**ТПО. ОТВЕТЫ**

1. **Тестирование ПО. Формальное определение, основные понятия**

*Тестирование* – главный способ обеспечить качество программного продукта.

Тестирование сопровождает различные этапы жизненного цикла разработки программного продукта.

Оценка распределения трудоемкости между фазами создания ПП:

* 40% – Проектирование (дизайн приложения)
* 20% – Разработка кода
* 40% – Тестирование + отладка

Формальное определение тестирование сводится к доказательству тождества:

ТЗ фактический результат

f – функция, которая описана в техническом описании

f1\* f2\* f3\*…\* fn – фактический результат (если аналог, то операторы языка программирования, то их формула – программа).

Существует два метода обоснования истинности формул:

1. *Формальный подход*

Из исходных формул с помощью формальных процедур выводятся искомые формулы и утверждения (теоремы).

С его помощью удается избегать обращений к бесконечной области значений и на каждом шаге доказательства оперировать только конечным множеством символов.

Пример: (a+b)2=a2+2ab+b2

1. *Интерпретационный подход*

Тестирование проводится на некотором наборе входных данных:

X = {x1, x2, …, xn} – множество всех входных данных

Y = {y1, y2, …, yn} – ожидаемые значения на выходе

(X, Y) = {(x1, y1), …, (xn, yn)} – траектория верного исполнения

Тестирование – это проверка (Xi, Yвых) ∊ (X, Y). Реальная траектория выполнения программы может отличаться от ожидаемой (Yвых i != Yi).

СЛОЖНОСТЬ: на конечных множествах комбинации возможных значений для реализации исчерпывающей проверки могут оказаться достаточно велики.

Используется при экспериментальной проверке соответствия программы своей спецификации.

Основные понятия:

*Тестирование* – это процесс контроля выполнения программы с подачей на вход тестовых данных и анализом результатов. Тестирование обеспечивает выявление расхождений с требованиями (ошибок).

*Отладка* – процесс поиска, локализации и исправления ошибок в программе.

Если программа не содержит синтаксических ошибок и может быть выполнена на компьютере, она обязательно вычисляет какую-либо функцию, осуществляющую отображение входных данных в выходные.

Тестирование разделяют на статическое и динамическое:

* *Статическое тестирование* выявляет ошибки формальными методами анализа без выполнения тестируемой программы с помощью специальных инструментов контроля кода – *CodeChecker*.
* *Динамическое тестирование* осуществляет выявление ошибок только на выполняющейся программе с помощью специальных инструментов автоматизации тестирования – *Testbed* или *Testbench*.

1. **Организация тестирования ПО**

Тестирование включает 3 основных этапа:

* Составление тестовых данных (test suite);
* Прогон программы на тестовых данных под управлением тестовых драйверов или тестового монитора с получением протокола результатов тестирования (logging);
* Оценка результатов выполнения программы на наборе тестов с целью принятия решения о продолжении или остановке тестирования;

Тестирование осуществляется на заданном заранее множестве входных данных X и множестве предполагаемых результатов Y – (X, Y), которые задают график желаемой функции. Кроме того, описана процедура Оракул (oracle), которая определяет, соответствуют ли выходные данные – Yвых (вычисленные по входным данным – X) желаемым результатам – Y.

Оракулом может быть даже заказчик или программист, производящий соответствующие вычисления в уме.

При выявлении (x, yвых), не принадлежащего (X, Y), запускается процедура исправления ошибки, которая заключается во внимательном анализе (просмотре) протокола промежуточных вычислений, приведших к (x, yвых).

1. **Основные подходы к поиску ошибок. Проблемы при тестировании**

Анализ протокола промежуточных вычислений может проводится с помощью следующих методов:

* "Выполнение программы в уме";
* Вставка операторов протоколирования (печати) промежуточных результатов;
* Пошаговое выполнение программы:

Команды пошагового выполнения:

* *Step Into* – вызов, программа останавливается на первой строчке вызываемой функции, процедуры или метода.
* *Step Over* – вызов и выполнение всей функции, программа останавливается на первой строчке после вызываемой функции.
* *Step Out* – продолжит выполнение функции и остановит выполнение на первой строчке после вызываемой функции.
* Выполнение с заказанными остановками, анализом трасс или состояний памяти – дампов:

Контрольная точка (breakpoint) – точка программы, которая при её достижении посылает отладчику сигнал, приостанавливается выполнение программы, либо запускается программа "агент".

Затем отлаживаемая программа переходит в режим останова (break mode). Вход в режим останова позволяет анализировать состояние отдельных переменных или структур данных. Возврат из режима break mode в режим выполнения может произойти в любой момент по желанию пользователя.

Когда в контрольной точке вызывается программа "агент", она приостанавливает выполнение отлаживаемой программы на время, необходимое для фиксации состояния дампа в Log-файле, после чего происходит возврат в режим исполнения.

Трасса – это "сохраненный путь" на управляющем графе программы.

Дамп – область памяти, состояние которой фиксируется в контрольной точке в виде единого массива или нескольких связанных массивов.

* Реверсивное (обратное) выполнение:

Обратное выполнение программы возможно при условии сохранения на каждом шаге программы всех значений переменных или состояний программы для соответствующей трассы.

В процессе тестирования сравнение промежуточных результатов с полученными независимо эталонными результатами позволяет найти причины и место ошибки, исправить текст программы и продолжить тестирование.

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

Можно выделить следующие проблемы тестирования:

* Невозможно тестирование на всех входных значениях;
* Невозможно тестирование на всех путях;
* Нет 100% гарантии качества тестирования;

Следовательно, надо отбирать конечный набор тестов, позволяющий проверить программу на основе интуитивных представлений

Требование к тестам: программа на любом из них должна останавливаться, т.е. не зацикливаться. Можно ли заранее гарантировать останов на любом тесте?

* Общего метода для решения этого вопроса не существует.

Задача о выборе конечного набора тестов (X, Y) для проверки программы в общем случае неразрешима.

1. **Классы критериев тестирования**

* Структурные критерии используют информацию о структуре программы (критерии "белого ящика");
* Функциональные критерии основываются на спецификации требований, структура программы при этом не учитывается (критерии "черного ящика");
* Критерии стохастического тестирования формулируются в терминах проверки некоторой статистической гипотезы;
* Мутационные критерии ориентированы на проверку свойств программного продукта на основе подхода Монте-Карло;

1. **Идеальный критерий тестирования**

Для того чтобы проверить качество тестирования, используются критерии тестирования.

Идеальный критерий тестирования должен быть:

* *Достаточным*, т.е. показывать, когда некоторое конечное множество тестов достаточно для тестирования данной программы;
* *Полным*, т.е. в случае ошибки должен существовать тест из множества тестов, удовлетворяющих критерию, который раскрывает ошибку;
* *Надежным*, т.е. любые два множества тестов, удовлетворяющих ему, одновременно должны раскрывать или не раскрывать ошибки программы;
* *Легко проверяемым*, например вычисляемым на тестах;

Для нетривиальных классов программ в общем случае не существует полного и надежного критерия, зависящего от программ или спецификаций.

1. **Структурные критерии тестирования**

Структурные критерии используют модель программы в виде "белого ящика", что предполагает знание исходного текста программы или спецификации.

Для анализа структуры алгоритма строится управляющий граф.

Независимые ветви – разные пути выполнения программы, независимые друг от друга. Набор тестов соответствует структурному критерию если он заставляет программу пройти по каждой независимой ветви графа потока управления.

Виды:

* Критерий тестирования команд (набор тестов в совокупности должен обеспечить прохождение каждой команды не менее 1 раза)
* Критерий тестирования ветвей (набор тестов в совокупности должен обеспечить прохождение каждой ветви не менее 1 раза)
* Критерий тестирования путей (набор тестов в совокупности должен обеспечить прохождение каждого пути не менее 1 раза)

1. **Функциональные критерии тестирования**

Функциональный критерий обеспечивает контроль степени выполнения требований заказчика в программном продукте.

Поскольку требования формулируются к продукту в целом, они отражают взаимодействие тестируемого приложения с окружением. При функциональном тестировании преимущественно используется модель "черного ящика".

ПРОБЛЕМА: слишком большая трудоёмкость (документы, фиксирующие требования к программному изделию, достаточно объёмны), слишком много тестов.

Виды:

* Тестирование пунктов спецификации (набор тестов качественный, если каждый пункт спецификации требований проверяется хотя бы 1 раз)
* Тестирование классов входных данных (набор тестов в совокупности должен обеспечить проверку для каждого класса входных данных не менее 1 раза, область эквивалентности входных данных подмножества области определения функции, каждый элемент которых обрабатывает программа одинаково)
* Критерий тестирования правил (набор тестов в совокупности должен обеспечить проверку каждого правила, если входные и выходные значения описываются набором правил некоторой грамматики)
* Критерий тестирования функций (набор тестов считается качественным если он проверяет работу каждой функции хотя бы 1 раз)
* Критерий тестирования вызовов (каждый вызов функции будет проверятся набором тестов не менее 1 раза)

1. **Стохастические критерии тестирования**

Применяются, если множество входных данных имеет слишком большую мощность и не может быть уменьшено количество элементов.

* Разработать программы-имитаторы случайных последовательностей входных сигналов {x}.
* Вычислить независимым способом значения {y} для соответствующих входных сигналов {x} и получить тестовый набор (X,Y).
* Протестировать приложение на тестовом наборе (X,Y), используя два способа контроля результатов:

Недетерминированный контроль - проверка соответствия вычисленного значения y значению y, полученному в результате прогона теста на наборе {x} - случайной последовательности входных сигналов, сгенерированной имитатором.

Стохастический контроль - проверка соответствия множества значений {y}, полученного в результате прогона тестов на наборе входных значений {x}, заранее известному распределению результатов F(Y).

1. **Мутационный критерий тестирования**

Мутационный критерий основывается на том, что профессиональные программисты пишут почти идеальные программы, отличающиеся от правильных мелкими ошибками или очепятками.

*Мутации* – мелкие ошибки в программе.

*Мутанты* – программы, отличающиеся друг от друга мутациями

Метод мутационного тестирования – в разрабатываемую программу P вносят мутации, т.е. искусственно создают программы-мутанты P1, P2... Затем программа P и ее мутанты тестируются на одном и том же наборе тестов (X,Y).

Если на наборе (X,Y) подтверждается правильность программы P и, кроме того, выявляются все внесенные в программы-мутанты ошибки, то набор тестов (X,Y) соответствует мутационному критерию, а тестируемая программа объявляется правильной.

Если некоторые мутанты не выявили всех мутаций, то надо расширять набор тестов (X,Y) и продолжать тестирование.

1. **Модульное тестирование ПО**

Модульное тестирование – тестирование на уровне отдельно взятых модулей (классов, функций и др.).

Модуль – минимальная отдельно компилируемая единица кода.

Цель тестирования: удостовериться, что каждый модуль соответствует своей спецификации, выявить локальный алгоритм ошибки, обеспечить готовность к переходу на следующий уровень разработки и тестирования.

Использует подход белого ящика: знание внутренней структуры программы.

Необходимые элементы:

* Тестируемый программный модуль;
* Тестовый драйвер (программа, делающая вызов тестового модуля с тестовыми данными, анализ и вывод результатов);
* Программные заглушки (заменяют нереализованные части программы, с которыми необходимо взаимодействовать);

Тестирование классов обычно выполняется путем разработки тестового драйвера, который создает экземпляры классов и окружает их тестовым окружением, чтобы стал возможен прогон соответствующего тестового случая. Драйвер посылает сообщения экземпляру класса, а затем проверяет исход этих сообщений. Драйвер должен удалять созданные им экземпляры класса.

Способы реализации тестового драйвера:

1. Реализация в виде отдельного класса. Так можно тестировать public часть класса.
2. Реализация в виде класса, наследуемого от тестируемого. Тестовому драйверу доступна protected часть.
3. Реализация непосредственно внутри тестируемого класса (в класс добавляются диагностические методы). Тестовый драйвер имеет доступ ко всей реализации класса, включая private члены.

Если класс является законченным (final) и не предполагает наследования, необходимо тестирование его public части. Если же класс рассчитан на расширение за счет наследования, необходимо тестирование также его protected части.

1. **Разработка ПО через тестирование**

Разработка через тестирование (TDD) – одна из современных «гибких» методологий разработки программного обеспечения. Предполагает использование модульного тестирования для контроля разрабатываемого программного кода.

Основная идея: модульные тесты разрабатываются до разработки программного кода, который они будут тестировать.

Разработка таких тестов заставляет программиста подробно разобраться в требованиях к разрабатываемому модулю до начала разработки, четко определиться с конечными целями разработки; успешное выполнение тестов является критерием прекращения разработки.

Основные этапы:

1. получение программы в непротиворечивом состоянии и набора успешно выполняемых модульных тестов;
2. разработка нового модульного теста;
3. максимально быстрая разработка минимального программного кода, позволяющего успешно выполнить весь набор тестов;
4. рефакторинг разработанного программного кода с целью улучшения его структуры и устранения избыточности;
5. контроль переработанного программного кода с помощью полного набора тестов.

В результате выполнения этих шагов в программу будет добавлена новая функциональность, а работоспособность модифицированной программы будет гарантироваться выполнением полного набора модульных тестов. Такой подход позволяет избежать ситуации, когда добавление новой функции нарушает работоспособность ранее разработанного кода.

Преимущества:

* предотвращается возникновение ошибок во вновь разработанном коде;
* возможность вносить изменения в существующий программный код без риска нарушить его работоспособность;
* модульные тесты могут быть использованы в качестве документации к программному коду, показывать способы обращения к соответствующим программным модулям;
* улучшается дизайн кода: тесты заставляют создавать более «легкие» и независимые компоненты;
* повышается квалификация разработчиков (разработка качественных модульных тестов требует глубоких знаний ООП и паттернов проектирования).

Недостатки:

* высокие требования к квалификации разработчиков;
* тестовое покрытие программного кода.

В идеале набор модульных тестов должен обеспечивать покрытие 100% программного кода. Однако на практике этого достичь не удается. Поэтому необходимо обеспечить покрытие тестами наиболее критических модулей, а также компонентов, которые будут подвергаться рефакторингу.

1. **Интеграционное тестирование ПО**

Интеграционное тестирование (тестирование взаимодействия) – тестирование собранных вместе, взаимодействующих модулей (от двух до всех объектов тестируемой системы). Используется подход «белого ящика».

Цель: проверка правильности взаимодействия объектов.

Взаимодействие объектов представляет собой просто запрос одного объекта (отправителя) на выполнение другим объектом (получателем) одной из операций получателя и всех видов обработки, необходимых для завершения этого запроса.

Способы взаимодействия:

1. Общедоступная операция имеет один или большее число формальных параметров объектного типа.
2. Общедоступная операция возвращает значения объектного типа. На класс может быть возложена задача создания возвращаемого объекта, либо он может возвращать модифицированный параметр.
3. Метод одного класса создает экземпляр другого класса как часть своей реализации.
4. Метод одного класса ссылается на глобальный экземпляр некоторого другого класса (не рекомендуется).

Подходы к тестированию (и разработке):

1. *Метод большого взрыва*: все модули соединяются в единую программную систему и тестируются одновременно, проводится только после разработки всех отдельных модулей
2. *Инкрементное тестирование*: процесс тестирования программных модулей по одному, использует заглушки и драйверы для настройки передачи.
   1. *Восходящее*: разработка низкоуровневых модулей и объединение их в модули следующего уровня до тех пор, пока не будет собрана система

* Интегрируемые модули проще
* Путь от менее сложного к более сложному
* Меньше программных заглушек, более простые заглушки
  1. *Нисходящее*: разработка идёт от модулей верхнего уровня, низкоуровневые заменяются на заглушки
* При разработке необходимо разрабатывать сложные заглушки
* При тестировании поток управления тестом движется сверху вниз, начиная с пользовательского интерфейса (UI) и заканчивая базой данных программного обеспечения

1. **Тестирование интерфейсов**

*Интерфейс* – набор способов взаимодействия с модулем.

Цель тестирования: выявить ошибки неправильного взаимодействия с интерфейсами.

* Ошибки реализации (виноват вызываемый модуль)
* Ошибки предположений о работе интерфейса (виноват вызывающий модуль)

Классификация интерфейсов:

1. Параметрические:

* Передача указателей на данные/функции через параметры

1. Интерфейсы разделяемой памяти:

* Одна подсистема размещает данные в области памяти, другая считывает их

1. Процедурные:

* Подсистема инкапсулирует набор процедур, вызываемых другими подсистемами

1. Интерфейсы передачи сообщений

* Передача запроса от подсистемы отправителя к подсистеме-ответчику;

Классификация ошибок:

1. Неправильное использование интерфейсов:

* Сематические ошибки
* Характерны для параметрических интерфейсов

1. Неправильное понимание интерфейсов

* В вызывающий компонент заложена ошибочная интерпретация вызываемого компонента

1. Ошибки синхронизации

* Характерно для систем реального времени (разделяемой памяти, передачи сообщений)
* Ошибки совместной работы с памятью

Тестирование осложнено тем, что ошибки могут возникать только во внештатных ситуациях.

Алгоритм тестирования:

1. Просмотреть код, создать список вызовов внешних компонентов;
2. Для обнаруженных вызовов разработать области тестовых данных (рассмотреть минимальные и максимальные значения параметров);
3. Тестирование с нулевыми указателями;
4. Рассмотреть/придумать тестовые наборы, вызывающие сбой компонента;
5. Интерфейс передачи сообщений тестировать с нагрузкой: увеличивать интенсивность и объём сообщений. Определить лимит передачи.
6. Для интерфейса разделяемой памяти случайным образом менять порядок запуска компонентов;
7. Значительную часть ошибок позволяют вычислить статические методы, не требующие запуска программы;
8. **Тестирование с нагрузкой**

*Нагрузочное тестирование* – тестирование, направленное на проверку способности программы эффективно работать при пиковых или очень высоких нагрузках, связанных с большим количеством запросов пользователей.

* Сталкивает систему с загрузкой, превышающей предельно допустимую;
* Нужно убедиться, что выход из строя не приводит к необратимым последствиям;
* Подача запросов с предельной интенсивностью;
* Интенсивность увеличивается до первого сбоя;
* После отказа системы анализируется целостность данных и состояние системы;

Виды нагрузочного тестирования:

* *Непосредственно нагрузочное*. Проверка приложения при средней нагрузке.
* *Стабильности*. Тестирование на длительном промежутке времени.
* *Отказоустойчивости и восстановления*. Как система отреагирует на внешний сбой и развернётся заново.
* *Стресс-тест*. Массивное увеличение нагрузки на систему (число запросов);
* *Объёма*. Что будет, если в базу данных резко записать 100 000 000 новых клиентов?
* *Масштабируемости*. Как быстро развернутся новые кластеры в облаке при необходимости?
* *Потенциальных возможностей*. Проверяются вообще все пределы приложения: максимум клиентов в минуту, максимум записей в БД и т.д.

1. **Системное тестирование**

Системное тестирование выполняется после полной сборки системы, системные тесты не являются частью кода.

* Подача данных через внешний интерфейс (UI/API);

Цель тестирования: проверка правильности реализации верхнеуровневых общих системных функций, интегрированных общесистемных свойств.

* Последний этап перед окончательной аттестацией;
* Автоматизированное или ручное;
* Необходим тестовый сценарий: последовательность взаимодействия с системой;
* Может проводится в конце каждой итерации;

*Приемочное тестирование* – вид тестирования, проводимый на этапе сдачи готового продукта (или готовой части продукта) заказчику.

Цель: определение готовности продукта, что достигается путем прохода тестовых сценариев и случаев, которые построены на основе спецификации требований к разрабатываемому ПО.

Результатом приемочного тестирования может стать:

* + - Отправка проекта на доработку.
    - Принятие его заказчиком, в качестве выполненной задачи.

1. **Регрессионное тестирование ПО.**

*Регрессионное тестирование* — это процесс повторного выполнения тестов на программном обеспечении для обнаружения новых ошибок, возникших в результате внесения изменений или доработок.

Основная цель регрессионного тестирования — обеспечить стабильность программного продукта после внесения изменений.

Метод помогает убедиться, что все предыдущие функции все еще работают корректно, и не возникло новых ошибок.

Проблема: компромисс между полным тестирование системы и тестированием лишь изменённых модулей (дорого vs недостаточно полно).

Необходимо найти минимально достаточный набор тестов, подтверждающих правильную сборку системы.

1. **Верификация и аттестация информационных систем**

Процессы верификации и аттестации охватывают весь цикл разработки.

Это проверка и анализ, в ходе которого проверяется программное обеспечение на соответствие требованиям и ожиданиям заказчика

* Верификация отвечает на вопрос правильно ли система создана
* Аттестация отвечает на вопрос правильно ли система работает

*ЭТО НЕ ОДНО И ТОЖЕ!*

*Верификация* – проверка ПО на соответствие требованиям (функциональным и нефункциональным), описанным в ТЗ.

*Аттестация* – процесс приемки ПО, где проверяется его соответствие с ожиданием заказчика (подходит или не подходит заказчику).

То есть система, успешно пройденная верификацию, может не соответствовать ожиданиям заказчика.

1. **Инспектирование программного обеспечения**

Используется 2 основных подхода верификации и аттестации:

* + - Тестирование – динамический способ верификации и аттестации.
    - Инспектирование ПО – статический анализ программного кода, спецификации требования и т.п. (не требующий запуск и выполнение программы).

С помощью статического анализа нельзя проверить производительность, надежность и т.п.

Инспектирование ПО – просмотр программы с целью обнаружения ошибок.

Инспектирование может происходить вручную, либо с помощью программных инструментов.

Какие нефункциональные требования работают нельзя увидеть с помощью инспектирования. Это формализованный процесс, который выполняет инспекционная группа.

В состав инспекционной группы входит:

* + Автор
  + Инспектор – проверяет код
  + Координатор – отвечает за процесс (координирует)
  + Рецензент – озвучивает программный код

Для того чтобы начать инспектирование, требуется:

* + Нужна спецификация требований
  + Все участники инспекционной группы должны знать и понимать стандартное написание кода, которое принято в команде
  + В распоряжении группы должна быть синтаксически верная последняя версия программы

Инспектирование представляет следующие этапы:

* 1. Планирование
  2. Предварительный просмотр
  3. Индивидуальная подготовка
  4. Собрание инспекционной группы
  5. Исправление ошибок
  6. Доработка

1-2) Координатор составляет план инспектирования и подбирает участников инспекционной группы, этот же координатор организует собрание, чтобы убедиться, что спецификация программы в завершенном состоянии. Программа передается инспекционной группы, автор описывает назначение инспектируемого программного модуля. Происходит это все на этапе предварительного просмотра.

3) На этапе индивидуальной подготовки каждый участник изучает ПО самостоятельно и выявляет дефекты.

4) Озвучивается результат, аномалии, нестандартные методы и т.п. Не более 2-х часов длится. Инспекционная группа не учит автора программировать. Озвучиваются найденные дефекты.

5) Идет процесс исправления дефектов ПО

6) Координатор решает, делать ли повторное инспектирование. Если это не требуется, документально фиксируются найденные дефекты.

Есть общие оценки, например на этапе предварительного просмотра, можно просмотреть 500~ операторов за один час. Показывается ОГРАНИЧЕННОЕ количество кода.

1. **Оценка качества программного обеспечения**

Качественной целью тестирования, является качественный ПП.

Качество ПО определяется как совокупность характеристик относящиеся к возможности удовлетворяющей потребности всех заинтересованных лиц. Даже если ТЗ было не полное:

* *Качество процесса* (на сколько качественно организован процесс разработки)
* *Внутреннее качество* (Характеристика ПО, без учёта его использования)
* *Внешнее качество* (характеризует ПП с точки зрения его поведения)
* *Качество использования* (при использовании в разных сценариях)

Характеристики качественного ПП:

* Функциональность – способность ПП в определённых условиях решать задачи нужные пользователю
* Точность (способность выдавать нужный результат)
* Защищённость
* Надёжность – способность поддерживать работоспособность в заданное время без сбоев
* Понятность – усилия, затраченные пользователем, для того чтобы разобраться с ПП
* Удобство обучения – величина, обратная усилиям, затраченным на обучение работе с ПП
* Эффективность – способ обеспечить необходимую работоспособность приложения при выделенных ресурсах
* Удобство сопровождения
* Переносимость – способность сохранять работоспособность при переносе с одного устройства на другое
* Адаптируемость

1. **Показатели качества программного обеспечения**

Качество использования определяется характеристиками:

* Эффективность – способность ПП решать задачи пользователя с большой точностью.
* Продуктивность – способность предоставлять пользователю результат, не выходя за ожидания.
* Безопасность – способность минимизации ущерба при использовании
* Удовлетворительность пользователей

Методы контроля качества:

* Верификация (разработано в соответствии с требованиями)
* Валидация (аттестация)
* Тестирование
* Стоимость качества – стоимость тех проблем, которые могут принести не удовлетворительность

Характеристики:

* Стоимость предотвращения низкого качества
* Стоимость оценки
* Цена ошибки

1. **Профилирование программного обеспечения**

После разработки программного кода возникает задача оптимизации его производительности и использования памяти. Для анализа характеристик выполнения программы используется процесс, называемый *профилированием*, выполняется с помощью специальных программных инструментов, называемых *профайлерами*.

Программа-профайлер запускает программу и анализирует ее характеристики выполнения. Процесс профилирования включает *декомпозицию* программы на отдельные выполняющиеся элементы и измерение времени, затрачиваемого на выполнение каждого элемента, а также анализ использования памяти. В результате профилирования может быть построен *граф вызовов*, который показывает последовательность вызовов функций, а также определено время выполнения каждой функции и ее вклад в общее время выполнения программы.

Анализ этой информации позволяет выявить *критические участки* программного кода, называемые "хотспотами" или "узкими местами".

*Хотспоты* – это участки кода, которые занимают значительное время выполнения программы или используют большое количество памяти. Идентифицировав хотспоты, разработчики могут сосредоточиться на их оптимизации, чтобы улучшить производительность и эффективность программы.

Инструмент профилирования позволяет выполнять анализ производительности программы, строить граф вызовов методов, анализировать время выполнения программы и относительный вклад времени выполнения вызываемых функций.

1. **Тестирование на основе анализа областей эквивалентности данных**

Входные данные программ часто можно разбить на несколько классов. Данные, принадлежащие одному классу, имеют общие свойства. Обычно для всех данных из какого-либо класса поведение программы одинаково (эквивалентно). Из-за этого такие классы данных иногда называют *областями эквивалентности*.

*Области эквивалентности входных данных* — это множества данных, все элементы которых обрабатываются одинаково. *Области эквивалентности выходных данных* — это данные на выходе программы, имеющие общие свойства. *Корректные и некорректные* *входные данные* также образуют две области эквивалентности.

Один из методов обнаружения дефектов состоит в определении всех областей эквивалентности, обрабатываемых программой. Контрольные тесты разрабатываются так, чтобы входные и выходные данные лежали в пределах этих областей.

После определения областей эквивалентности, выбираются тестовые последовательности, принадлежащие каждой из областей. При этом руководствуются следующими правилами:

* 1. Тестирующая последовательность может состоять из одного элемента. Если ввести последовательность из *одного элемента*, программа может работать неправильно.
  2. Следует использовать в разных тестах различные последовательности, содержащие *разное количество элементов*. Это уменьшает вероятность того, что программа, имеющая дефекты, случайно выдаст правильные результаты.
  3. Следует использовать тестирующие последовательности, в которых ключевой элемент является *первым, средним и последним* элементом последовательности.

Такой метод помогает выявить проблемы на границах областей эквивалентности.

1. **Структурное тестирование**

Структурное тестирование создает тесты на основе структуры и реализации системы («белый ящик»). Оно обычно применяется к небольшим программным элементам, например, подпрограммам или методам, связанным с объектами.

Тестирование методом *белого ящика* - метод тестирования ПО, который предполагает, что внутренняя структура/устройство/реализация системы известны тому, кто её тестирует

Тестирование ветвей – это метод структурного тестирования, при котором проверяются все независимо выполняемые ветви компонента или программы.

Если выполняются все независимые ветви, то и все операторы должны выполняться, по крайней мере, один раз. Все эталонные операторы тестируются как с истинными, так и с ложными значениями условия.

При тестировании ветвей не проверяются все возможные комбинации ветвей программы. Полная проверка компонента оказывается нереальной, так как в программах с циклами существует бесконечное число возможных комбинаций ветвей. В программе могут быть дефекты, которые проявляются только при определенных комбинациях ветвей, даже если все операторы программы протестированы.

Метод тестирования ветвей основывается на графе потоков управления программы (скелетная модель всех ветвей программы). Граф состоит из узлов, соответствующих ветвлениям решений, и дуг, показывающих поток управления. Каждое ветвление операторов условного перехода представлено отдельной ветвью, а циклы обозначаются стрелками, концы которых замкнуты на узле с условием цикла.

Цель структурного тестирования: удостовериться, что каждая независимая ветвь программы выполняется хотя бы один раз.

*Независимая ветвь программы* — это ветвь, которая проходит, по крайней мере, по одной новой дуге графа потоков. В терминах программы это означает ее выполнение при новых условиях.

Если все независимые ветви выполняются, то можно быть уверенным в том, что:

* 1. каждый оператор выполняется по крайней мере один раз;
  2. каждая ветвь выполняется при условиях, принимающих как истинные, так и ложные значения.

Количество независимых ветвей в программе можно определить, вычислив цикломатическое число графа потоков управления программы. Цикломатическое число любого связанного графа G вычисляется по формуле:

С (G) = количество дуг - количество узлов + 2.

После определения количества независимых ветвей в программе путем вычисления цикломатического числа разрабатываются контрольные тесты для проверки каждой ветви. Минимальное количество тестов, требующееся для проверки всех ветвей программы равно цикломатическому числу.

1. **Использование программных фреймворков для реализации модульных тестов**

NUnit — открытый фреймворк для тестирования приложений под .NET Framework. Включает библиотеки для написания и ПО для выполнения тестов.

Основные атрибуты NUnit:

[TestFixture]: Атрибут, который указывает, что класс содержит модульные тесты.

[Test]: Атрибут, который указывает, что метод является модульным тестом. Этот атрибут может быть применен к отдельным методам внутри класса тестов.

[SetUp]: Атрибут, который указывает, что метод должен быть выполнен перед каждым модульным тестом. Это позволяет настроить начальные условия перед выполнением каждого теста.

[TearDown]: Атрибут, который указывает, что метод должен быть выполнен после каждого модульного теста. Это позволяет выполнять очистку или завершающие действия после каждого теста.

[TestCase]: Атрибут, который позволяет задавать параметризованные тесты. Вы можете указать различные значения аргументов для каждого выполнения теста, чтобы проверить различные сценарии.

Проверки, доступные в классе Assert:

1. Assert.AreEqual – проверяет равенство входных параметров;

2. Assert.AreNotEqual – проверяет то, что входные параметры неравны;

3. Assert.AreSame – проверка на то, что входные параметры ссылаются на один и тот же объект;

4. Assert.AreNotSame – входные параметры не ссылаются на один и тот же объект;

5. Assert.Contains – метод получает на входе объект и коллекцию и проверяет, что данный объект содержится в это коллекции;

6. Assert.IsNull – входной параметр – null;

7. Assert.IsEmpty – входной параметр – пустая коллекция.

8. Assert.Fail – прерывает выполнение теста и среде NUnit, что тест не пройден. Эта функция может быть использована, когда нужно организовать более сложные типы проверок.