Министерство образования Республики Беларусь
Учреждение образования «Полоцкий государственный университет»

Факультет информационных технологий
Кафедра технологий программирования

Методические указания
к выполнению лабораторной работы №12

по дисциплине «**Надёжность программного обеспечения**»
для специальности 1-40 01 01 Программное обеспечение информационных технологий

на тему «**Основы модульного тестирования**»

Новополоцк, 2017 г.

Название: «Основы модульного тестирования».

Цель работы: ознакомиться с основами модульного, объектно-ориентированного и пошагового тестирования. Научиться принципам тестирования структуры программных модулей.

Теоретическая часть

Модульное тестирование и его задачи

Модульное тестирование – вид тестовой деятельности, при которой проверке подвергаются внутренние рабочие части программы, элементы или модули независимо от способа их вызова. Под модулем принято понимать программу или ограниченную часть кода с одной точкой входа и одной точкой выхода, которая выполняет одну и только одну первичную функцию.

Этот тип тестирования, как правило, осуществляется программистом, а не тестировщиком, поскольку для его проведения необходимы доскональные знания внутренней структуры и кода программы. Такое тестирование не всегда выполняется просто, например, в случае, если прикладная программа не имеет четко определенной архитектуры с компактным кодом, и для его проведения может понадобиться разработка модулей тестовых драйверов или средств тестирования.

Тестировать свои собственные продукты – это весьма трудное занятие для разработчиков, поскольку они вынуждены изменить свою точку зрения или отношение, перейдя от роли создателя в роль критика. Многие разработчики на самом деле не любят тщательно тестировать свои программные продукты, так как считают скучным и даже угнетающим наблюдать за тем, как прикладная программа справляется с возложенными на нее нагрузками. Другие не имеют ничего против подобного подхода и прекрасно выполняют такого рода работу. Такой подход, несомненно, приводит к обнаружению ошибок на более ранних этапах разработки программного продукта, а это в свою очередь способствует снижению стоимости ошибки. Даже самый опытный программист допускает ошибки. Некоторые исследования показывают, что плотность распределения дефектов находится в диапазоне от 49,5 до 94,6 на тысячу строк кода. Поэтому каждый разработчик должен самостоятельно тестировать свои «произведения» с максимальной тщательностью, прежде чем передать рабочий продукт независимому испытателю, т.е. тестировщику.

Методы, используемые при модульном тестировании, различаются по следующим признакам:

– по степени автоматизации – ручные и автоматизированные методы;

– по форме представления модуля – символьное представление (на языке программирования) или в машинном коде;

– по компонентам программы, на которые направлено тестирование – структура программы или преобразование переменных;

– статические или динамические методы.

В книге В.В. Липаева «Тестирование программ» отмечается, что в первую очередь следует тестировать структуру программы, так как операторы анализа условий составляют в среднем 10-15% от общего числа операторов программы. Искажение логики работы программы приводит к серьезным ошибкам. К тому же данный вид тестирования имеет наилучшие показатели «эффективность/стоимость». Там же предлагается следующая методика тестирования модулей: последовательно проводить различные виды тестирования, начиная с самых простых:

– ручное тестирование (работа за столом);

– символическое тестирование (инспекции, сквозные просмотры);

– тестирование структуры;

– тестирование обработки данных;

– функциональное тестирование (сравнение со спецификацией, взаимодействие с другими модулями).

Из приведенных выше видов тестирования, ручное и символическое относятся к статическому тестированию, которое проводится без запуска программы. Эти два вида, как правило, основаны на обзоре программного кода, только первый проводится автором-разработчиком, а второй – сторонними специалистами (другими разработчиками, инженерами по качеству или приглашенными инспекторами).

Последние три из перечисленных видов относятся к динамическому тестированию, и проведение каждого из них состоит из следующих этапов:

– планирование тестирования (разработка тестов, формирование контрольных примеров);

– собственно тестирование;

– обработка результатов тестирования.

Обзоры

Обзоры являются основой статического тестирования и способны нейтрализовать действие человеческого фактора при выполнении поставленных задач. При модульном тестировании можно выделить следующие положительные моменты обзоров (учитывая, что оно может проводиться как самим разработчиком, так и сторонними специалистами):

– обзоры позволяют обнаружить посторонние элементы, что нельзя сделать в ходе традиционного тестирования. Обзоры могут следовать по всем выполняемым ветвям разрабатываемого ПО, а с помощью тестирования невозможно проверить все ветви, в результате чего редко проходимые ветви часто приводят к появлению ошибок;

– разные точки зрения, личностные качества и жизненный опыт помогают при обнаружении всех видов проблем, которые незаметны при поверхностном взгляде;

– разработчики могут больше внимания уделять творческому аспекту процесса разработки, зная, что их эксперты участвуют в проекте и действуют в качестве «подстраховки», и всегда готовы увидеть реальную картину;

– происходит разделение труда, благодаря чему рецензенты могут сконцентрироваться на этапе обнаружения проблем;

– распространение информации и обучение происходят тогда, когда разработчики встречаются при проведении обзора. Люди быстро воспринимают и распространяют хорошие идеи, которые они замечают в работе других пользователей. По словам некоторых экспертов – это самый важный результат выполнения обзора;

– возрастает степень согласованности работы между членами команды. Происходит установление фактических групповых норм, что облегчает понимание сути программных продуктов и их сопровождение;

– менеджеры проекта могут получить представление о действительном состоянии разрабатываемых продуктов.

В конечном счете, в результате обзора программные продукты улучшаются в силу следующих причин:

– проблемы обнаруживаются тогда, когда их можно относительно легко исправить без значительных понесенных затрат;

– локализация проблем происходит практически в месте их возникновения;

– инспектируются исходные данные какой-либо фазы с целью проверки их соответствия исходным требованиям или критериям выхода;

– предотвращается возникновение дальнейших проблем с помощью оглашения решений часто происходящих затруднений;

– распространяются сведения о проекте, что способствует повышению его управляемости;

– разработчики обучаются тому, каким образом избежать возникновения дефектов при дальнейшей работе;

– предотвращается возникновение дефектов в текущем продукте, поскольку процесс подготовки материалов для инспекционной проверки способствует уточнению требований и проекта;

– обучаются новые участники проекта;

– менеджерам проекта предоставляются надежные опорные точки и предварительные оценки;

– облегчается поддержка установленного порядка выполнения проекта и обеспечивается объективная, измеримая обратная связь.

Принципы тестирования структуры программных модулей

Как правило, тестирование структуры программного модуля происходит с помощью методов графического отображения модуля, например, диаграммы Чейпина или Несси Шнайдермана, ориентированные графы Мак-Кейба. Подобно Чейпину, Мак-Кейб применяет следующие конструкции: if-then-else, if then, do-while, case и repeat-until. Эти конструкции изображены на рисунке 1.

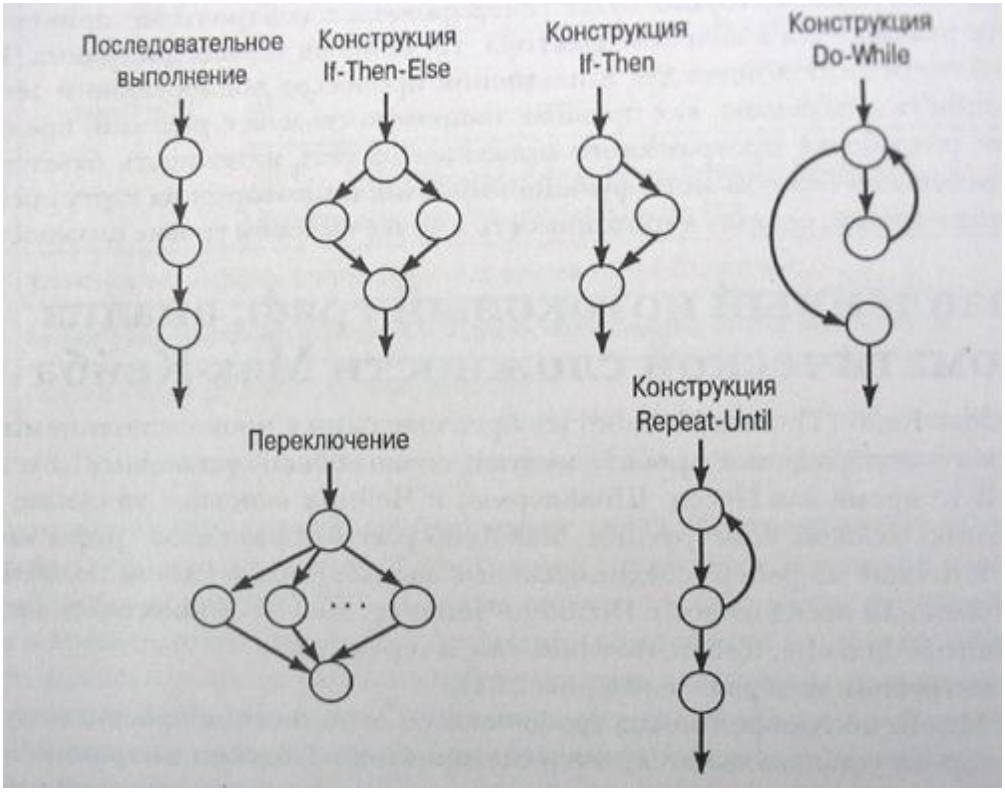

Рисунок 1 – Конструкции из теории графов

Метрический показатель сложности или цикломатическое число ***G*** потокового графа определяется по формуле:

G=R-V+2,

где ***R*** – количество ребер графа,

V – количество вершин графа.

Чтобы вычислить этот коэффициент вручную для большой программы, потребуется немало усилий. К счастью, существуют автоматические средства, которые вычисляют метрический показатель сложности Мак-Кейба путем анализа существующего исходного кода.

Метрический показатель сложности не только может пригодиться при установлении проблемных областей, но также может использоваться при создании контрольных примеров. Как только потоковый граф создан, очевидными становятся пути, ведущие через программу, которые предоставляют информацию, необходимую для их выполнения в тесте.

При планировании тестирования структуры программных модулей решаются 2 задачи:

– формирование критериев выделения маршрутов в программе;

– выбор стратегий упорядочения выделенных маршрутов.

Критерии выделения маршрутов в программе могут быть следующими:

– минимальное покрытие графа программы;

– маршруты, образующиеся при всех возможных комбинациях входящих дуг.

Стратегия упорядочения маршрутов выбирается по:

– длительности исполнения и числу команд в маршрутах. Такая стратегия выбирается при тестировании программ вычислительного характера;

– количеству условных переходов, определяющих формирование данного маршрута. Данная стратегия выбирается при тестировании логических программ с небольшим объемом вычислений по вероятности исполнения маршрутов. Применяется в тех случаях, когда сложно оценить вероятности ветвления в условных переходах, а также число исполнений циклов.

Способы тестирования взаимодействия модулей

Майерс Г. в книге «Искусство тестирования программ» выделяет два способа проверки взаимодействия модулей:

– монолитное тестирование;

– пошаговое тестирование.

Пусть имеется программа, состоящая из нескольких модулей, представленная на рисунке 2.

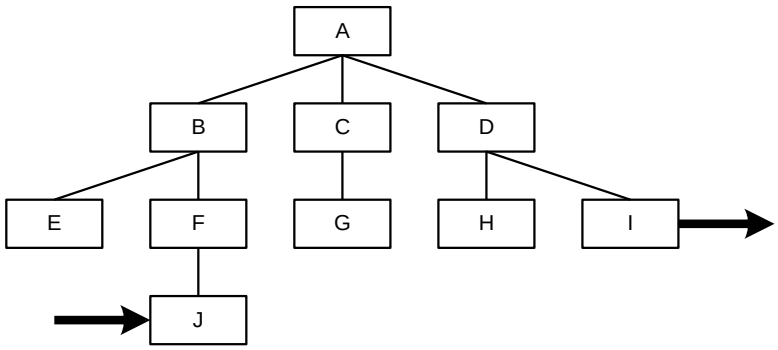

Рисунок 2 – Схема многомодульной программы

Обозначения:

Прямоугольники – модули, тонкие линии представляют иерархию управления (связи по управлению между модулями), жирные стрелки – ввод и вывод данных в программу.

Монолитное тестирование заключается в том, что каждый модуль тестируется отдельно. Для каждого модуля пишется один модуль драйвер, который передает тестируемому модулю управление, и один или несколько модулей-заглушек. Например, для модуля B нужны 2 заглушки, имитирующие работу модулей E и F. Когда все модули протестированы, они собираются вместе, и тестируется вся программа в целом.

Пошаговое тестирование предполагает, что модули тестируются не изолированно, а подключаются поочередно к набору уже оттестированных модулей.

Можно выделить следующие *недостатки* монолитного тестирования (перед пошаговым):

1 Требуется много дополнительных действий (написание драйверов и заглушек).

2 Поздно обнаруживаются ошибки в межмодульных взаимодействиях.

3 Следствие из 2 – труднее отлаживать программу.

К *преимуществам* монолитного тестирования можно отнести:

1 Экономию машинного времени (в настоящее время существенной экономии не наблюдается).

2 Возможность параллельной организации работ на начальной фазе тестирования.

Стратегии выполнения пошагового тестирования

Существуют две принципиально различных стратегии выполнения пошагового тестирования:

– нисходящее тестирование;

– восходящее тестирование.

Нисходящее тестирование начинается с головного модуля, в нашем случае с модуля A. Возникают проблемы: как передать тестовые данные в A, ведь ввод и вывод осуществляется в других модулях? Как передать в A несколько тестов?

Решение можно представить в следующем виде:

а) написать несколько вариантов заглушек B (для каждого теста);

б) написать заглушку B так, чтобы она читала тестовые данные из внешнего файла.

В качестве стратегии подключения модулей можно использовать одну из следующих:

а) подключаются наиболее важные с точки зрения тестирования модули;

б) подключаются модули, осуществляющие операции ввода/вывода (для того, чтобы обеспечивать тестовыми данными «внутренние» модули.

Нисходящее тестирование имеет ряд недостатков: предположим, что модули I и J уже подключены, и на следующем шаге тестирования заглушка H меняется на реальный модуль H. Как передать тестовые данные в H? Это нетривиальная задача, потому что между J и H имеются промежуточные модули и может оказаться невозможным передать тестовые данные, соответствующие разработанным тестам. К тому же достаточно сложно интерпретировать результаты тестирования H, так как между H и I также существуют промежуточные модули.

Восходящее тестирование практически полностью противоположно нисходящему тестированию. Начинается с терминальных (не вызывающих другие модули) модулей.

Стратегия подключения новых модулей также должна основываться на степени критичности данного модуля в программе. *Восходящее тестирование* лишено недостатков нисходящего тестирования, однако имеет

свой, главный недостаток: рабочая программа не существует до тех пор, пока не добавлен последний (в нашем случае А) модуль.

Выбор одной из двух представленных стратегий определяется тем, на какие модули (верхнеуровневые или модули нижнего уровня) следует обратить внимание при тестировании в первую очередь.

Объектно-ориентированное тестирование

С традиционной точки зрения наименьший контролепригодный элемент – это, элемент или модуль, который можно протестировать методом «белого ящика». При объектно-ориентированном тестировании этот функциональный элемент не отделяется от своего класса, в котором он инкапсулирован со своими методами. Это означает, что поэлементное тестирование по существу заменяется классовым тестированием. Однако мы не отклоняемся слишком далеко от наших корневых принципов относительно того, что каждый метод класса объектов можно рассматривать как «небольшой элемент», тестирование которого можно выполнять обособленно, перед тестовыми последовательностями, применяя для этого методы «белого и черного ящика». Классы объектов необходимо протестировать во всех возможных состояниях. Это означает, что необходимо сымитировать все события, вызывающие изменения состояния объекта.

В традиционном отношении элементы компилируются в подсистемы и подвергаются интеграционным тестам. При объектно-ориентированном подходе не применяется тестирование структурной схемы сверху вниз, поскольку точно не известно, какой класс будет вызван пользователем за предыдущим. Интеграционные тесты заменяются тестированием функциональных возможностей, при котором тестируется набор классов, которые должны реагировать на один входной сигнал или системное событие, или же эксплуатационным тестированием, при котором описывается один способ применения системы, основанный на сценариях случаев эксплуатации.

Тестирование взаимодействия объектов следует по путям сообщения с целью отследить последовательность взаимодействий объектов, которая заканчивается только тогда, когда был вызван последний объект. При этом не посылается сообщение и не вызываются службы любого другого объекта.

Системное, альфа-, бета-тестирование и приемочное испытание пользователем изменяются незначительно, поскольку объектно-ориентированная система проходит эксплуатационные тесты точно также, как и разработанные традиционным способом системы. Это означает, что процесс V&V по существу не изменяется.

Тестирование классов охватывает виды деятельности, направленные на проверку соответствия реализации этого класса спецификации. Если реализация корректна, то каждый экземпляр этого класса ведет себя подобающим образом.

Тестирование классов в первом приближении аналогично тестированию модулей и может проводиться с помощью обзоров (ревизий) и выполнения тестовых случаев.

К недостаткам обзоров (ревизий) относятся:

– возможные ошибки, обусловленные человеческим фактором;

– значительно большие затраты и усилия, нежели регрессионное тестирование.

Тестирование в режиме прогона тестовых случаев лишено указанных выше недостатков, однако необходимо приложить значительные усилия для отбора подходящих тестовых случаев и для разработки тестовых драйверов.

Прежде чем приступать к тестированию класса, необходимо определить, тестировать его в автономном режиме как модуль или каким-то другим способом, как более крупный компонент системы. Для этого необходимо выяснить:

– роль класса в системе, в частности, степень связанного с ним риска;

– сложность класса, измеряемая количеством состояний, операций и связей с другими классами;

– объем трудозатрат, связанных с разработкой тестового драйвера для тестирования этого класса.

Если какой-либо класс должен стать частью некоторой библиотеки классов, целесообразно выполнять всестороннее тестирование классов, даже если затраты на разработку тестового драйвера окажутся высокими.

При тестировании классов можно выделить 5 оцениваемых факторов:

1 Кто выполняет тестирование. Как и при модульном тестировании, тестирование классов выполняет разработчик, поскольку он знаком со всеми подробностями программного кода. Основной недостаток того, что тестовые драйверы и программные коды разрабатываются одним и тем же персоналом, заключается в том, что неправильное понимание спецификации разработчиками распространяется на тестовые наборы и тестовые драйверы. Проблем такого рода можно избежать путем привлечения независимых тестировщиков или других разработчиков для ревизий программных кодов.

2 Что тестировать. Тестировать нужно программный код на точное соответствие его требованиям, т.е. в классе должно быть реализовано все запланированное и ничего лишнего. Если для конкретного класса характерны статические элементы, то их также необходимо тестировать. Такие элементы принадлежат самому классу, но не каждому экземпляру.

3 Когда тестировать. Тестирование класса должно проводиться до того, как возникнет необходимость его использования. Необходимо также проводить регрессионное тестирование класса каждый раз, когда меняется его реализация. Однако до начала тестирования класса необходимо закодировать класс и разработать тестовые случаи использования класса. Ранняя разработка тестовых случаев позволяет программисту лучше понять спецификацию класса и построить более успешную реализацию класса, которая пройдет все тестовые случаи. Существует даже практика первоначальной разработки тестовых случаев, а затем программного кода.

Такой подход направлен на изначально все предусматривающий программный код. Главное, чтобы этот подход не привел к проблемам во время интеграции этого класса в более крупную часть системы.

4 Каким образом тестировать. Тестирование классов обычно выполняется путем разработки тестового драйвера. Тестовый драйвер создает экземпляры классов и окружает их соответствующей средой, чтобы стал возможен прогон соответствующего тестового случая. Драйвер посылает одно или большее количество сообщений экземпляру класса (в соответствии с тестовым случаем), затем сверяет результат его работы с ожидаемым и составляет протокол о прохождении или не прохождении теста. В обязанности тестового драйвера обычно входит и удаление созданного им экземпляра.

5 В каких объемах тестировать. Адекватность может быть измерена полнотой охвата тестами спецификации и реализации класса. Т.е необходимо тестировать операции и переходы состояний в различных сочетаниях. Поскольку объекты находятся в одном из возможных состояний, эти состояния определяют значимость операций. Поэтому требуется определить, целесообразно ли проводить исчерпывающее тестирование. Если нет, то рекомендуется выполнить наиболее важные тестовые случаи, а менее важные выполнять выборочно.

Практическая часть

Содержание задания

Провести модульное тестирование для программного продукта из предыдущей лабораторной работы. Составить отчет о проделанной работе.

Порядок выполнения работы

1 Ознакомиться с теоретической частью.

2 Провести обзор разработанного программного кода. Составить отчет обнаружения ошибок (**не менее трех ошибок**). Пример отчета приведен в таблице 1.

Таблица 1 – Ошибки обзора

№ Ошибки	**Название модуля/функции**	**Описание ошибки**	**Важность ошибки (высокая, средняя, низкая)**	**Ошибка исправлена Да/Нет**
1	Func1()	В первом цикле for не верно указано значение окончания цикла	Средняя	Да

3 Для **двух** модулей/функций построить графы и вычислить цикломатические числа.

4 Разработать тестовые сценарии для модульного тестирования. Тестовые сценарии для каждого графа представить в виде таблицы. Пример сценария приведен в таблице 2.

Таблица 2 – Модуль Func2()

G	**№ сценария**	**Описание прохода**	**Контрольные примеры, позволяющие реализовать описанную ситуацию**	**Тест пройден Да/Нет**
G=2	1	a-b-d-f	Field=4, x=-5	Нет
	2	a-b-e-f	Field=-3, x=-5	Да

5 Построить схему взаимодействия модулей/функций (**всех модулей/функций программы**).

6 Разработать стратегию тестирования взаимодействия модулей/функций. Пример взаимодействия модулей/функций приведен в таблице 3.

Таблица 3 – Взаимодействие модулей/функций

№ в последовательности	**Описание последовательности**	**Контрольные примеры, позволяющие реализовать описанную ситуацию**	**Тест пройден Да/Нет**
1	Func1()->Func2()	Argument=0	

7 Составить отчет о проделанной работе.

8 Показать отчёт преподавателю.

9 После защиты, выполненного задания, скинуть отчёт преподавателю (в **названии** указать **фамилию и номер лабораторной**, например, Ivanov12).

Содержание отчета

1 Титульный лист (**на титульном листе вместо варианта указать название тестируемого программного продукта**).

2 Цель работы.

3 Краткое описание программы из варианта задания.

4 Результаты выполнения задания с пояснениями.

5 Вывод о проделанной работе.

Защита работ проводится индивидуально.

Контрольные вопросы

1 Что такое модульное тестирование?

2 Какие стратегии выполнения пошагового тестирования существуют?

3 Какие факторы можно выделить при тестировании классов?

4 В чем различие тестирования методами «белого» и «черного» ящика?

5 Что такое монолитное тестирование?

6 Дайте определение пошагового тестирования?