Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Оренбургский государственный университет»

Кафедра программного обеспечения вычислительной
техники и автоматизированных систем

Т.М. Зубкова

ОТЛАДКА И ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Рекомендовано к изданию Редакционно-издательским советом федерального государственного бюджетного образовательного учреждения высшего образования «Оренбургский государственный университет» в качестве методических указаний для студентов, обучающихся по программам высшего образования по направлению подготовки 09.03.04 Программная инженерия

Оренбург

2016

УДК 681.3 (07)
ББК 32.973.26-018я73
З 91

Рецензент – профессор, доктор экономических наук В.Н. Шепель

Зубкова, Т.М.
З 91 Отладка и тестирование программного обеспечения: методические указания к лабораторным занятиям по дисциплине «Тестирование программного обеспечения» /Т.М. Зубкова; Оренбургский гос. ун-т.- Оренбург: ОГУ, 2016. – 34 с.

Методические указания для выполнения лабораторных работ по дисциплине «Тестирование программного обеспечения» предназначены для оказания помощи студентам при выполнении индивидуальных заданий. Данная дисциплина входит в базовую часть профессионального цикла дисциплин бакалавров очной формы обучения по направлению 09.03.04 – «Программная инженерия» с профилем подготовки «Разработка программно-информационных систем».

В методических указаниях изложены задания, теоретические основы для их выполнения.

УДК 681.3 (07)
ББК 32.973.26-018я73

Содержание

Введение

Основная **цель** освоения дисциплины «Тестирование программного обеспечения» ООП ВПО по направлению подготовки 09.03.04 «Программная инженерия» заключается в создании теоретической основы для проведения всех видов отладки, тестирования и испытания программного обеспечения.

Основными **задачами**, решаемыми в процессе освоения дисциплины, являются:

- обучить основным методам, способам и принципам тестирования программных средств;
- обучить проектированию и выполнению комплексных тестов программных средств;
- обучить проведению испытаний надежности сложных программных средств;
- обучить составлению протоколов и отчетов по проведенному тестированию программных средств.

Процесс изучения дисциплины направлен на формирование элементов следующих компетенций в соответствии с ФГОС ВПО и ООП ВПО по данному направлению подготовки 09.03.04 «Программная инженерия»:

а) общекультурных (ОК):

- владеть культурой мышления, способность к обобщению, анализу, восприятию информации, постановке цели и выбору путей ее достижения (ОК-1);
- готов к кооперации с коллегами, работе в коллективе (ОК-3);
- способен находить организационно-управленческие решения в нестандартных ситуациях и готов нести за них ответственность (ОК-4);
- умеет использовать нормативно-правовые документы в своей деятельности (ОК-5);
- стремится к саморазвитию, повышению своей квалификации и мастерства (ОК-6);

- умеет критически оценивать свои достоинства и недостатки, наметить пути и выбрать средства развития достоинств и устранения недостатков (ОК-7).

б) профессиональных (ПК):

- понимание основных концепций, принципов, теорий и фактов, связанных с информатикой (ПК-1);
- способность к формализации в своей предметной области с учетом ограничений используемых методов исследования (ПК-2);
- готовность к использованию методов и инструментальных средств исследования объектов профессиональной деятельности (ПК-3);
- умение готовить презентации, оформлять научно-технические отчеты по результатам выполненной работы, публиковать результаты исследований в виде статей и докладов на научно-технических конференциях (ПК-5);
- умение применять основы информатики и программирования к проектированию, конструированию и тестированию программных продуктов (ПК-10);
- понимание методов управления процессами разработки требований, оценки рисков, приобретения, проектирования, конструирования, тестирования, эволюции и сопровождения (ПК-23);

В результате освоения дисциплины обучающийся должен:

Знать:

- основные факты, концепции, принципы и теории, связанные с информатикой;
- теоретические основы архитектуры и программной организации вычислительных и информационных систем;
- формальные методы, технологии и инструменты разработки программного продукта;
- основы верификации и аттестации программного обеспечения.

Уметь:

- инсталлировать, тестировать, испытывать и использовать программные средства;

- работать с современными системами программирования.

Владеть:

- языками процедурного объектно-ориентированного программирования;
- навыками разработки и отладки программ на алгоритмических языках программирования;
- методами и средствами разработки и оформления технической документацией.

Приобрести опыт деятельности в проведение тестирования отладки и испытание программного обеспечения.

1 Теоретические основы курса «Тестирование программного обеспечения»

1.1 Основные пути борьбы с ошибками

В процессе создания программы программист старается предвидеть все возможные ситуации. Этап тестирования является последней попыткой определить надежность и корректность программы. Проверка надежности включает в себя просмотр проектной документации, анализ текста программы, тестирование, и наконец, демонстрацию заказчику того, что программа работает надежно.

Дейкстра выделяет три интеллектуальные возможности человека, используемые при разработке программных систем (ПС):

- способность к перебору;
- способность к абстракции;
- способность к математической индукции.

Способность человека к перебору связана с возможностью последовательного переключения внимания с одного предмета на другой, позволяя *узнавать* искомый предмет. Эта способность ограничена – в среднем человек может уверенно (не сбиваясь) перебирать в пределах 1000 предметов (элементов). Человек должен научиться действовать с учетом этой ограниченности. Средством преодоления этой ограниченности является его способность к абстракции, благодаря которой человек может объединять разные предметы или экземпляры в одно понятие, заменять множество элементов одним элементом (другого рода). Способность человека к математической индукции позволяет ему справляться с бесконечными последовательностями.

При разработке программного обеспечения (ПО) человек имеет дело с системами. Под *системой* понимают совокупность взаимодействующих (находящихся в отношениях) друг с другом элементов. ПС можно рассматривать как пример системы. Понять систему – значит осмысленно перебрать все пути взаимодействия между ее элементами. В силу ограниченности человека к пере-

бору различают простые и сложные системы. Под *простой* понимают такую систему, в которой человек может уверенно перебирать все пути взаимодействия между ее элементами, а в *сложной* соответственно нет.

Учитывая особенности человека можно указать следующие пути борьбы с ошибками:

- сужение пространства перебора (упрощение создаваемых систем);
- обеспечение требуемого уровня подготовки разработчика (это функции менеджеров коллектива разработчиков);
- обеспечение однозначности интерпретации представления информации;
- контроль правильности перевода (включая и контроль однозначности интерпретации).

Причины появления ошибок:

- технологические, включающие возможности описание задачи, ее решения;
- организационные, включающие распределение рабочей нагрузки, доступность информации, средств связи и ресурсов;
- психологические, включающие желание сотрудничать, распределение ролей, микроклимат в коллективе;
- индивидуальные, включающие квалификацию, талант, личные качества сотрудничающих программистов.

1.2 Основные понятия и принципы тестирования ПО

Отладка ПС – это деятельность, направленная на обнаружение и исправление ошибок с использованием процессов выполнения программ.

Тестирование ПС – это процесс выполнения программ на некотором наборе данных, для которого заранее известен результат или известны правила поведения этих программ. Указанный набор данных называется *тестовым* или просто *тестом*.

Таким образом, отладку можно представить в виде многократного повторения трех процессов: тестирования, в результате которого может быть констатировано наличие в ПС ошибки, поиска места ошибки в программах и документации, редактирования программ и документации с целью устранения обнаруженной ошибки. Другими словами:

Отладка = Тестирование + Поиск ошибок + Редактирование

Отладка имеет место тогда, когда программа со всей очевидностью работает неправильно. Поэтому отладка начинается всегда в предвидении отказа программы. Если же оказывается, что программа работает верно, то она тестируется. Часто случается так, что после прогона тестов программа вновь подвергается отладке. Таким образом, тестирование устанавливает факт наличия ошибки, а отладка выявляет ее причину.

Основная цель выделения отладки и тестирования как отдельных этапов создания программы заключается в том, чтобы обратить внимание обязательности обеих стадий и на необходимость специального планирования временных затрат по каждой из них в отдельности.

Нельзя гарантировать, что тестированием можно установить наличие каждой имеющейся в ПС ошибки. Поэтому возникает две задачи. Первая задача: подготовить такой набор тестов и применить к ним ПС, чтобы обнаружить в нем по возможности большее число ошибок. Однако чем дольше продолжается процесс тестирования (и отладки в целом), тем большей становится стоимость ПС. Отсюда вторая задача: определить момент окончания отладки ПС (или отдельной его компоненты). Признаком возможности окончания отладки является полнота охвата пропущенными через ПС тестами, множество различных ситуаций, возникающих при выполнении программ ПС и относительно редкое проявление ошибок на последнем отрезке процесса тестирования. Последнее определяется в соответствии с требуемой степенью надежности ПС, указанной в спецификации качества.

Тестирование проводиться с целью обнаружения ошибок. Каждый тест определяет:

- свой набор исходных данных и условий для запуска программы;
- набор ожидаемых результатов работы программы.

Тестирование обеспечивает:

- обнаружение ошибок;
- демонстрацию соответствия функций программы ее назначению;
- демонстрацию реализации требований к характеристикам программы;
- отображение надежности как индикатора качества программы.

Тестирование не может показать отсутствия дефектов (оно может показывать только присутствие дефектов). Необходимо помнить это при проведении тестирования.

Рассмотрим информационные потоки процесса тестирования. Они показаны на рисунке 1.

Рисунок 1 – Информационные потоки процесса тестирования

На входе процесса тестирования два потока:

- текст программы;
- исходные данные для запуска программы.

На выходе – ожидаемые результаты.

При выполнении тестов все полученные результаты оцениваются. Это означает, что реальные результаты тестов сравниваются с ожидаемыми результатами. Если обнаруживается несовпадение, фиксируется ошибка – начинается отладка. Процесс отладки непредсказуем по времени. На поиск места дефекта и исправление может потребоваться час, день, месяц. Неопределенность в отладке приводит к большим трудностям в планировании работ по созданию ПС в целом.

После сбора результатов тестирования начинается оценка качества и надежности ПО. Если регулярно встречаются серьезные ошибки, требующие проектных изменений, то качество и надежность ПО не удовлетворительные, констатируется необходимость усиления тестирования. С другой стороны, если функции ПО реализованы правильно, а обнаруженные ошибки легко исправляются, может быть сделан один из двух выводов:

- качество и надежность ПО удовлетворительны;
- тесты не способны обнаруживать серьезные ошибки.

Если тесты не обнаруживают ошибок, то возможно, что тестовые варианты не достаточно продуманы и в ПО есть скрытые ошибки. Такие ошибки будут обнаруживаться пользователями и корректироваться разработчиком на этапе сопровождения (стоимость исправления возрастает в 60-100 раз по сравнению с этапом разработки).

Результаты, накопленные в ходе тестирования, могут оцениваться и более формальным способом. Для этого используют модели надежности ПО, выполняющие прогноз надежности по реальным данным об интенсивности ошибок.

Как правило, тестирование программы выполняется поэтапно, начиная с проверки каждого модуля и заканчивая проверкой всей системы в целом. Если при этом не придерживаться какой-либо четкой последовательности действий, то нельзя надеяться на создании надежного ПО.

1.3 Тестирование интеграции

Тестирование интеграции поддерживает сборку цельной программной системы.

Цель сборки и тестирования интеграции: взять модули, протестированные как элементы, и построить программную структуру, требуемую проектом.

Тесты проводятся для обнаружения ошибок интерфейса. Некоторые категории ошибок интерфейса:

- потеря данных при прохождении через интерфейс;
- отсутствие в модуле необходимой ссылки;
- неблагоприятное влияние одного модуля на другой;
- подфункции при объединении не образуют требуемую главную функцию;
- отдельные (допустимые) неточности при интеграции выходят за допустимый уровень;
- проблемы при работе с глобальными структурами данных.

Существует два варианта тестирования, поддерживающих процесс интеграции: нисходящее тестирование и восходящее тестирование.

Нисходящее и восходящее тестирование

При нисходящем проектировании ПС «скелет» программы постепенно «обрастает» новыми модулями, также должны добавляться и новые тестовые данные, объем которых увеличивается постепенно одновременно с разрастанием программы. Преимуществом тестирования сверху вниз является то, что стержневая логика программы тестируется на раннем этапе, и эта проверка повторяется многократно с добавлением новых модулей. При тестировании же снизу вверх стержневая логика программы испытывается в последнюю очередь. Немаловажным преимуществом метода тестирования сверху вниз является распределенное тестирование, проводимое фактически на протяжении всей разработки проекта.

В данном подходе модули объединяются движением сверху вниз по управляющей иерархии, начиная от главного управляющего модуля. Подчиненные модули добавляются в структуру или в результате поиска в глубину, или в результате поиска в ширину.

Рассмотрим пример, приведенный на рисунке 2. Интеграция поиском в глубину будет подключать все модули, находящиеся на главном управляющем пути структуры (по вертикали). Выбор главного управляющего пути отчасти произволен и зависит от характеристик, определяемых приложением. Например, при выборе левого пути прежде всего будут подключены модули Ml, M2, M5. Следующим подключается модуль M8 или M6 (если это необходимо для правильного функционирования M2). Затем строится центральный или правый управляющий путь.

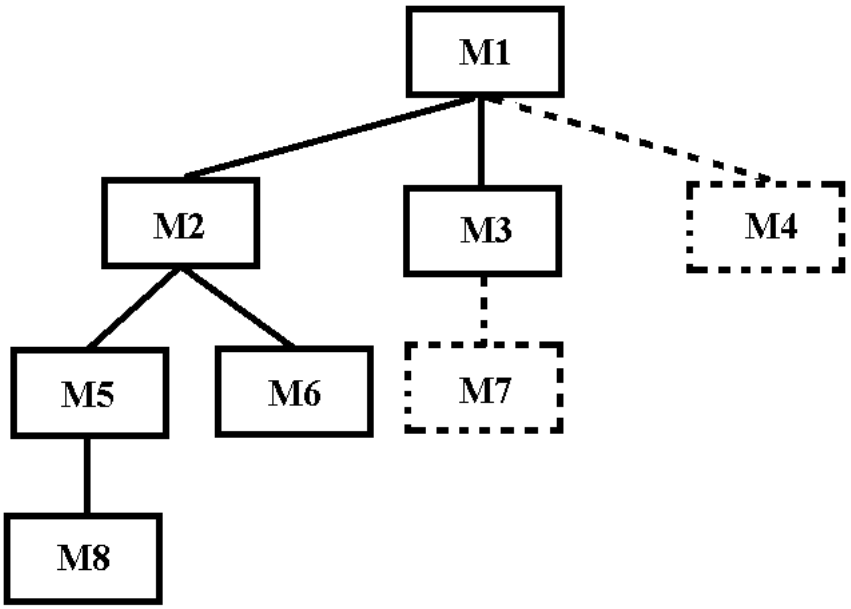

Рисунок 2 – Нисходящая интеграция системы

При интеграции поиском в ширину структура последовательно проходится по уровням-горизонталям. На каждом уровне подключаются модули, непосредственно подчиненные управляющему модулю – «начальнику». В этом случае прежде всего подключаются модули M2, M3, M4. На следующем уровне – модули M5, M6 и т. д.

Возможные шаги процесса нисходящей интеграции.

1. Главный управляющий модуль (находится на вершине иерархии) используется как тестовый драйвер. Все непосредственно подчиненные ему модули временно замещаются заглушками.

2. Одна из заглушек заменяется реальным модулем. Модуль выбирается поиском в ширину или в глубину.

3. После подключения каждого модуля (и установки на нем заглушек) проводится набор тестов, проверяющих полученную структуру.

4. Если в модуле-драйвере уже нет заглушек, производится смена модуля-драйвера (поиском в ширину или в глубину).

5. Выполняется возврат на шаг 2 (до тех пор, пока не будет построена целая структура).

Достоинство нисходящей интеграции: ошибки в главной, управляющей части системы выявляются в первую очередь.

Недостаток: трудности в ситуациях, когда для полного тестирования на верхних уровнях нужны результаты обработки с нижних уровней иерархии.

Существуют 3 возможности борьбы с этим недостатком:

1) откладывать некоторые тесты до замещения заглушек модулями;

2) разрабатывать заглушки, частично выполняющие функции модулей;

3) подключать модули движением снизу вверх.

Первая возможность вызывает сложности в оценке результатов тестирования.

Для реализации второй возможности выбирается одна из следующих категорий заглушек:

- заглушка A – отображает трассируемое сообщение;
- заглушка B – отображает проходящий параметр;
- заглушка C – возвращает результат;
- заглушка D – выполняет поиск по ключу (входному параметру) и возвращает связанный с ним выходной параметр.

Категории заглушек представлены на рисунке 3.

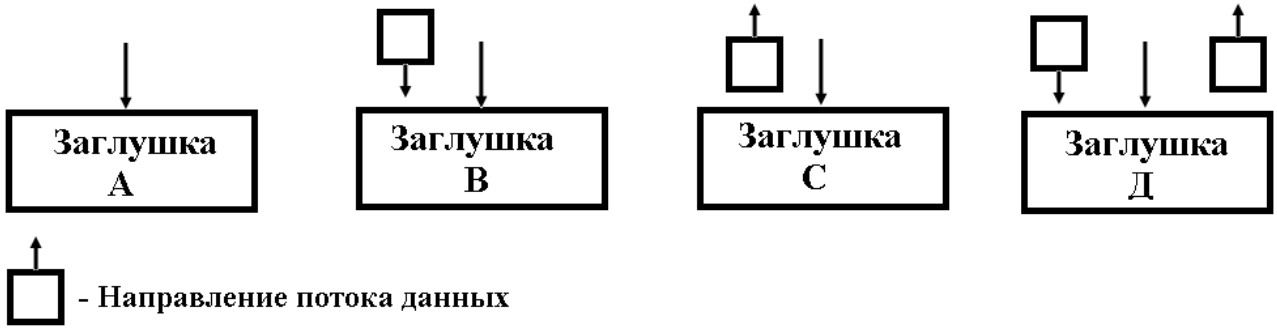

Рисунок 3 – Категории заглушек

Очевидно, что заглушка A наиболее простая, а заглушка D наиболее сложная в реализации.

Этот подход работоспособен, но может привести к существенным затратам, так как заглушки становятся все более сложными.

При восходящем тестировании интеграции сборка и тестирование системы начинаются с модулей-атомов, располагаемых на нижних уровнях иерархии. Модули подключаются движением снизу вверх. Подчиненные модули всегда доступны, и нет необходимости в заглушках.

Рассмотрим шаги методики восходящей интеграции.

1. Модули нижнего уровня объединяются в кластеры (группы, блоки), выполняющие определенную программную подфункцию.

2. Для координации вводов-выводов тестового варианта пишется драйвер, управляющий тестированием кластеров.

3. Тестируется кластер.

4. Драйверы удаляются, а кластеры объединяются в структуру движением вверх. Пример восходящей интеграции системы приведен на рисунке 4.

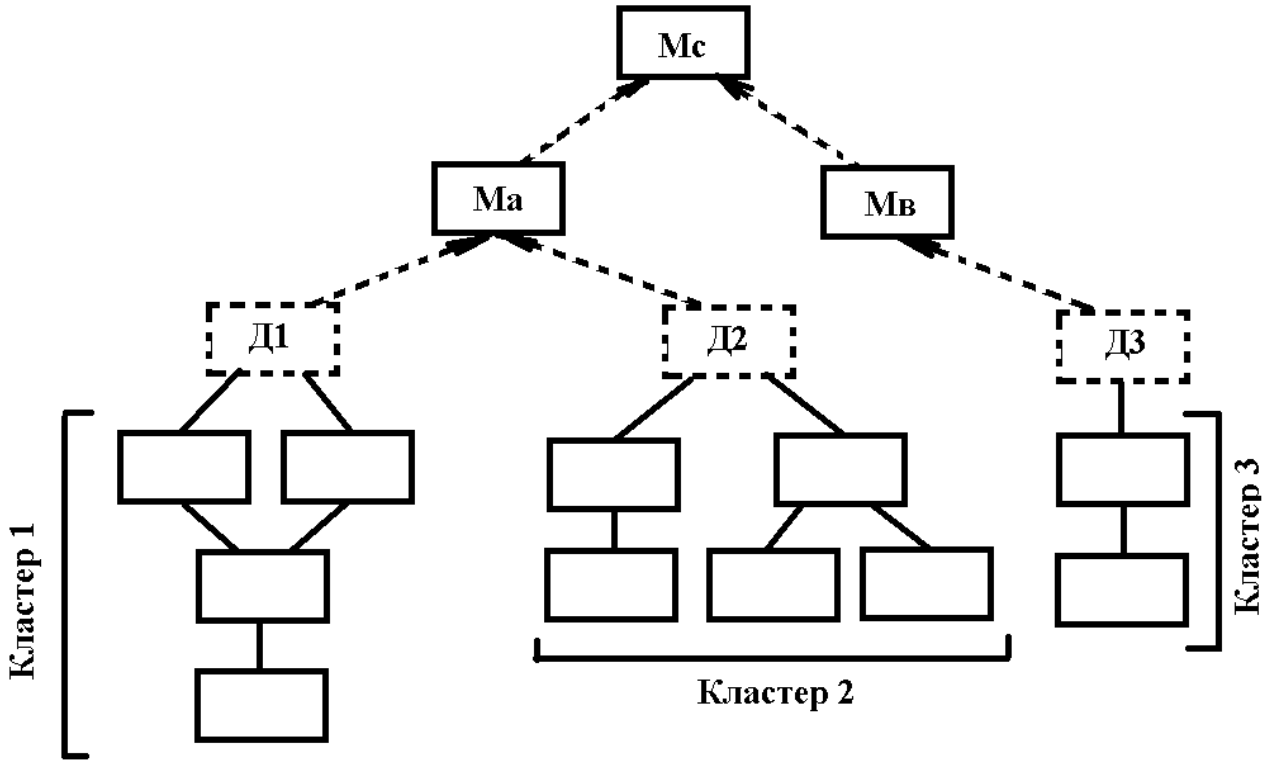

Рисунок 4 – Восходящая интеграция системы

Модули объединяются в кластеры 1,2,3. Каждый кластер тестируется драйвером. Модули в кластерах 1 и 2 подчинены модулю Ма, поэтому драйверы Д1 и Д2 удаляются и кластеры подключают прямо к Ма. Аналогично драйвер Д3 удаляется перед подключением кластера 3 к модулю Мв. В последнюю очередь к модулю Мс подключаются модули Ма и Мв.

Рассмотрим различные типы драйверов:

- драйвер A – вызывает подчиненный модуль;
- драйвер B – посылает элемент данных (параметр);
- драйвер C – отображает параметр из подчиненного модуля;
- драйвер D – является комбинацией драйверов B и C.

Очевидно, что драйвер A наиболее прост, а драйвер D наиболее сложен в реализации. Различные типы драйверов представлены на рисунке 5.

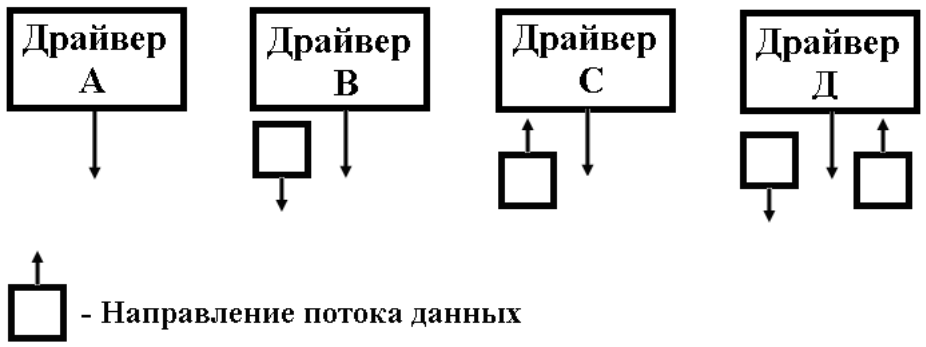

Рисунок 5 –Типы драйверов

По мере продвижения интеграции вверх необходимость в выделении драйверов уменьшается. Как правило, в двухуровневой структуре драйверы не нужны.

Различают следующие уровни тестирования:

• *модульное тестирование*. Тестируется минимально возможный для тестирования компонент, например отдельный класс или функция;

• *интеграционное тестирование*. Проверяется, есть ли какие-либо проблемы в интерфейсах и взаимодействии между интегрируемыми компонентами, например, не передается информация, передается некорректная информация;

• *системное тестирование.* Тестируется интегрированная система на ее соответствие исходным требованиям:

– *альфа-тестирование* – имитация реальной работы с системой штатными разработчиками либо реальная работа с системой потенциальными пользователями/заказчиком на стороне разработчика. Часто альфа-тестирование применяется для законченного продукта в качестве внутреннего приемочного 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 ПО;

– *бета-тестирование* – в некоторых случаях выполняется распространение версии с ограничениями (по функциональности или времени работы) для некоторой 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.

1.4 Классификация ошибок

Виды ошибок и способы их обнаружения приведены в таблице 1.

Таблица 1 – Виды программных ошибок и способы их обнаружения

Виды программных ошибок	**Способы обнаружения**
Синтаксические	Статический контроль и диагностика компиляторами и компоновщиком
Ошибки выполнения, выявляемые автоматически: а) переполнение, защита памяти; б) несоответствие типов; в) зацикливание	Динамический контроль: аппаратурой процессора; run-time системы программирования; операционной системой – по прерыванию лимита времени
Программа не соответствует спецификации	Целенаправленное тестирование
Спецификация не соответствует требованиям	Испытания, бета-тестирование

Классификация ошибок, возникающих при выполнении программ, приведена на рисунке 6.

Рисунок 6 – Классификация ошибок

Заповеди по отладке программных средств, предложенные Майерсом.

Заповедь 1. Считайте тестирование ключевой задачей разработки ПС, поручайте его самым квалифицированным и одаренным программистам. Нежелательно тестировать свою собственную программу.

Заповедь 2. Хорош тот тест, для которого высока вероятность обнаружить ошибку, а не тот, который демонстрирует правильную работу программы.

Заповедь 3. Готовьте тесты как для правильных, так и для неправильных данных.

Заповедь 4. Документируйте пропуск тестов через компьютер, детально изучайте результаты каждого теста, избегайте тестов, пропуск которых нельзя повторить.

Заповедь 5. Каждый модуль подключайте к программе только один раз, никогда не изменяйте программу, чтобы облегчить ее тестирование.

Заповедь 6. Пропускайте заново все тесты, связанные с проверкой работы какой-либо программы ПС или ее взаимодействия с другими программами, если в нее были внесены изменения (например, в результате устранения ошибки).

1.5 Тестирование по методу «белого ящика» и «черного ящика»

Существуют 2 принципа тестирования программы:

- функциональное тестирование (тестирование «черного ящика»);
- структурное тестирование (тестирование «белого ящика»).

При тестировании «белого ящика» (англ. white-box testing, также говорят – прозрачного ящика) разработчик теста имеет доступ к исходному коду 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 определенной степени.

Известна: внутренняя структура программы.

Исследуются: внутренние элементы программы и связи между ними (рисунок 7).

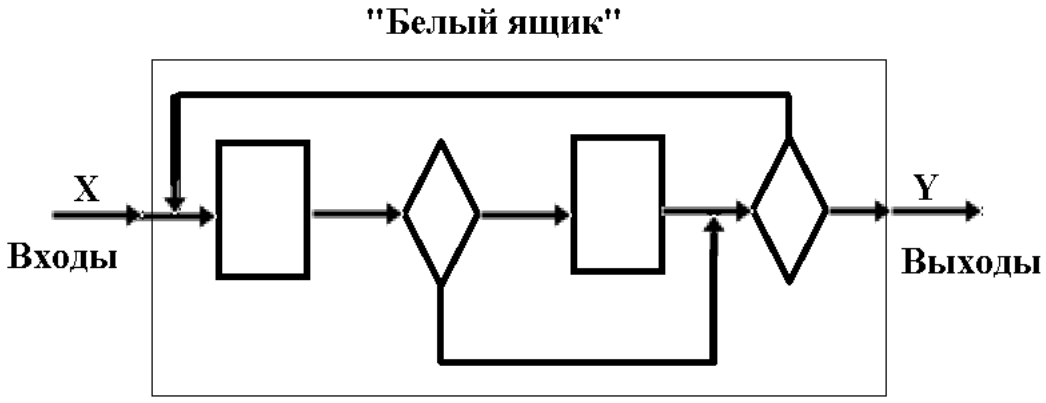

Рисунок 7 – Тестирование «белого ящика»

Объектом тестирования здесь является не внешнее, а внутреннее поведение программы. Проверяется корректность построения всех элементов программы и правильность их взаимодействия друг с другом. Обычно анализируются управляющие связи элементов, реже – информационные связи. Тестирование по принципу «белого ящика» характеризуется степенью, в какой тесты выполняют или покрывают логику (исходный текст) программы.

Особенности тестирования «белого ящика»

Обычно тестирование «белого ящика» основано на анализе управляющей структуры программы. Программа считается полностью проверенной, если проведено исчерпывающее тестирование маршрутов (путей) ее графа управления.

В этом случае формируются тестовые варианты, в которых:

- гарантируется проверка всех независимых маршрутов программы;
- проходятся ветви True, False для всех логических решений;
- выполняются все циклы (в пределах их границ и диапазонов);
- анализируется правильность внутренних структур данных.

При тестировании «белого ящика» проводят тесты:

- тестирование базового пути;
- тестирования условий;
- тестирование ветвей и операторов отношений;
- тестирования потоков данных;
- тестирование циклов.

Тестирование по методу «черного ящика»

При тестировании «черного ящика» (англ. black-box testing) 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, что эти события вызывают тот же отклик, что и реальные нажатия клавиш и кнопок мыши.

Известны: функции программы.

Исследуется: работа каждой функции на всей области определения.

Как показано на рисунке 8, основное место приложения тестов «черного ящика» – интерфейс ПО.

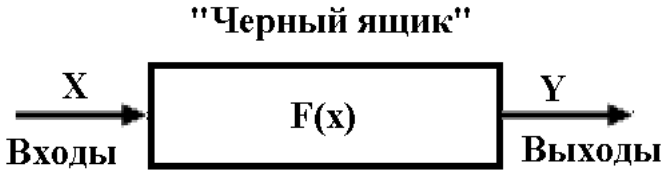

Рисунок 8 – Тестирование «черного ящика»

Тесты по методу «черного ящика» демонстрируют:

- как выполняются функции программ;
- как принимаются исходные данные;
- как вырабатываются результаты;
- как сохраняется целостность внешней информации.

При тестировании «черного ящика» рассматриваются системные характеристики программ, игнорируется их внутренняя логическая структура. Исчерпывающее тестирование, как правило, невозможно. Например, если в программе 10 входных величин и каждая принимает по 10 значений, то потребуется 10^{10} тестовых вариантов. Тестирование «черного ящика» не реагирует на многие особенности программных ошибок.

Тестирование «черного ящика» (функциональное тестирование) позволяет получить комбинации входных данных, обеспечивающих полную проверку всех функциональных требований к программе. При таком подходе желательно иметь:

- набор, образуемый такими входными данными, которые приводят к аномалиям поведения программы (назовем его *I T);*

- набор, образуемый такими выходными данными, которые демонстрируют дефекты программы (назовем его *OT)*.

Как показано на рисунке 9, любой способ тестирования «черного ящика» должен:

- выявить такие входные данные, которые с высокой вероятностью принадлежат набору *IT*;
- сформулировать такие ожидаемые результаты, которые с высокой вероятностью являются элементами набора *OT*.

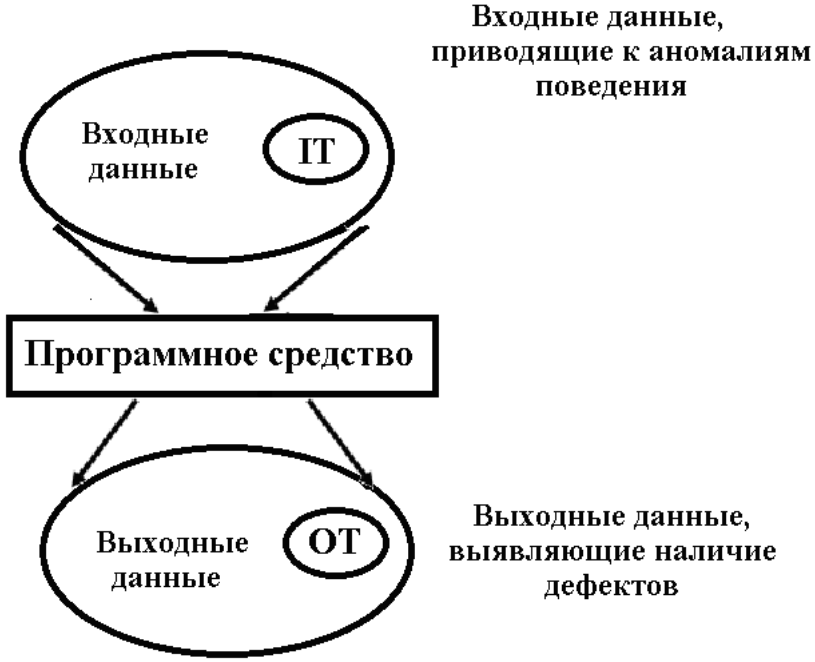

Рисунок 9 – Тестирование по методу «черного ящика»

Во многих случаях определение таких тестовых вариантов основывается на предыдущем опыте инженеров тестирования. Они используют свое знание и понимание области определения для идентификации тестовых вариантов, которые эффективно обнаруживают дефекты.

Принцип «черного ящика» не альтернативен принципу «белого ящика» – это дополняющий подход, который обнаруживает другой класс ошибок.

Тестирование «черного ящика» обеспечивает поиск следующих категорий ошибок:

- некорректные или отсутствующие функции;

- ошибки интерфейса;
- ошибки во внешних структурах данных или в доступе к внешней базе данных;
- ошибки характеристик (необходимая емкость памяти и т. д.);
- ошибки инициализации и завершения.

Подобные категории ошибок способами «белого ящика» не выявляются.

В отличие от тестирования «белого ящика», которое выполняется на ранней стадии процесса тестирования, тестирование «черного ящика» применяют на поздних стадиях тестирования. При тестировании «черного ящика» пренебрегают управляющей структурой программы, а концентрируют внимание на информационной области определения программной системы.

Техника «черного ящика» ориентирована на решение следующих задач:

- сокращение необходимого количества тестовых вариантов (из-за проверки не статических, а динамических аспектов системы);
- выявление классов ошибок, а не отдельных ошибок.

Способ разбиения по эквивалентности

Разбиение по эквивалентности – самый популярный способ тестирования «черного ящика».

В этом способе входная область данных программы *делится* на классы эквивалентности. Для каждого класса эквивалентности разрабатывается один тестовый вариант.

Класс эквивалентности – набор данных с общими свойствами. Обрабатывая разные элементы класса, программа должна вести себя одинаково. То есть при обработке любого набора из класса эквивалентности в программе задействуется один и тот же набор операторов (и связей между ними).

На рисунке 10 каждый класс эквивалентности показан эллипсом. Выделены входные классы эквивалентности допустимых и недопустимых исходных данных, а также классы результатов.

Рисунок 10 – Разбиение по эквивалентности

Классы эквивалентности могут быть определены по спецификации на программу.

Например, если спецификация задает в качестве допустимых входных величин 5-разрядные целые числа в диапазоне 15 000...70 000, то класс эквивалентности допустимых ИД (исходных данных) включает величины от 15 000 до 70 000, а два класса эквивалентности недопустимых ИД составляют:

- числа меньшие, чем 15 000;
- числа большие, чем 70 000.

Класс эквивалентности включает множество значений данных, допустимых или недопустимых по условиям ввода.

Условие ввода может задавать:

- определенное значение;
- диапазон значений;
- множество конкретных величин;
- булево условие.

Способ анализа граничных значений

Как правило, большая часть ошибок происходит на границах области ввода, а не в центре. Анализ граничных значений заключается в получении те-

стовых вариантов, которые анализируют граничные значения. Данный способ тестирования дополняет способ разбиения по эквивалентности.

Основные отличия анализа граничных значений от разбиения по эквивалентности:

1) тестовые варианты создаются для проверки только ребер классов эквивалентности;

2) при создании тестовых вариантов учитывают не только условия ввода, но и область вывода.

1.6 Методика тестирования программных систем

Процесс тестирования объединяет различные способы тестирования в спланированную последовательность шагов, которые приводят к успешному построению программной системы. Методика тестирования ПС может быть представлена в виде разворачивающейся спирали (рисунок 11).

Риснок 11 – Спираль процесса тестирования ПС

В начале осуществляется *тестирование элементов (модулей),* проверяющее результаты этапа *кодирования* ПС. На втором шаге выполняется *тестирование интеграции,* ориентированное на выявление ошибок этапа *проектирования* ПС. На третьем обороте спирали производится *тестирование правильности,* проверяющее корректность этапа *анализа требований* к ПС. На заключительном витке спирали проводится *системное тестирование,* выявляющее дефекты этапа *системного анализа* ПС.

Охарактеризуем каждый шаг процесса тестирования.

1. *Тестирование элементов.* Цель – индивидуальная проверка каждого модуля. Используются способы тестирования «белого ящика».

2. *Тестирование интеграции.* Цель – тестирование сборки модулей в программную систему. В основном применяют способы тестирования «черного ящика».

3. *Тестирование правильности.* Цель – проверить реализацию в программной системе всех функциональных и поведенческих требований, а также требования эффективности. Используются способы тестирования «черного ящика».

4. *Системное тестирование.* Цель – проверка правильности объединения и взаимодействия всех элементов компьютерной системы, реализации всех системных функций.

Организация процесса тестирования в виде эволюционной разворачивающейся спирали обеспечивает максимальную эффективность поиска ошибок. Возникает вопрос – когда заканчивать тестирование?

Ответ практика обычно основан на статистическом критерии: «Можно с 95%-ной уверенностью сказать, что провели достаточное тестирование, если вероятность безотказной работы центрального процессора с программным изделием в течение 1000 часов составляет по меньшей мере 0,995».

Научный подход при ответе на этот вопрос состоит в применении математической модели отказов. Например, для логарифмической модели Пуассона формула расчета текущей интенсивности отказов имеет вид:

$$\lambda(t) = \frac{\lambda_0}{\lambda_0 \times p \times t + 1}, \qquad (1)$$

где $\lambda(t)$ – текущая интенсивность программных отказов (количество отказов в единицу времени); λ_0 – начальная интенсивность отказов (в начале тестирования); p – экспоненциальное уменьшение интенсивности отказов за счет обнаруживаемых и устраняемых ошибок; t –время тестирования.

С помощью уравнения (1) можно предсказать снижение ошибок в ходе тестирования, а также время, требующееся для достижения допустимо низкой интенсивности отказов.

Проверка программы в нормальных экстремальных и исключительных ситуациях

Случаи, когда программа должна работать со всеми возможными данными, чрезвычайно редки. Обычно имеют место конкретные ограничения на область изменения данных, в которой программа должна сохранять свою работоспособность. Поэтому проверка в нормальных условиях должна показать, что программа выдает правильные результаты для характерных совокупностей данных.

Проверка в экстремальных условиях включает граничные значения области изменения входных переменных, которые должны восприниматься программой как правильные данные. Для числовых данных это начальное и конечное значения допустимой области. Для нечисловых данных могут быть типичные символы, охватывающие всевозможные ситуации. Типовыми примерами являются очень «большие» числа, очень «малые» числа, отсутствие информации.

Проверка в исключительных ситуациях это проверка программы с данными, которые лежат за пределами допустимой области изменения. Наихудшая ситуация, когда программа воспринимает неверные данные как правильные и выдает неверный, но правдоподобный результат. Программа должна отвергать данные, которые она не в состоянии обработать правильно.

Стрессовое тестирование

Стрессовые тесты проектируются для навязывания программам ненормальных ситуаций. В сущности, проектировщик стрессового теста спрашивает, как сильно можно расшатать систему, прежде чем она откажет?

Стрессовое тестирование производится при ненормальных запросах на ресурсы системы (по количеству, частоте, размеру-объему).

Примеры:

- генерируется 10 прерываний в секунду (при средней частоте 1,2 прерывания в секунду);
- скорость ввода данных увеличивается прямо пропорционально их важности (чтобы определить реакцию входных функций);
- формируются варианты, требующие максимума памяти и других ресурсов;
- генерируются варианты, вызывающие переполнение виртуальной памяти;
- проектируются варианты, вызывающие чрезмерный поиск данных на диске.

По существу, испытатель пытается разрушить систему. Разновидность стрессового тестирования называется *тестированием чувствительности.* В некоторых ситуациях (обычно в математических алгоритмах) очень малый диапазон данных, содержащийся в границах правильных данных системы, может вызвать ошибочную обработку или резкое понижение производительности. Тестирование чувствительности обнаруживает комбинации данных, которые могут вызвать нестабильность или неправильность обработки.

1.7 Искусство отладки

Отладка – это локализация и устранение ошибок. Отладка является следствием успешного тестирования. Это значит, что если тестовый вариант обнаруживает ошибку, то процесс отладки уничтожает ее.

Процессу отладки предшествует выполнение тестового варианта, его результаты оцениваются, регистрируется несоответствие между ожидаемыми и реальными результатами. Несоответствие является скрытой причиной, процесс отладки приводит к исправлению ошибки. Возможны два исхода процесса отладки:

1) причина найдена, исправлена, уничтожена;

2) причина не найдена.

Во втором случае отладчик может предполагать причину. Для проверки этой причины он просит разработать дополнительный тестовый вариант, который поможет проверить предположение. Таким образом, запускается итерационный процесс коррекции ошибки.

Возможные разные способы проявления ошибок:

- программа завершается нормально, но выдает неверные результаты;
- программа зависает;
- программа завершается по прерыванию;
- программа завершается, выдает ожидаемые результаты, но хранимые данные испорчены (это самый неприятный вариант).

Характер проявления ошибок также может меняться. Симптом ошибки может быть:

- постоянным;
- мерцающим;
- пороговым (проявляется при превышении некоторого порога в обработке – 200 самолетов на экране отслеживаются, а 201-й – нет);
- отложенным (проявляется только после исправления маскирующих ошибок).

В ходе отладки встречаются ошибки в широком диапазоне: от мелких неприятностей до катастроф.

Различают две группы методов отладки:

- аналитические;
- экспериментальные.

Аналитические методы базируются на анализе выходных данных для тестовых прогонов. Экспериментальные методы базируются на использовании вспомогательных средств отладки (отладочные печати, трассировки), позволяющих уточнить характер поведения программы при тех или иных исходных данных.

Общая стратегия отладки – обратное прохождение от замеченного симптома ошибки к исходной аномалии (месту в программе, где ошибка совершена).

Цель отладки – найти оператор программы, при исполнении которого правильные аргументы приводят к неправильным результатам. Если место проявления симптома ошибки не является искомой аномалией, то один из аргументов оператора должен быть неверным. Поэтому надо перейти к исследованию предыдущего оператора, выработавшего этот неверный аргумент. В итоге пошаговое обратное прослеживание приводит к искомому ошибочному месту.

В разных методах прослеживание организуется по-разному. В аналитических методах – на основе логических заключений о поведении программы. Цель – шаг за шагом уменьшать область программы, подозреваемую в наличии ошибки. Здесь определяется корреляция между значениями выходных данных и особенностями поведения.

Основное *преимущество аналитических методов отладки* состоит в том, что исходная программа остается без изменений.

В экспериментальных методах для прослеживания выполняется:

- выдача значений переменных в указанных точках;
- трассировка переменных (выдача их значений при каждом изменении);
- трассировка потоков управления (имен вызываемых процедур, меток, на которые передается управление, номеров операторов перехода).

Преимущество экспериментальных методов отладки состоит в том, что основная рутинная работа по анализу процесса вычислений перекладывается на компьютер. Многие трансляторы имеют встроенные средства отладки для получения информации о ходе выполнения программы.

Недостаток экспериментальных методов отладки – в программу вносятся изменения, при исключении которых могут появиться ошибки. Некоторые системы программирования создают специальный отладочный экземпляр программы, а в основной экземпляр не вмешиваются.

2 Лабораторные работы

Цель лабораторных работ – закрепление теоретических знаний и приобретение практических навыков в проведении отладки и тестирования разработанного программного обеспечения.

Лабораторная работа №1

Согласно варианту протестируйте программу вычислительного типа в нормальных, экстремальных и исключительных ситуация.

Сделайте выводы о проделанной работе.

Лабораторная работа №2

Согласно варианту протестируйте программу по обработке информации в нормальных, экстремальных и исключительных ситуация.

Сделайте выводы о проделанной работе.

Лабораторная работа №3

Согласно варианту протестируйте программу по методу «белого ящика».

Сделайте выводы о проделанной работе.

Лабораторная работа №4

Согласно варианту протестируйте программу по методу «черного ящика».

Сделайте выводы о проделанной работе.

Заключение

Создание программной системы – весьма трудоемкая задача, особенно когда объем программного обеспечения превышает сотни тысяч операторов. Будущий специалист в области разработки программного обеспечения должен иметь представление о методах анализа, проектирования, реализации и тестирования программных систем, а также ориентироваться в существующих подходах и технологиях.

Тестирование и отладка – очень важные, сложные и трудоемкие этапы процесса разработки программного обеспечения, так как правильное тестирование позволяет выявить подавляющее большинство ошибок, допущенных при составлении программ.

Целью всех проверок программного обеспечения является поиск несоответствий техническому заданию. Считают, что только после выполнения всех видов тестирования программный продукт может быть представлен пользователю или к реализации. Однако на практике выполняют не все виды тестирования, так как это очень дорого и трудоемко. Для каждого типа программного обеспечения выполняют те виды тестирования, которые являются для него наиболее важными, так например базы данных тестируют на предельных объемах информации, а системы реального времени – на предельных нагрузках.

В методических указаниях приведен теоретический материал по курсу «Тестирование ПО». Рассмотрены основные причины появления ошибок в ПС, даны определения основным понятиям курса. Приведена классификация ошибок, рассмотрено тестирование интеграции. Приведена методика тестирования программного обеспечения, а также тестирование по методам «белого ящика» и «черного ящика». Рассмотрены вопросы, связанные с искусством отладки ПО.

В практической части методических указаний приведены задания для выполнения лабораторных работ.

Список использованных источников

1 Кулямин, В. В. Технологии программирования. Компонентный подход: / В. В. Кулямин . - М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2011. - 463 с. : ил. ISBN 978-5-94774-544-3 (БИНОМ.ЛЗ), ISBN 5-9556-0067-1 (ИНТРУИТ.РУ)

2 Орлов, С. А. Технологии разработки программного обеспечения: разработка сложных программных систем: учеб. для вузов / С. А. Орлов .- 3-е изд. - СПб. [и др.] : Питер, 2004. - 527 с.: ил. ISBN 5-94723-145-X

3 Пилон, Д. Управление разработкой ПО/ Д. Пилон, Р. Майлз. – СПб.: Питер, 2011. – 464 с.: ил. ISBN 978-5-459-00522-6

4 Брауде, Э. Технология разработки программного обеспечения/ Э. Брауде. – СПб.: Питер, 2004. – 655 с.: ил. ISBN 5-94723-663-X

5 Гагарина, Л. Г. Технология разработки программного обеспечения: учеб. пособие для вузов по направлению "Информатика и вычислительная техника" / Л. Г. Гагарина, Е. В. Кокорева, Б. Д. Виснадул; под ред. Л. Г. Гагариной. - М. : ИД «ФОРУМ»: ИНФРА-М, 2008. – 400 с: ил. ISBN 978-5-8199-0342-1 (ИД «ФОРУМ») ISBN 978-5-16-003193-4 (ИНФРА-М)

6 Иванова, Г. С. Технология программирования: учебник для вузов / Г. С. Иванова .- 3-е изд., перераб. и доп. - М. : МГТУ им. Н.Э. Баумана, 2006. - 336 с.: ил. ISBN ISBN 5-7038-2077-4