

- Структурная информация понятна и доступна разработчикам подсистем и модулей приложения, поэтому данный вид тестирования часто используется на этапах модульного и интеграционного тестирования (Unit testing, Integration testing).
- Структурные критерии тестирования базируются на основных элементах управляющего графа программы - операторах, ветвях и путях.

При тестировании белого ящика (англ. *white-box testing*, также говорят — *прозрачного ящика*), разработчик теста имеет доступ к исходному коду программ и может писать код, который связан с библиотеками тестируемого ПО. Это типично для юнит-тестирования (англ. *unit testing*), при котором тестируются только отдельные части системы. Оно обеспечивает то, что компоненты конструкции — работоспособны и устойчивы, до определённой степени. При тестировании белого ящика используются метрики покрытия кода или мутационное тестирование.

«Черный ящик»

При тестировании чёрного ящика, тестировщик имеет доступ к ПО только через те же интерфейсы, что и заказчик или пользователь, либо через внешние интерфейсы, позволяющие другому компьютеру либо другому процессу подключиться к системе для тестирования. Например, тестирующий модуль может виртуально нажимать клавиши или кнопки мыши в тестируемой программе с помощью механизма взаимодействия процессов, с уверенностью в том, все ли идёт правильно, что эти события вызывают тот же отклик, что и реальные нажатия клавиш и кнопок мыши. Как правило, тестирование чёрного ящика ведётся с использованием спецификаций или иных документов, описывающих требования к системе. Как правило, в данном виде тестирования критерий покрытияскладывается из покрытия структуры входных данных, покрытия требований и покрытия модели (в тестировании на основе моделей).

6. Тестирование нефункциональных требований (http://www.protesting.ru/testing/testtypes.html)

Нефункциональное тестирование описывает тесты, необходимые для определения характеристик программного обеспечения, которые могут быть измерены различными величинами. В целом, это тестирование того, "Как" система работает. Далее перечислены основные виды нефункциональных тестов:

- **Все виды тестирования производительности**:
  - нагрузочное тестирование (Performance and Load Testing)
  - стрессовое тестирование (Stress Testing)
  - тестирование стабильности или надежности (Stability / Reliability Testing)
  - объемное тестирование (Volume Testing)
- **Тестирование установки (Installation testing)**
- **Тестирование удобства пользования (Usability Testing)**
- **Тестирование на отказ и восстановление (Failover and Recovery Testing)**
- **Конфигурационное тестирование (Configuration Testing)**

Нагрузочное тестирование или тестирование производительности

Нагрузочное тестирование или **тестирование производительности** - это автоматизированное тестирование, имитирующее работу определенного количества бизнес пользователей на каком-либо общем (разделяемом ими) ресурсе.

Основные виды тестирования производительности

Рассмотрим основные виды нагрузочного тестирования, также задачи стоящие перед ними.

Тестирование производительности (**Performance testing**)

Задачей тестирования производительности является определение масштабируемости приложения под нагрузкой, при этом происходит:

- измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций
- определение количества пользователей, одновременно работающих с приложением
- определение границ приемлемой производительности при увеличении нагрузки (при увеличении интенсивности выполнения этих операций)
- исследование производительности на высоких, предельных, стрессовых нагрузках

Стрессовое тестирование (**Stress Testing**)

Стрессовое тестирование позволяет проверить насколько приложение и система в целом работоспособны в условиях стресса и также оценить способность системы к регенерации, т.е. к возвращению к нормальному состоянию после прекращения воздействия стресса. Стрессом в данном контексте может быть повышение интенсивности выполнения операций до очень высоких значений или аварийное изменение конфигурации сервера. Также одной из задач при стрессовом тестировании может быть оценка деградации производительности, таким образом цели стрессового тестирования могут пересекаться с целями тестирования производительности.

Объемное тестирование (**Volume Testing**)

Задачей объемного тестирования является получение оценки производительности при увеличении объемов данных в базе данных приложения, при этом происходит:

- измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций
- может производиться определение количества пользователей, одновременно работающих с приложением

Тестирование стабильности или надежности (**Stability / Reliability Testing**)

Задачей тестирования стабильности (надежности) является проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки. Время выполнения операций может играть в данном виде тестирования второстепенную роль. При этом на первое место выходит отсутствие **утечек памяти**, перезапусков серверов под нагрузкой и другие аспекты влияющие именно на стабильность работы.

Тестирование Установки или Installation Testing

Тестирование установки направленно на проверку успешной инсталляции и настройки, а также обновления или удаления программного обеспечения.

Тестирование удобства пользования или Usability Testing

Тестирование удобства пользования - это метод тестирования, направленный на установление степени удобства использования, обучаемости, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий. [**ISO 9126**]

Тестирование удобства пользования дает оценку уровня удобства использования приложения по следующим пунктам:

- **производительность, эффективность** (**efficiency**) - сколько времени и шагов понадобится пользователю для завершения основных задач приложения, например, размещение новости, регистрации, покупка и т.д.? (*меньше - лучше*)
- **правильность** (**accuracy**) - сколько ошибок сделал пользователь во время работы с приложением? (*меньше - лучше*)
- **активизация в памяти** (**recall**) – как много пользователь помнит о работе приложения после приостановки работы с ним на длительный период времени? (*повторное выполнение операций после перерыва должно проходить быстрее чем у нового пользователя*)
- **эмоциональная реакция** (**emotional response**) – как пользователь себя чувствует после завершения задачи - растерян, испытал стресс? Порекомендует ли пользователь систему своим друзьям? (*положительная реакция - лучше*)

Тестирование на отказ и восстановление (Failover and Recovery Testing)

Тестирование на отказ и восстановление (**Failover and Recovery Testing**) проверяет тестируемый продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками программного обеспечения, отказами оборудования или проблемами связи (например, отказ сети). **Целью**данного **вида тестирования** является проверка систем восстановления (или дублирующих основной функционал систем), которые, в случае возникновения сбоев, обеспечат сохранность и целостность данных тестируемого продукта.

Конфигурационное тестирование или Configuration Testing

Конфигурационное тестирование (**Configuration Testing**) — специальный вид тестирования, направленный на проверку работы программного обеспечения при различных конфигурациях системы (заявленных платформах, поддерживаемых драйверах, при различных конфигурациях компьютеров и т.д.)

7. Взаимосвязь разработки и тестирования. V-модель разработки ПО (ТРО_02, wiki)

Каскадная модель (англ. *waterfall model*) — модель процесса разработки программного обеспечения, в которой процесс разработки выглядит как поток, последовательно проходящий фазы анализа требований, проектирования, реализации, тестирования, интеграции и поддержки. В качестве источника названия часто указывают статью, опубликованную У. У. Ройсом (*W. W. Royce*) в 1970 году.

8. Уровни тестирования. Модульное (unit), интеграционное (integration), системное (system), приемочное (acceptance) тестирование (ТРО_03)

Модульное тестирование (Unit-testing)

- ***Модульное тестирование*** - это тестирование программы на уровне отдельно взятых модулей, функций или классов.
- Цель модульного тестирования состоит в выявлении локализованных в модуле ошибок в реализации алгоритмов, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования.
- Модульное тестирование чаще всего проводится по принципу "белого ящика".
- Модульное тестирование обычно подразумевает создание вокруг каждого модуля определенной среды
- На уровне модульного тестирования проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов.
- Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и т.п. обычно выявляются на более поздних стадиях тестирования.
- (белый и черный ящик)

Интеграционное тестирование

- ***Интеграционное тестирование*** (тестирование сборки) - тестирование части системы, состоящей из двух и более модулей.
- Основная задача - поиск дефектов, связанных с ошибками в реализации и интерпретации взаимодействия между модулями.
- Так же, как и модульное тестирование, оперирует интерфейсами модулей и подсистем и требует создания тестового окружения
- Основная разница между модульным и интеграционным тестированием состоит в типах обнаруживаемых дефектов. В частности, на уровне интеграционного тестирования часто применяются методы, связанные с покрытием интерфейсов
- Интеграционное тестирование использует модель "белого ящика" на модульном уровне.

Методы сборки модулей

- **Монолитный**, характеризующийся одновременным объединением всех модулей в тестируемый комплекс.

  Для замены неразработанных к моменту тестирования модулей необходимо дополнительно разрабатывать **драйверы** (**test driver**) и/или **заглушки** (**stub**)
- **Инкрементальный**, характеризующийся помодульным наращиванием комплекса программ с **пошаговым тестированием** собираемого комплекса.

В инкрементальном методе выделяют две стратегии добавления модулей:

- "Сверху вниз" (*нисходящее тестирование)*
- "Снизу вверх" (*восходящее тестирование)*
- «Сэндвич»

Системное тестирование

- Основная *задача системного тестирования* - выявление дефектов, связанных с работой системы в целом:
  - отсутствующая или неверная функциональность
  - неверное использование ресурсов системы
  - непредусмотренные комбинации данных пользовательского уровня
  - несовместимость с окружением
  - непредусмотренные сценарии использования
  - неудобство в применении и тому подобное.
- Системное тестирование производится над проектом в целом с помощью метода «черного ящика».

Категории тестов системного тестирования

- Полнота решения функциональных задач.
- Тестирование целостности (соответствие документации, комплектность).
- Проверка инсталляции и конфигурации на разных платформах.
- Оценка производительности.
- Стрессовое тестирование - на предельных объемах нагрузки входного потока.
- Корректность использования ресурсов (утечка памяти, возврат ресурсов).
- Эффективность защиты от искажения данных и некорректных действий.
- Корректность документации и т.д.

Объемы данных на этом уровне таковы, что обычно более эффективным подходом является полная или частичная *автоматизация тестирования*

	Модульное	Интеграционное	Системное
Типы дефектов	Локальные дефекты	Интерфейсные дефекты	Отсутствующая функциональность, ошибки совместимости, документации, переносимости, проблемы производительности, инсталляции и т.п.
Необходимость в системе тестирования	Да	Да	Нет (прямой необходимости в системе тестирования нет, но цена процесса *системного тестирования* часто настолько высока, что требует использования систем автоматизации, несмотря на возможно высокую их стоимость.)
Цена разработки системы тестирования	Низкая	Низкая до умеренной	Умеренная до высокой или неприемлимой
Цена процесса тестирования	Низкая	Низкая	Высокая

Приемочное тестирование

Приемочное тестирование *(Acceptance testing)* - тестирование готового продукта конечными пользователями в реальном окружении. Приемочные тесты разрабатываются пользователями (обычно в виде сценариев).

Unit Testing -> Integration Testing -> System Testing -> Acceptance Testing

9. Техники тест дизайна. Разбиение на классы эквивалентности и тестирование граничных значений (ТРО_03, http://www.protesting.ru/testing/testdesign_technics.html)

Наиболее распространенные техники тест-дизайна

- **Эквивалентное Разделение** (**Equivalence Partitioning - EP**). Как пример, у вас есть диапазон допустимых значений от 1 до 10, вы должны выбрать одно верное значение внутри интервала, скажем, 5, и одно неверное значение вне интервала - 0.
- **Анализ Граничных Значений** (**Boundary Value Analysis - BVA**). Если взять пример выше, в качестве значений для позитивного тестирования выберем минимальную и максимальную границы (1 и 10), и значения больше и меньше границ (0 и 11). Анализ Граничный значений может быть применен к полям, записям, файлам, или к любого рода сущностям имеющим ограничения.
- **Причина / Следствие** (**Cause/Effect - CE**). Это, как правило, ввод комбинаций условий (причин), для получения ответа от системы (Следствие). Например, вы проверяете возможность добавлять клиента, используя определенную экранную форму. Для этого вам необходимо будет ввести несколько полей, таких как "Имя", "Адрес", "Номер Телефона" а затем, нажать кнопку "Добавить" - эта "Причина". После нажатия кнопки "Добавить", система добавляет клиента в базу данных и показывает его номер на экране - это "Следствие".
- **Предугадывание ошибки** (**Error Guessing - EG**). Это когда тест аналитик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы "предугадать" при каких входных условиях система может выдать ошибку. Например, спецификация говорит: "пользователь должен ввести код". Тест аналитик, будет думать: "Что, если я не введу код?", "Что, если я введу неправильный код? ", и так далее. Это и есть предугадывание ошибки.
- **Исчерпывающее тестирование** (**Exhaustive Testing - ET**) - это крайний случай. В пределах этой техники вы должны проверить все возможные комбинации входных значений, и в принципе, это должно найти все проблемы. На практике применение этого метода не представляется возможным, из-за огромного количества входных значений.

10. Понятие дефекта. Основные определения и классификация дефектов. Описание дефектов (ТРО_04)

Основные определения

- Ошибки в ПО - все возможные несоответствия между демонстрируемыми характеристиками его качества и сформулированными или подразумеваемыми требованиями и ожиданиями пользователей.
- Англ. термины, кот. часто переводят как "ошибка":
- defect — самое общее нарушение каких-либо требований или ожиданий, не обязательно проявляющееся вовне (нарушения стандартов кодирования, недостаточная гибкость системы и пр.).
- failure — наблюдаемое нарушение требований, проявляющееся при каком-то реальном сценарии работы ПО. Это можно назвать проявлением ошибки.
- **fault** — ошибка в коде программы, вызывающая нарушения требований при работе (failures), то место, которое надо исправить.

Хотя это понятие используется довольно часто, оно не вполне четкое, поскольку для устранения нарушения может потребоваться исправить программу в нескольких местах. Что именно надо исправлять, зависит от дополнительных условий, выполнение которых мы хотим при этом обеспечить, хотя в некоторых ситуациях наложение дополнительных ограничений не устраняет неоднозначность.

- **error** — используется в двух смыслах:
    a. Ошибка в ментальной модели программиста, в его рассуждениях о программе, которая заставляет его делать ошибки в коде (faults). Это ошибка, которую сделал человек в своем понимании свойств программы.
    b. Некорректные значения данных (выходных или внутренних), которые возникают при ошибках в работе программы

- **Недостаток** (**Fault**) **/ Дефект** (**Defect**) – некорректное поведение системы, вызывающее сбой

- **Сбой** (**Failure**) – наблюдаемый нежелательный эффект, вызванный дефектами в системе

- **Ошибка** (**Error**) – человеческое действие, которое вызывает некорректный результат работы системы

Все это – баги (bugs)
Определения по стандарту
Отказ (IEEE - fault) - наблюдаемое аномальное поведение любого объекта, такое как несоответствие требованиям и возникновение незапланированных явлений - (**симптом**).
Сбой (IEEE - failure) - просчет в проектировании, ведущий к появлению отказов (симптомов) у какого-либо тестируемого объекта при прохождении этим объектом определенного теста - (**ошибка**).

11. Атрибуты дефектов. Приоритет (priority) и серьезность (severity) дефектов (ТРО_04)

2 основных признака классификации:

- **Серьезность** (**severity**) – степень влияния дефекта на продукт
- **Приоритет** (**priority**) – степень важности/срочности исправления дефекта

их соотношение определяется спецификой проекта
Severity (серьезность):

- фатальная (fatal)
- серьезная (serious)
- ошибка неудобства (inconvenient)
- косметическая (cosmetic)
- предложение по улучшению (suggestion for improvement, feature request)

Priority (приоритет):

- высокий (high)
- нормальный (medium)
- низкий (low)

Жизненный цикл дефекта

Поддерживают жизненный цикл дефектов

- **Test Track (Pro)**
  - Разработчик - Seapine Software
  - Клиент\серверный баг-трекер для Windows
- **Rational ClearQuest**
  - IBM Rational
- **Bugzilla**
  - Mozilla

12. Определение серьезности дефекта по его описанию – практическое задание (ТРО_04)

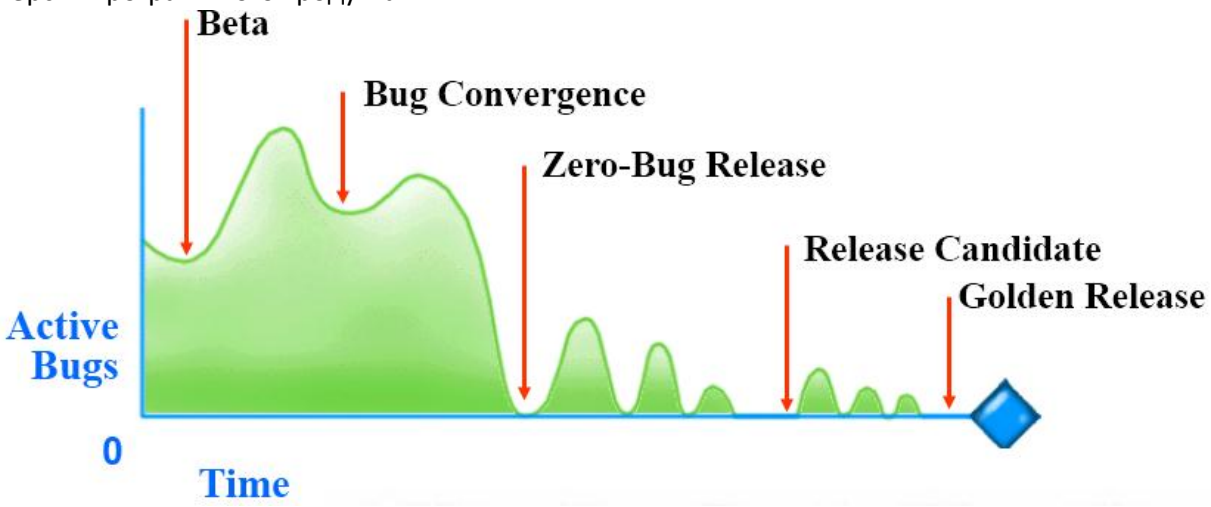
Пример дефекта

Bug report for Heron.exe

Short bug description	Severity	Priority	Full bug description
Overflow	High	Medium	If set large values for sides we get wrong area calculation. For example: First side = 999; Second side = 999; Third side = 999 Calculated area = 15,2
No error message if the area couldn`t be calculated	Medium	Low	If set “0” for one more sides we get area = 0. If set one side greater than sum of two others we get area = 0. A message should pop up that it`s impossible to from triangle

13. Версионирование ПО на разных стадиях разработки (TPO_04)

Версии программного продукта



- **Альфа (Alpha)** – версия с основной функциональностью, готовая к внутреннему тестированию
- **Бета (Beta)** – версия с основной функциональностью, готовая к тестированию внешними тестерами и\или пилотными пользователями

Бета версия (Beta)

Производится тестирование стабильной версии продукта внешними пользователями

- Техническая бета
- Маркетинговая бета

Точка конвергенции багов (Bug Convergence)

Точка проекта, в которой количество исправленных багов намного превосходит количество найденных

- Трудно вычислить эту точку, так как количество багов – величина постоянно меняющаяся
- Показывает скорее тренд, а не состояние проекта

Версия «0 багов» (Zero-Bug Release)

Первая версия проекта, в которой все выявленные баги высокого и среднего приоритета исправлены

- Требует проведения анализа багов, их приоритета
- «Начало конца»

Версия «Кандидат» (Release Candidate)

Версия проекта, на которой проводится финальное тестирование

- Собирается, когда продукт потенциально готов к выпуску
- После финального тестирования может появиться необходимость еще одного «кандидата»

Версия «Выпускной» (Golden Release)

Утвержденный «кандидат»

- Закончена разработка
- Закончено тестирование
- Подписаны документы

14. Инструментальные средства поддержки тестирования. Системы отслеживания ошибок (Bug Tracking Systems) (TPO_07, TPO_04, wiki)

Автоматизация тестирования

Обзор инструментов: основные игроки рынка

- IBM Rational
- Mercury Interactive
- Segue
- LDRA

Rational IBM

- **Rational Test Manager (**управление тестированием)
- **Rational Robot (**функциональное и нагрузочное тестирование)
- **Rational XDE Tester** (Rational Functional Tester for Java and Web) («eXtended Development Environment»; функциональное тестирование Java и Web приложений)
- **Rational PurifyPlus (Purify, PureCoverage, Quantify) (**анализ работы системы в режиме RunTime: контроль над ошибками доступа памяти, определение покрытия кода, измерение перфоманса)

Mercury

- **WinRunner (**функциональное тестирование)
- **QuickTest Pro (**инструмент для тестирования Web приложений)
- **XRunner (**поддержка Java приложений)
- **LoadRunner (**тестирование перфоманса)

Segue

- **SilkCentral Test Manager (**управление тестированием)
- **SilkCentral Performance Manager + Silk Performer (**тестирование производительности и управление им)
- **Silk Test (**инструмент функционального тестирования)

LDRA

- **LDRA Testbed (**тестирование «белого ящика» (покрытие кода, анализ источника кода))
- **TBreq (**автоматизация управления требованиями)
- **TBRun (**тестирование юнитов (unit testing))

Системы отслеживания ошибок

Система отслеживания ошибок (англ. *bug tracking system*) — прикладная программа, разработанная с целью помочь разработчикам программного обеспечения(программистам, тестировщикам и др.) учитывать и контролировать ошибки и неполадки, найденные в программах, пожелания пользователей, а также следить за процессом устранения этих ошибок и выполнения или невыполнения пожеланий.

Состав информации о дефекте

Главный компонент такой системы — база данных, содержащая сведения об обнаруженных дефектах. Эти сведения могут включать в себя:

- номер (идентификатор) дефекта;
- кто сообщил о дефекте;
- дата и время, когда был обнаружен дефект;
- версия продукта, в которой обнаружен дефект;
- серьёзность (критичность) дефекта и приоритет решения[1];

- описание шагов для выявления дефекта (воспроизведения неправильного поведения программы);
- кто ответственен за устранение дефекта;
- обсуждение возможных решений и их последствий;
- текущее состояние (статус) дефекта;
- версия продукта, в которой дефект исправлен.

Кроме того, развитые системы предоставляют возможность прикреплять файлы, помогающие описать проблему (например, дамп памяти или скриншот).

Примеры систем отслеживания ошибок

Свободно распространяемые

- Redmine
- BUGS - the Bug Genie http://www.thebuggenie.com/
- Bugzilla http://www.bugzilla.org/features/
- GNATS
- Mantis bug tracking system
- Trac

Проприетарные

- Atlassian JIRA
- Bontq
- Project Kaiser
- TrackStudio Enterprise
- YouTrack

Разное

- BugTracker.NET
- BugNet
- ClearQuest
- StarTeam

15. Артефакты разработки ПО, относящиеся к тестированию. Тест-кейсы (ТРО_05)

В настоящее время широко используется термин *test case* (тестовый пример) в качестве синонима слова *тест*

Тестовый пример (test case) – это совокупность

- Конфигурации системы
- Входных данных
- Начальных условий
- Алгоритма действий (сценарий). Может содержать ветвления (условия, переходы), однако лучше, чтобы он был линейным и как можно более коротким

- Ожидаемых результатов (и конечного состояния, которое может отличаться от начального состояния/условий)

Структура тестового примера (test case) – основное

- Идентификатор
- Название
- Автор
- Название проекта
- Цель
- Ссылки
- Среда выполнения
- **Пошаговое описание**
- Критерий выполнения

Структура тестового примера – дополнительные поля

- Краткое описание
- Полное описание
- Метка (для конфигурационного менеджмента)
- Приоритет
- Статус
- Название модуля

Пример test case

##	**Test point / Description**
	CTC#1.19 "Location register verification" Description: TC is to verify data from "Location" register. PURPOSE: To verify data from "Location" register, check values and the type of fields. DataSet: DDEMO2
1.	Open Location register: **Registers->Location data->Location**
2.	Add Location by pressing **add** button. Take values from below table. Write value.
3.	Press **Save** button
4.	**Repeat this test case with min, max and invalid values.**
3.	Close Location register by pressing **Exit** button.
4.	**End.**

16. Артефакты разработки ПО, относящиеся к тестированию. План тестирования (TPO_05)

Тестовый план - это документ, включающий:

- объем
- ресурсы
- календарный план работ по тестированию
- выполняемые тесты
- тестируемые элементы
- задачи тестирования

- ответственные сотрудники
- вероятность возникновения непредвиденных обстоятельств и меры, которые потребуется при этом принимать

(стандарт ANSI/IEEE 829-2983 for Software Test Documentation)

Назначение тестового плана

- служит для поиска ошибок
- облегчает управление работами и контроль хода их выполнения
- облегчает организацию технических аспектов тестирования
- помогает организовать и скоординировать усилия сотрудников, разрабатывающих и тестирующих программный продукт
- повышает эффективность и полноту тестирования
- документация должна быть не объемной, а эффективной. Любые составляющие плана, не помогающие в поиске ошибок и организации тестирования, являются пустой тратой ресурсов
- продукт (стОит дороже)
- рабочий инструмент

Разработка тестового плана

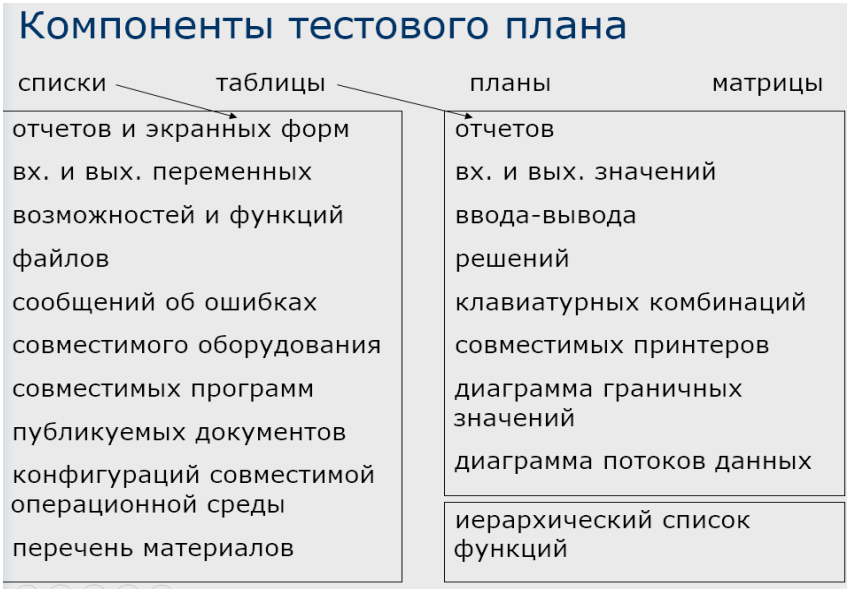
Как правило, применяется эволюционный подход (проведение тестирования параллельно с разработкой его плана)

Первый этап - начальная разработка:

- Проработка спецификации / пользовательской документации
- Первая версия списка функций программы (полнота списка определяет полноту тестирования, список будет постепенно расширяться)
- Анализ входных данных и ограничений (простейший анализ граничных условий)

Направления развития плана

- Наиболее вероятные ошибки (чем больше ошибок обнаружено в некоторой области программы, тем больше их там же)
- Наиболее заметные ошибки (пользователю)
- Наиболее часто используемые области программы
- Отличительные особенности программы (то, что отличает от конкурентов)
- Самые сложные аспекты для тестирования
- Самые понятные функциональные области



Матрицы:

- аппаратной и программной совместимости
- аппаратных конфигураций
- операционных окружений
- комбинаций входных значений
- сообщений об ошибках и клавиатурных комбинаций

Источники материалов:

- спецификация
- заметки разработчиков
- черновики руководства пользователя и другой документации
- устные беседы с руководством и программистами
- результат собственного опыта, полученного в ходе экспериментов над программой

Иерархический список функций системы

- Перечень всех высокоуровневых действий пользователя
- Подфункции всех функций (все доступные опции и варианты)
- Детализация до элементарных логических действий программы
- Перечислить входные и выходные условия для каждой функции и подфункции
- Список всех способов диалога с программой при выполнении каждой из функций (клавиатура, мышь)

Каждая строка этого списка в конце концов преобразуется в тестовый пример

Разделы тестового плана по стандарту

- идентификатор
- введение
- тестируемые элементы (программные компоненты, подлежащие тестированию)
- тестируемые функции
- нетестируемые функции
- подход к тестированию (кто, виды работ, технологии и средства, критерии, крайние сроки)
- критерии прохождения тестов
- документация
- необходимое оборудование
- календарный план
- ответственность