Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«Самарский государственный технический университет»

Факультет Автоматики и информационных технологий

Кафедра «Вычислительная техника»

ЛЕКЦИИ
для студентов заочной формы обучения

по дисциплине: **«Тестирование и отладка программного обеспечения»**

(8 часов)
основная образовательная программа по направлению подготовки (специальности): 231000 «Программная инженерия»

по уровню высшего образования: бакалавр

направленность (профиль) программы: «Программная инженерия»

Самара 2015

Концепция и организация процесса тестирования

Существующие на сегодня методы тестирования программного обеспечения не позволяют однозначно и полностью выявить все дефекты и установить корректность функционирования анализируемой программы, поэтому все существующие методы тестирования действуют в рамках формального процесса проверки исследуемого или разрабатываемого программного обеспечения.

Такой процесс формальной проверки, или верификации, может доказать, что дефекты отсутствуют с точки зрения используемого метода. (То есть нет никакой возможности точно установить или гарантировать отсутствие дефектов в программном продукте с учётом человеческого фактора, присутствующего на всех этапах жизненного цикла программного обеспечения.)

Существует множество подходов к решению задачи тестирования и верификации программного обеспечения, но эффективное тестирование сложных программных продуктов — это процесс в высшей степени творческий, не сводящийся к следованию строгим и чётким процедурам или созданию таковых.

Качество программного обеспечения можно определить как совокупную характеристику исследуемого ПО с учётом следующих составляющих:

- надёжность
- сопровождаемость,
- практичность,
- эффективность,
- мобильность,
- функциональность.

Состав и содержание документации, сопутствующей процессу тестирования, определяется стандартом IEEE 829-1998.

Первые программные системы разрабатывались в рамках программ научных исследований или программ для нужд министерств обороны. Тестирование таких продуктов проводилось строго формализованно с записью всех тестовых процедур, тестовых данных, полученных результатов. Тестирование выделялось в отдельный процесс, который начинался после

завершения кодирования, но при этом, как правило, выполнялось тем же персоналом.

В 1960-х много внимания уделялось «исчерпывающему» тестированию, которое должно проводиться с использованием всех путей в коде или всех возможных входных данных. Было отмечено, что в этих условиях полное тестирование программного обеспечения невозможно, потому что, во-первых, количество возможных входных данных очень велико, во-вторых, существует множество путей, в-третьих, сложно найти проблемы в архитектуре и спецификациях. По этим причинам «исчерпывающее» тестирование было отклонено и признано теоретически невозможным.

В начале 1970-х годов тестирование программного обеспечения обозначалось как «процесс, направленный на демонстрацию корректности продукта» или как «деятельность по подтверждению правильности работы программного обеспечения». В зарождавшейся **программной инженерии** верификация ПО значилась как «**доказательство правильности**». Хотя концепция была теоретически перспективной, на практике она требовала много времени и была недостаточно всеобъемлющей. Было решено, что доказательство правильности — неэффективный метод тестирования программного обеспечения. Однако, в некоторых случаях демонстрация правильной работы используется и в наши дни, например, приёмо-сдаточные испытания. Во второй половине 1970-х тестирование представлялось как выполнение программы с намерением найти ошибки, а не доказать, что она работает. Успешный тест — это тест, который обнаруживает ранее неизвестные проблемы. Данный подход прямо противоположен предыдущему. Указанные два определения представляют собой «парадокс тестирования», в основе которого лежат два противоположных утверждения: с одной стороны, тестирование позволяет убедиться, что продукт работает хорошо, а с другой — выявляет ошибки в программах, показывая, что продукт не работает. Вторая цель тестирования является более продуктивной с точки зрения улучшения качества, так как не позволяет игнорировать недостатки программного обеспечения.

В 1980-е годы тестирование расширилось таким понятием, как **предупреждение дефектов**. Проектирование тестов — наиболее эффективный из известных методов предупреждения ошибок. В это же время стали высказываться мысли, что необходима методология тестирования, в частности, что тестирование должно включать проверки на всем протяжении цикла разработки, и это должен быть управляемый процесс. В ходе тестирования надо проверить не только собранную программу, но и требования, код, архитектуру, сами тесты. «Традиционное» тестирование, существовавшее до начала 1980-х, относилось только к скомпилированной, готовой системе (сейчас это обычно называется системное тестирование), но в дальнейшем тестировщики стали вовлекаться во все аспекты жизненного цикла разработки. Это позволяло

раньше находить проблемы в требованиях и архитектуре и тем самым сокращать сроки и бюджет разработки. **В середине 1980-х появились первые инструменты для автоматизированного тестирования**. Предполагалось, что компьютер сможет выполнить больше тестов, чем человек, и сделает это более надёжно. Поначалу эти инструменты были крайне простыми и не имели возможности написания сценариев на скриптовых языках.

В начале 1990-х годов в понятие «тестирование» стали включать планирование, проектирование, создание, поддержку и выполнение тестов и тестовых окружений, и это означало переход от тестирования к обеспечению качества, охватывающего весь цикл разработки программного обеспечения. В это время начинают появляться различные программные инструменты для поддержки процесса тестирования: более продвинутые среды для автоматизации с возможностью создания скриптов и генерации отчетов, системы управления тестами, ПО для проведения нагрузочного тестирования. В середине 1990-х годов с развитием Интернета и разработкой большого количества веб-приложений особую популярность стало получать «гибкое тестирование» (по аналогии с гибкими методологиями программирования).

В 2000-х появилось ещё более широкое определение тестирования, когда в него было добавлено понятие «оптимизация бизнес-технологий». Основной подход заключается в оценке и максимизации значимости всех этапов жизненного цикла разработки программного обеспечения для достижения необходимого уровня качества, производительности, доступности.

Критерии и виды тестирования

Существует несколько признаков, по которым принято производить классификацию видов тестирования. Обычно выделяют следующие:

По объекту тестирования

- Функциональное тестирование
- Тестирование производительности
  - Нагрузочное тестирование
  - Стресс-тестирование
  - Тестирование стабильности
- Конфигурационное тестирование
- Юзабилити - тестирование
- Тестирование интерфейса пользователя
- Тестирование безопасности
- Тестирование локализации
- Тестирование совместимости

По знанию системы

- Тестирование чёрного ящика
- Тестирование белого ящика
- Тестирование серого ящика

По степени автоматизации

- Ручное тестирование
- Автоматизированное тестирование
- Полуавтоматизированное тестирование

По степени изолированности компонентов

- Модульное тестирование
- Интеграционное тестирование
- Системное тестирование

По времени проведения тестирования

- Альфа-тестирование
  - Дымовое тестирование (англ. *smoke testing*)
  - Тестирование новой функции (*new feature testing*)
  - Подтверждающее тестирование
  - Регрессионное тестирование
  - Приёмочное тестирование
- Бета-тестирование

По признаку позитивности сценариев

- Позитивное тестирование
- Негативное тестирование

По степени подготовленности к тестированию

- Тестирование по документации (формальное тестирование)
- Интуитивное тестирование (англ. *ad hoc testing*)

Уровни тестирования

Модульное тестирование — тестируется минимально возможный для тестирования компонент, например, отдельный класс или функция. Часто модульное тестирование осуществляется разработчиками программного обеспечения.

Интеграционное тестирование — тестируются интерфейсы между компонентами, подсистемами или системами. При наличии резерва времени

на данной стадии тестирование ведётся итерационно, с постепенным подключением последующих подсистем.

Системное тестирование — тестируется интегрированная система на её соответствие требованиям.

Альфа-тестирование — имитация реальной работы с системой штатными разработчиками, либо реальная работа с системой потенциальными пользователями/заказчиком. Чаще всего альфа-тестирование проводится на ранней стадии разработки продукта, но в некоторых случаях может применяться для законченного продукта в качестве внутреннего приёмочного тестирования. Иногда альфа-тестирование выполняется под отладчиком или с использованием окружения, которое помогает быстро выявлять найденные ошибки. Обнаруженные ошибки могут быть переданы тестировщикам для дополнительного исследования в окружении, подобном тому, в котором будет использоваться программа.

Бета-тестирование — в некоторых случаях выполняется распространение предварительной версии (в случае проприетарного программного обеспечения иногда с ограничениями по функциональности или времени работы) для некоторой большей группы лиц с тем, чтобы убедиться, что продукт содержит достаточно мало ошибок. Иногда бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей.

Часто для свободного и открытого программного обеспечения стадия альфа-тестирования характеризует функциональное наполнение кода, а бета-тестирования — стадию исправления ошибок. При этом как правило на каждом этапе разработки промежуточные результаты работы доступны конечным пользователям.

Особенности процесса и технологии тестирования

Статическое и динамическое тестирование

Описанные ниже техники — тестирование белого ящика и тестирование чёрного ящика — предполагают, что код исполняется, и разница состоит лишь в той информации, которой владеет тестировщик. В обоих случаях это *динамическое тестирование*.

При статическом тестировании программный код не выполняется — анализ программы происходит на основе исходного кода, который вычитывается вручную, либо анализируется специальными инструментами. В некоторых случаях анализируется не исходный, а промежуточный код (такой как байт-код или код на MSIL).

Также к статическому тестированию относят тестирование требований, спецификаций, документации.

Регрессионное тестирование

После внесения изменений в очередную версию программы, регрессионные тесты подтверждают, что сделанные изменения не повлияли на работоспособность остальной функциональности приложения. Регрессионное тестирование может выполняться как вручную, так и средствами автоматизации тестирования.

Тестовые сценарии

Тестировщики используют тестовые сценарии на разных уровнях: как в модульном, так и в интеграционном и системном тестировании. Тестовые сценарии, как правило, пишутся для проверки компонентов, в которых наиболее высока вероятность появления отказов или вовремя не найденная ошибка может быть дорогостоящей.

Тестирование «белого ящика» и «чёрного ящика»

В зависимости от доступа разработчика тестов к исходному коду тестируемой программы различают «тестирование белого ящика» и «тестирование чёрного ящика».

При тестировании **белого ящика** (также говорят — *прозрачного ящика*), разработчик теста имеет доступ к исходному коду программ и может писать код, который связан с библиотеками тестируемого программного обеспечения. Это типично для модульного тестирования, 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.

При тестировании **чёрного ящика**, тестировщик имеет доступ к программе 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. Обычно в данном виде тестирования критерий покрытия складывается из покрытия структуры

входных данных, покрытия требований и покрытия модели (в тестировании на основе моделей).

При тестировании **серого ящика** разработчик теста имеет доступ к исходному коду, но при непосредственном выполнении тестов доступ к коду, как правило, не требуется.

Если «альфа-» и «бета-тестирование» относятся к стадиям до выпуска продукта (а также, неявно, к объёму тестирующего сообщества и ограничениям на методы тестирования), тестирование «белого ящика» и «чёрного ящика» имеет отношение к способам, которыми тестировщик достигает цели.

Бета-тестирование в целом ограничено техникой чёрного ящика (хотя постоянная часть тестировщиков обычно продолжает тестирование белого ящика параллельно бета-тестированию). Таким образом, термин «бета-тестирование» может указывать на состояние программы (ближе к выпуску чем «альфа»), или может указывать на некоторую группу тестировщиков и процесс, выполняемый этой группой. То есть, тестировщик может продолжать работу по тестированию белого ящика, хотя программа уже «бета-стадии», но в этом случае он не является частью «бета-тестирования».

Покрытие кода

Покрытие кода, по своей сути, является тестированием методом белого ящика. Тестируемое программное обеспечение собирается со специальными настройками или библиотеками или запускается в особом окружении, в результате чего для каждой используемой (выполняемой) функции программы определяется местонахождение этой функции в исходном коде. Этот процесс позволяет разработчикам и специалистам по обеспечению качества определить части системы, которые при нормальной работе используются очень редко или никогда не используются (такие как код обработки ошибок и т. п.). Это позволяет сориентировать тестировщиков на тестирование наиболее важных режимов.

Тестировщики могут использовать результаты теста покрытия кода для разработки тестов или тестовых данных, которые расширят покрытие кода на важные функции.

Как правило, инструменты и библиотеки, используемые для получения покрытия кода, требуют значительных затрат производительности и/или памяти, недопустимых при нормальном функционировании ПО. Поэтому они могут использоваться только в лабораторных условиях.

Разработка, тестирование и отладка программ

1. Составление текста программы

Это, наверное, самый сложный из этапов, требующий наибольшего внимания. По сути, составление текста программы – это запись алгоритма задачи при помощи одного из языков программирования. Чтобы этот текст был понятен пользователю и составителю, используются комментарии.

2. Синтаксическая отладка программы

Отладка программы — это специальный этап в разработке программы, состоящий в выявлении и устранении программных ошибок, факт существования которых уже установлен.

Синтаксическая отладка – поиск синтаксических ошибок в тексте программы. Обнаружив ошибку, транслятор выводит сообщение, указывая на место ошибки в программе и ее характер. Получив такое сообщение, программист должен исправить ошибку и снова повторить трансляцию. Так продолжается до тех пор, пока не будут исправлены все синтаксические ошибки.

Если вы сталкиваетесь с синтаксической ошибкой, то чаще всего вы можете решить проблему с помощью справочной системы, из которой можно получить дополнительную информацию об ошибке, и исправить эту ошибку, уделив дополнительное внимание точному синтаксису используемых вами функций, объектов, методов и свойств.

3 Тестирование и семантическая отладка

Тестирование – это динамический контроль программы, т.е. проверка правильности программы при ее выполнении на компьютере.

Каждому программисту известно, сколько времени и сил уходит на отладку и тестирование программ. На этот этап приходится около 50% общей стоимости разработки программного обеспечения. Но не каждый из разработчиков программных средств может верно, определить цель тестирования. Нередко можно услышать, что тестирование - это процесс выполнения программы с целью обнаружения в ней ошибок. Но эта цель недостижима: ни какое самое тщательное тестирование не дает гарантии, что программа не содержит ошибок. Другое определение: это процесс выполнения программы с целью обнаружения в ней ошибок. Отсюда ясно, что “удачным” тестом является такой, на котором выполнение программы завершилось с ошибкой. Напротив, “неудачным” можно назвать тест, не позволивший выявить ошибку в программе. Определение также указывает на объективную трудность тестирования: это деструктивный (т.е. обратный созидательному) процесс. Поскольку программирование - процесс конструктивный, ясно, что большинству разработчиков программных средств сложно “переключиться” при тестировании созданной ими продукции.

Основные принципы организации тестирования:

3.1. необходимой частью каждого теста должно являться описание ожидаемых результатов работы программы, чтобы можно было быстро выяснить наличие или отсутствие ошибки в ней;

3.2. следует по возможности избегать тестирования программы ее автором, т.к. кроме уже указанной объективной сложности тестирования для программистов здесь присутствует и тот фактор, что обнаружение недостатков в своей деятельности противоречит человеческой психологии (однако отладка программы эффективнее всего выполняется именно автором программы);

3.3. по тем же соображениям организация - разработчик программного обеспечения не должна “единолично ” его тестировать (должны существовать организации, специализирующиеся на тестировании программных средств);

3.4. должны являться правилом доскональное изучение результатов каждого теста, чтобы не пропустить малозаметную на поверхностный взгляд ошибку в программе;

3.5. необходимо тщательно подбирать тест не только для правильных (предусмотренных) входных данных, но и для неправильных (непредусмотренных);

3.6. при анализе результатов каждого теста необходимо проверять, не делает ли программа того, что она не должна делать;

3.7. следует сохранять использованные тесты (для повышения эффективности повторного тестирования программы после ее модификации или установки у заказчика);

3.8. тестирования не должно планироваться исходя из предположения, что в программе не будут обнаружены ошибки (в частности, следует выделять для тестирования достаточные временные и материальные ресурсы);

3.9. следует учитывать так называемый “принцип скопления ошибок” : вероятность наличия не обнаруженных ошибок в некоторой части программы прямо пропорциональна числу ошибок, уже обнаруженных в этой части;

3.10. следует всегда помнить, что тестирование - творческий процесс, а не относиться к нему как к рутинному занятию.

Существует **два основных вида тестирования: функциональное и структурное. При функциональном тестировании программа рассматривается как “черный ящик”** (то есть ее текст не используется). Происходит проверка соответствия поведения программы ее внешней

спецификации. Возможно ли при этом полное тестирование программы? Очевидно, что критерием полноты тестирования в этом случае являлся бы перебор всех возможных значений входных данных, что невыполнимо.

Поскольку исчерпывающее функциональное тестирование невозможно, речь может идти о разработки методов, позволяющих подбирать тесты не “вслепую”, а с большой вероятностью обнаружения ошибок в программе. **При структурном тестировании программа рассматривается как “белый ящик”** (т.е. ее текст открыт для пользования). Происходит проверка логики программы. Полным тестированием в этом случае будет такое, которое приведет к перебору всех возможных путей на графе передач управления программы (ее управляющем графе). Даже для средних по сложности программ числом таких путей может достигать десятков тысяч.

Таким образом, ни структурное, ни функциональное тестирование не может быть исчерпывающим. Рассмотрим подробнее основные этапы тестирования программных комплексов. В тестирование многомодульных программных комплексов можно выделить четыре этапа:

1) тестирование отдельных модулей;

2) совместное тестирование модулей;

3) тестирование функций программного комплекса (т.е. поиск различий между разработанной программой и ее внешней спецификацией);

4) тестирование всего комплекса в целом (т.е. поиск несоответствия созданного программного продукта, сформулированным ранее целям проектирования, отраженным обычно в техническом задании).

На первых двух этапах используются, прежде всего, методы структурного тестирования, т.к. на последующих этапах тестирования эти методы использовать сложнее из-за больших размеров проверяемого программного обеспечения; последующие этапы тестирования ориентированы на обнаружение ошибок различного типа, которые не обязательно связаны с логикой программы.

4 Структурное тестирование

Поскольку исчерпывающее структурное тестирование невозможно, необходимо выбрать такие критерии его полноты, которые допускали бы их простую проверку и облегчали бы целенаправленный подбор тестов.

Наиболее слабым из критериев полноты структурного тестирования является требование хотя бы однократного выполнения (покрытия) каждого оператора программы. Более сильным критерием является так называемый

критерий C1: каждая ветвь алгоритма (каждый переход) должна быть пройдена (выполнена) хотя бы один раз.

Использование критерия покрытия условий может вызвать подбор тестов, обеспечивающих переход в программе, который пропускается при использовании критерия C1 (например, в программе на языке Паскаль, использующей конструкцию цикла WHILE x AND y DO... , применение критерия покрытия условий требует проверки обоих вариантов выхода из цикла: NOT x и NOT y). С другой стороны покрытие условий может не обеспечивать покрытия всех переходов.

Например, конструкция IF A AND B THEN... требует по критерию покрытия условий двух тестов (например, A=true, B=false и A=false, B=true), при которых может не выполняться оператор, расположенный в then - ветви оператора if. Практически единственным средством, предоставляемым современными системами программирования, является возможность определения частоты выполнения различных операторов программы. Но с помощью этого инструмента поддержки тестирования можно проверить выполнение только слабейшего из критериев полноты - покрытие всех операторов. Правда, с помощью этого же инструмента можно проверить и выполнение критерия C1. Но для этого предварительно текст программы должен быть преобразован таким образом, чтобы все конструкции условного выбора (IF, CASE или SWITCH) содержали ветви ELSE или DEFAULT, хотя бы и пустые. В этом случае все ветви алгоритма, не выполнявшиеся на данном тесте, будут “видимы” из таблицы частоты выполнения операторов преобразованной программы.

Актуальной остается задача создания инструментальных средств, позволяющих:

- накапливать информации о покрытых и непокрытых ветвях для всех использованных тестов;
- выделять разработчику еще не покрытые при тестировании участки программы, облегчая выбор следующих тестов;
- поддерживать более мощные критерии полноты структурного тестирования.

5 Совместимое тестирование модулей

Известны **два подхода к совместному тестированию модулей: пошаговое и монолитное тестирование.** При **монолитном** тестировании сначала по отдельности тестируются все модули программного комплекса, а затем все они объединяются в рабочую программу для комплексного тестирования. При **пошаговом** тестировании каждый модуль для своего тестирования подключается к набору уже проверенных модулей.

В первом случае для автономного тестирования каждого модуля требуется **модуль – драйвер (то есть вспомогательный модуль, имитирующий вызов тестируемого модуля и один или несколько модулей - заглушек то есть вспомогательных модулей, имитирующих работу модулей, вызываемых из тестируемого).** При **пошаговом** тестировании модули проверяются не изолированно друг от друга, поэтому требуются либо только драйверы, либо только заглушки. При сравнении **пошагового** и монолитного тестирования можно отметить следующие **преимущества** первого подхода:

- меньшая трудоемкость (при монолитном тестировании требуются 5 драйверов и 5 заглушек; при пошаговом тестировании требуются или только 5 драйверов - если модули подключаются “снизу вверх ”, - или только 5 заглушек - если модули подключаются “сверху вниз”);
- более раннее обнаружение ошибок в интерфейсах между модулями (их сборка начинается раньше, чем при монолитном тестировании);
- легче отладка, то есть локализация ошибок (они в основном связаны с последним из подключенных модулей);
- более совершенные результаты тестирования (более тщательная проверка совместного использования модулей).

При использовании **пошагового** тестирования возможны две стратегии подключения модулей: **нисходящая и восходящая.**

Нисходящее тестирование начинается с главного (или верхнего) модуля программы, а выбор следующего подключаемого модуля происходит из числа модулей, вызываемых из уже протестированных.

Одна из основных проблем, возникающих при нисходящем тестировании, - создание заглушек. Другая проблема, которую необходимо решать при нисходящем тестировании, - форма представления тестов в программе, так как, как правило, главный модуль получает входные данные не непосредственно, а через специальные модули ввода, которые при тестировании сначала заменяются заглушками. Для передачи в главный модуль разных тестов нужно или иметь несколько разных заглушек, или записать эти тесты в файл во внешней памяти и с помощью заглушки считывать их. Поскольку после тестирования главного модуля процесс проверки может продолжаться по-разному, следует придерживаться следующих правил:

а) модули, содержащие операции ввода-вывода, должны подключаться к тестированию как можно раньше;

б) критические (т.е. наиболее важные) для программы в целом модули также должны тестироваться в первую очередь.

Основные достоинства нисходящего тестирования: уже на ранней стадии тестирования есть возможность получить работающий вариант разрабатываемой программы; быстро могут быть выявлены ошибки, связанные с организацией взаимодействия с пользователем.

Проблемы, которые могут возникать при нисходящем тестировании: появляется соблазн совмещения нисходящего проектирования с тестированием, что, как правило, неразумно, т.к. проектирование - процесс итеративный и в нем неизбежен возврат на верхние уровни и исправление принятых ранее решений, что обесценивает результаты уже проведенного тестирования; может возникнуть желание перейти к тестированию модуля следующего уровня до завершения тестирования предыдущего по объективным причинам (необходимости создания нескольких версий заглушек, использования модулями верхнего уровня ресурсов модулей нижних уровней).

При восходящем тестировании проверка программы начинается с терминальных модулей (т.е. тех, которые не вызывают не каких других модулей программы). Эта стратегия во многом противоположна нисходящему тестированию (в частности, преимущества становятся недостатками и наоборот). Нет проблемы выбора следующего подключаемого модуля - учитывается лишь то, чтобы он вызывал только уже протестированные модули. В отличие от заглушек драйверы не должны иметь несколько версий, поэтому их разработка в большинстве случаев проще.

Другие достоинства восходящего тестирования: поскольку нет промежуточных модулей (тестируемый модуль является для рабочего варианта программы модулем самого верхнего уровня), нет проблем, связанных с возможностью или трудностью задания тестов; нет возможности совмещения проектирования с тестированием; нет трудностей, вызывающих желание перейти к тестированию следующего модуля, не завершив проверки предыдущего. Основными недостатком восходящего тестирования является то, что проверка всей структуры разрабатываемого программного комплекса возможна только на завершающей стадии тестирования. Хотя однозначного вывода о преимущества той или иной стратегии пошагового тестирования сделать нельзя (нужно учитывать конкретные характеристики тестируемой программы), в большинстве случаев более предпочтительным является восходящее тестирование.

6 Семантическая отладка

Ошибки этапа выполнения или семантические ошибки происходят, когда вы компилируете полную программу, которая при ее выполнении делает что-то недопустимое. То есть, программа содержит допустимые операторы, но при их выполнении что-то происходит неверно. Например, программа может пытаться открыть для ввода несуществующий файл или выполнить деление на ноль.

Семантическая отладка - это процесс нахождения и исправления ошибок, связанных с неправильным указанием логических страниц данных.

Существует 3 способа отладки программы:

1. **Пошаговая отладка программ с заходом в подпрограммы;**

2. **Пошаговая отладка программ с выполнением подпрограммы как одного оператора;**

3. **Выполнение программы до точки остановки.**

Пошаговая отладка программ заключается в том, что выполняется один оператор программы и, затем контролируются те переменные, на которые должен был воздействовать данный оператор.

Если в программе имеются уже отлаженные подпрограммы, то подпрограмму можно рассматривать, как один оператор программы и воспользоваться вторым способом отладки программ.

Если в программе существует достаточно большой участок программы, уже отлаженный ранее, то его можно выполнить, не контролируя переменные, на которые он воздействует. Использование точек остановки позволяет пропускать уже отлаженную часть программы. Точка остановки устанавливается в местах, где необходимо проверить содержимое переменных или просто проконтролировать, передаётся ли управление данному оператору.

7 Документирование программы

Последней составляющей процесса программирования является документирование. Оно включает широкий спектр описаний, облегчающих процесс программирования и обогащающих результирующую программу. Постоянное документирование должно составлять неотъемлемую часть каждого шага программирования. Постановка задачи, проектные документы, алгоритмы и программы – все это документы. Внутренняя документация, включенная непосредственно в программу, облегчает чтение кода. Назначение учебного пособия (еще одной формы документации) – научить пользователя применять новую программу; справочное руководство позволяет ознакомиться с описанием команд программного обеспечения.

При разработке программы создается большой объем разнообразной документации. Она необходима как средство передачи информации между разработчиками программы, как средство управления разработкой программы и как средство передачи пользователям информации, необходимой для применения и сопровождения программы.

Пример. Система тестов для задачи нахождения корней квадратного уравнения:

Квадратное уравнение — это уравнение вида

$ax^2 + bx + c = 0$, где **a** не равно **0**.

Для решения квадратного уравнения можно использовать формулы:

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{и} \quad x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a},$$

где $D = b^2 - 4ac$ — дискриминант многочлена $ax^2 + bx + c$.
Если $D > 0$, то уравнение имеет два различных вещественных корня.
Если $D = 0$, то оба корня вещественны и равны.
Если $D < 0$, то оба корня являются комплексными числами.

Номер теста	Проверяемый случай	Коэффициенты			Результаты
		a	b	c	
1	d >0	1	1	-2	x1 = 1, x2 = - 2
2	d=0	1	2	1	Корни равны: x1 = - 1, x2 = - 1
3	d < 0	2	1	2	Действительных корней нет
4	a=0, b=0, c=0	0	0	0	Все коэффициенты равны нулю. x — любое число.
5	a=0, b=0, c<>0	0	0	2	Неправильное уравнение
6	a=0, b<>0	0	2	1	Линейное уравнение. Один корень: x = - 0,5
7	a <> 0, b <> 0, c = 0	2	1	0	x1 = 0, x2 = - 0,5

КОМПЛЕКТ ЗАДАНИЙ НА КОНТРОЛЬНУЮ РАБОТУ
для студентов заочной формы обучения

Тема контрольной работы: **«Проектирование теста для отладки программы»** одинаковая для всех студентов и различается только индивидуальными исходными данными.
Каждое задание является индивидуальным.
Контрольная работа содержит ***пояснительную записку, в которой приводится:***

1. **Текст программы**
2. **Тесты**
3. **Результаты тестирования.**

Исходными данными является словесное описание некоторой процедуры обработки данных.

Процесс тестирования содержит три этапа:

1. Тестирование на основе данных, **которые характерны для реальных условий** функционирования программы

2. Тестовые данные включают **граничные значения** области изменения входных переменных, которые должны восприниматься программой как правильные данные. Типичными примерами таких значений являются очень маленькие или очень большие числа и отсутствие данных. Еще один тип экстремальных условий — это **граничные объемы** данных, когда массивы состоят из слишком малого или слишком большого числа элементов.

3. Тестирование с использованием данных, значения которых лежат **за пределами допустимой области изменений.** Программа должна сама отвергать любые данные, которые она не в состоянии обработать правильно.

Тестовые данные должны обеспечить проверку всех возможных условий возникновения ошибок:

- должна быть испытана **каждая ветвь** алгоритма;
- очередной тестовый прогон должен контролировать нечто такое, что еще **не было проверено** на предыдущих прогонах;
- первый тест должен быть **максимально прост**, чтобы проверить, работает ли программа вообще;
- арифметические операции в тестах должны **предельно упрощаться** для уменьшения объема вычислений;
- количества элементов последовательностей, точность для итерационных вычислений, количество проходов цикла в тестовых примерах должны задаваться из соображений **сокращения объема вычислений;**

- минимизация вычислений не должна снижать надежности контроля;
- тестирование должно быть **целенаправленным и систематизированным**, так как случайный выбор исходных данных привел бы к трудностям в определении ручным способом ожидаемых результатов; кроме того, при случайном выборе тестовых данных могут оказаться непроверенными многие ситуации;
- **усложнение** тестовых данных **должно происходить постепенно.**

Индивидуальные задания для контрольной работы

Номер задания соответствует номеру студента в списке группы.

1. Найдите наибольший общий делитель двух заданных целых чисел.

2. Найдите наименьшее общее кратное двух заданных целых чисел.

3. Определите, является ли заданное число нечетным двузначным числом.

4. Заданы площади квадрата и круга. Определите, поместится ли квадрат в круге.

5. Решите биквадратное уравнение.

6. Найдите среднее арифметическое положительных элементов заданного одномерного массива.

7. Элементы заданного одномерного массива разделите на его первый элемент.

8. Определите, лежит ли заданная точка на одной из сторон треугольника, заданного координатами своих вершин.

9. Определите, имеют ли общие точки две плоские фигуры — треугольник с заданными координатами его вершин и круг заданного радиуса с центром в начале координат.

10. Задано целое $A > 1$. Найдите наименьшее целое неотрицательное k, при котором $2^k > A$.

11. Дана последовательность целых чисел. Определите, со скольких чётных чисел она начинается.

12. В заданном двумерном массиве найдите количество строк, не содержащих нули.

13. Определите, сколько строк заданного двумерного массива содержат элементы из заданного диапазона.

14. Преобразуйте число, заданное в римской системе счисления, в число десятичной системы.

15. В заданном двумерном массиве найдите сумму элементов диагонали, не содержащих нули.

16. В заданном двумерном массиве найдите сумму элементов , не содержащих нули и имеющих значение больше некоторой заданной величины M мин.

17. В заданном двумерном массиве найдите сумму элементов , не содержащих нули и имеющих значение меньше некоторой заданной величины M мах.

18. В заданном двумерном массиве найдите сумму элементов , не содержащих нули и имеющих значение больше некоторой заданной величины M мин но меньше некоторой заданной величины M мах..

19. В заданном двумерном массиве найдите процентное отношение каждого элемента в сумме элементов , не содержащих нули.

20. В текстовом файле провести проверку корректности данных, если известно, что это должны бать только целые положительные числа.

21. В текстовом файле провести проверку корректности данных, если известно, что это должны бать только слова определенной длинны.

22. В текстовом файле провести проверку корректности данных, если известно, что это должны бать только слова из букв латинского алфавита.

23. В текстовом файле провести проверку корректности данных, если известно, что это должны бать только слова из букв кириллицы.

24. В текстовом файле провести проверку корректности данных, если известно, что это должны бать только слова из заданного набора символов.

П 24. может иметь множество модификаций в зависимости от заданного набора символов