Министерство образования Республики Беларусь
Учреждение образования «Витебский государственный
университет имени П.М. Машерова»
Кафедра информатики и информационных технологий

В.В. Шедько

НАДЕЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Методические рекомендации к выполнению лабораторных работ

Витебск
ВГУ имени П.М. Машерова
2017

УДК 004.056.57(076.5)
ББК 32.972.53я73
Ш38

Печатается по решению научно-методического совета учреждения образования «Витебский государственный университет имени П.М. Машерова». Протокол № 1 от 19.10.2017 г.

Автор: старший преподаватель кафедры информатики и информационных технологий ВГУ имени П.М. Машерова **В.В. Шедько**

Р е ц е н з е н т :
доцент кафедры математики и информационных технологий УО «ВГТУ», кандидат технических наук *А.С. Дягилев*

Шедько, В.В.

Ш38 Надежность программного обеспечения : методические рекомендации к выполнению лабораторных работ / В.В. Шедько. – Витебск : ВГУ имени П.М. Машерова, 2017. – 26 с.

Учебное издание содержит материал по предмету, вопросы и индивидуальные задания для лабораторных занятий и самостоятельной работы и краткие теоретические сведения.

Методические рекомендации предназначены для студентов дневной и заочной форм обучения и могут использоваться при изучении дисциплин «Надежность программного обеспечения» **(**специальность «Программное обеспечение информационных технологий»), «Тестирование и оценка качества программного обеспечения» (специальность «Прикладная информатика (по направлениям)»).

УДК 004.056.57(076.5)
ББК 32.972.53я73

СОДЕРЖАНИЕ

ЛАБОРАТОРНАЯ РАБОТА № 1
Модели и способы повышения надежности ПО

Цель и задачи работы: Получить навыки практического использования численной оценки надежности ПО и сложных программных комплексов.

Теоретический материал

Характеристики надежности

При анализе надежности выполнения ЭВМ заданных функций компьютер следует рассматривать как единый комплекс программных и аппаратных средств и учитывать, что его надежность зависит также от надежности программ.

Надежность программного обеспечения – это свойство сохранять заданные характеристики при определенных условиях эксплуатации. Надежность ПО определяется его безотказностью и восстанавливаемостью. *Безотказность* ПО – его свойство сохранять работоспособность в процессе обработки информации, которую можно определить вероятностью работы без отказов при определенных условиях внешней среды в течение заданного периода наблюдения.

Отказ программы – это недопустимое отклонение характеристик процесса функционирования программы от требуемых. Определенные условия внешней среды – это совокупность входных данных и состояния вычислительной системы.Заданный период наблюдения обычно соответствует необходимому числу прогонов программы для решения задачи.

Безотказность ПО можно охарактеризовать также *средним временем между двумя отказами* в процессе выполнения программы *T* (при условии, что сбой аппаратных средств отсутствует). С точки зрения надежности принципиальное отличие программных средств от аппаратных состоит в том, что программы не изнашиваются и не подвержены физическому старению в процессе работы.

Поэтому характеристики надежности ПО зависят от тщательности разработки и отладки, а также от условий хранения носителей программ. Безотказность ПО определяется его корректностью, а значит целиком зависит от наличия в нем ошибок, внесенных на этапе создания и хранения, в то время как безотказность аппаратных средств зависит в основном от случайных отказов, связанных с физическими изменениями параметров элементов. Вид обрабатываемых данных не влияет на аппаратуру, но может привести к отказам ПО.

Интенсивность отказов ПО с течением времени уменьшается, так как в процессе эксплуатации обнаруживаются и устраняются его ошибки.

Восстанавливаемость программы может быть оценена сравнительной продолжительностью устранения ошибки в программе и восстановления ее работоспособности. Восстановление после отказа может заключаться в корректировке текста программы, исправлений данных, внесения измене-

ний в организацию вычислительного процесса. Восстанавливаемость зависит от сложности структуры комплекса программ, от алгоритмического языка, от качества документации и т.д. Можно также говорить об *устойчивости* ПО, понимая под этим способность ограничивать последствия собственных ошибок и противостоять неблагоприятным условиям внешней среды. Устойчивость ПО может быть повышена с помощью разных форм структурной, информационной и временной избыточности, позволяющей иметь дублирующие модули программ, альтернативные пути для решения одной задачи, позволяющих осуществлять контроль за процессом исполнения программ (зацикливание, блокировка и т.д.).

Причины отказов ПО:

- Ошибки, скрытые в самой программе.
- Искажение входной информации, подлежащей обработке.
- Неверные действия пользователя (могут быть связаны с некорректной документацией).
- Неисправность аппаратуры, на которой реализуется вычислительный процесс.

Последствия и признаки появления ошибок в программе.

В зависимости от степени серьезности последствий ошибок в программе, отклонения выполнения программой заданных функций можно разделить следующим образом: полное прекращение выполнения функций на длительное или неопределенное время или кратковременное прекращение хода вычислительного процесса.

Симптомы проявления ошибки в программе:

- преждевременное окончание программы;
- увеличение времени выполнения программы (зацикливание);
- потери или искажение накопленных данных;
- нарушение порядка вызова отдельных программ.

Для устранения ошибок программы необходимо предусмотреть специальные средства диагностики типа кодов завершения, вводить в ПО контрольные точки, обеспечить возможность рестарта с контрольных точек.

Способы повышения надежности ПО. Эргономичность программы. Роль структурного программирования в повышении эргономичности.

Эргономичность — в изначальном смысле это эффективность инструмента производства или системы в эргономике (Эргономика — научная дисциплина, комплексно изучающая производственную деятельность человека и ставящая целью её оптимизацию).

Эргономичность как характеристика программного продукта обозначает степень, с которой программа позволяет минимизировать усилия пользователя по подготовке исходных данных, обработке данных и оценке полученных результатов. Чем меньше движений совершает пользователь мышью, чем меньше информации вводит он с клавиатуры и чем быстрее он находит требуемую информацию – тем выше степень эргономичности.

Стержнем структурного программирования является создание максимально ясных, легко понимаемых программ - необходимого условия надежности и правильности программных продуктов.

Важнейшими концепциями структурного программирования, направленными на получение качественных программ, являются:

1. Упорядочение и ограничение управляющих и информационных структур таким образом, чтобы структура программы и данных отражала структуру решаемой задачи. Другими словами, должен выполняться следующий принцип (Э.Дейкстра): соответствие текстуальной упорядоченности программы порядку вычислений.

2. Разработка структуры программы путем систематизированного пошагового уточнения (метода "сверху-вниз"), ограничивающего сложность разработки на каждом уровне иерархии до приемлемой.

3. Использование системы обозначений, которая облегчает разработку и преобразование управляющих структур в конечный программный продукт.

Большая часть существующих языков программирования при своем создании не ориентировалась на поддержку структурного программирования, хотя многие из них имеют соответствующие управляющие структуры и при введении ограничений на использование ряда средств прекрасно согласуется со структурным подходом.

Аналитические модели надежности программ

Аналитические модели дают возможность исследовать закономерности появления ошибок ПО, а также прогнозировать надежность эксплуатации ПО. Модели строятся в предположении, что появление ошибок является случайным событием и имеет вероятностный характер.

Функцию надежности P(t) можно определить как вероятность того, что ошибка появится в программе не ранее, чем через время t. Обратная величина Q(t) - вероятность того, что ошибка произойдет за время t.

Из этих характеристик можно вывести величину интенсивности отказов L(t), которая будет определяться плотностью вероятности возникновения отказа:

$$L(t)= -(dP(t) / dt) / P(t)$$

Для ПО характерно ступенчатое изменение L(t). Поэтому наиболее простая модель надежности ПО – это дискретная модель:

$$L(t)= K(M-i(t)) =Li$$

M- постоянная, характеризующая начальное число ошибок;

i(t) – число отказов, устраняемое в момент времени t;

K- эмпирический коэффициент, зависящий от характеристик системы.

Эти параметры можно найти на основании последовательности наблюдений интервалов между обнаружением ошибок по методу максимального правдоподобия.

Если i(t) постоянна - получаем простую линейную модель.

Рассматриваемая модель является грубой. На практике условия, в которых она работает, не соблюдаются. Например, некоторые ошибки являются неустранимыми.

Модель надежности программ с дискретным увеличением времени наработки на отказ строится на гипотезе, что устранение ошибки приводит к увеличению времени наработки на отказ на некоторую случайную величину:

$$Ti = M[dT]\, m(m+1))/2$$

Ti- время наработки на отказ до возникновения m-го отказа;
M[dT] - математическое ожидание времени между двумя отказами:
$M[dT]=\sum ti / m;$.

Для того, чтобы посчитать, какое время тестирования необходимо для обеспечения некоторой наработки на отказ, нужно сначала определить математическое ожидание времени между двумя отказами по существующим данным. Далее надо последовательно высчитывать по формуле время наработки на отказ до тех пор, пока оно не достигнет нужного значения. Просчитанные данные лучше заносить в таблицу. После достижения нужного значения все времена нужно сложить, получив в результате искомое время тестирования.

Экспоненциальная модель надежности ПО основана на предположении об экспоненциальном характере изменения числа ошибок во времени. В этой модели прогнозируется надежность программы на основе данных, полученных во время тестирования. В модели вводится суммарное время функционирования , которое отсчитывается с момента начала тестирования до контрольного момента оценки надежности. Число ошибок m, выявленных за время tбудет определяться зависимостью (рис. 1): $m = M(1-\exp(-Kt))$, (1) где К – коэффициент пропорциональности; М – число ошибок, имевшееся перед фазой тестирования.

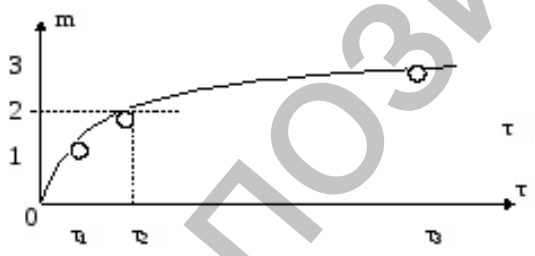

Рисунок 1. Экспоненциальная модель надежности ПО

Пусть среднее время наработки на отказ Tn =1/ L(t)
Если ввести величинуT^*n - начальное значение средней времени наработки на отказ, то: $T^*n= 1/ KM;$

$$Tn = T^*n\exp (t/ MT^*n) \qquad (.2)$$

Для того, чтобы определить, какое время тестирования необходимо для обеспечения указанного времени наработки, то необходимо рассчитать параметры закона распределения К и М. Зная эти параметры можно рассчитать прогнозируемое количество в любой момент времени. Следовательно, чтобы

определить, какое время тестирования необходимо для определения необходимого времени наработки, нужно итеративно высчитывать промежутки времени между ошибками по формуле (2) до тех пор, пока необходимое время наработки не будет достигнуто. Другим способом решения этой задачи является аппроксимация функции вида (1) по заданным точкам.

Для определения надежности больших программных комплексов используются *марковские модели*. В марковском процессе выбор следующего модуля зависит только от модуля, выполняемого в данный момент и не зависит от предыстории. Структуру управления программой по марковской модели можно представить в виде направленного графа (рис.2).

Каждое состояние программы может быть оценено вероятностью безотказной работы i-го модуля Ri. Вероятность перехода от i-го модуля к j-му показана величиной Pij.

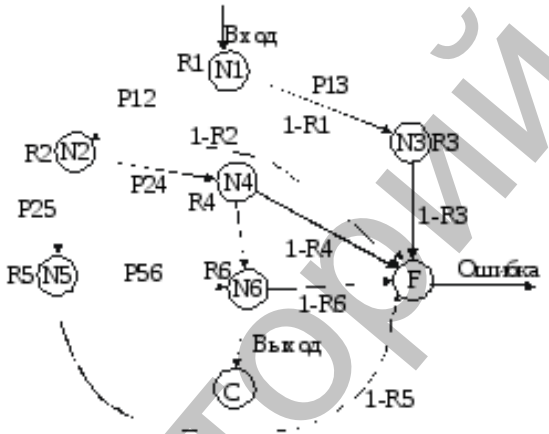

Рисунок 2. Граф надежности программного комплекса.

Вероятность отказа равна 1-P. Таким образом можно составить матрицу переходных вероятностей:

$$P = \begin{vmatrix} 0 & R_1P_{12} & R_1P_{13} & \dots & R_i & 0 & 1-R_1 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & R_iP_{ij} & \dots & \dots & 0 & 1-R_{n-1} \\ 0 & \dots & R_{n-1}P_{(n-1)j} & \dots & \dots & R_n & 1-R_n \\ 0 & \dots & \dots & \dots & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{vmatrix}$$

Для каждого целого k>0 $P^k(i, j)$ будет определять вероятность перехода из состояния i в состояние j за k шагов. Тогда матрица

$T= I+P+P^J+\dots\dots= \sum P^L$

Будет определять вероятность перехода и одного состояния в другое за произвольное число шагов.

Вычеркнув из матрицы P две последние строки и два последних столбца, соответствующие успеху C и отказу F, получаем матрицу Q.

$$S= I+Q+Q^J+\ldots\ldots= \sum Q^L.$$

Положив W = I – Q, имеем: $S = W^{-1} = (I – Q)^{-1}$, откуда надежность программного комплекса: $R = S (1,n) R_n$

Простая интуитивная модель. Использование этой модели предполагает проведение тестирования двумя группами программистов (или двумя программистами в зависимости от величины программы) независимо друг от друга, использующими независимые тестовые наборы. В процессе тестирования каждая из групп фиксирует все найденные ею ошибки. При оценке числа оставшихся в программе ошибок результаты тестирования обеих групп собираются и сравниваются.

Получается, что первая группа обнаружила N_1ошибок, вторая – N_2, а N_{12} – это ошибки, обнаруженные обеими группами.

Если обозначить через N неизвестное количество ошибок, присутствовавших в программе до начала тестирования, то можно эффективность тестирования каждой из групп определить как

$E1= N_1/N$; $E2= N_2 / N$; (*)

Предполагая, что возможность обнаружения всех ошибок одинакова для обеих групп, можно допустить, что если первая группа обнаружила определенное количество всех ошибок, она могла бы определить то же количество любого случайным образом выбранного подмножества. В частности, можно допустить: $E1= N_1/ N=N_{12}/ N_2$; (**)

Из формулы (*) $N_2 = E_2N$, подставив в (**), получим:

$$E1= N_{12} /E2\ N= N_1N_2/ N_{12}$$

ПРАКТИЧЕСКИЕ ЗАДАНИЯ.

Задание 1

1) Ознакомитесь с общими теоретическими положениями и ответьте на контрольные вопросы.

2) Подготовить краткую характеристику модели надежности, согласно варианту.

Вариант	Модель	Вариант	Модель
1.	Нельсона	7.	Джелинского – Моранды
2.	Коркорэна	8.	LaPadula
3.	Липова	9.	Шумана
4.	Миллса	10.	переходных вероятностей
5.	Муса	11.	Эмпирические
6.	Шика – Волвертона		

3) Составьте план вычисления надежности, используя модель из п.2.

Задание 2

1) Изучите теоретический материал о вычислении надежности ПО с помощью простой интуитивной модели.

2) Проведите необходимые организационные мероприятия и реализуйте вычисление надежности, используя простую интуитивную модель.

Задание 3

1) Определить, какое время тестирования требуется для достижения указанного времени наработки на отказ. В таблице указано время обнаружения первой и второй ошибки. Задачу решить путем составления программы численного решения уравнений (1) или (2).

Решение выполнять с экспоненциальной моделью и моделью с дискретным увеличением наработок на отказ.

Вариант	Время 1-го отказа, с	Время 2-го отказа, с	Время наработки, час
1	0,5	500	20
2	0,2	12	15
3	2	86	3
4	1	34	6
5	2	280	10

2) Определить надежность комплекса программ, который отображается графом, показанным на рис. 2.

Задание 4

1) Изучите теоретический материал о повышении надежности ПО. Определите факторы в простой интуитивной модели, с помощью которых можно повысить надежность ПО. Проанализируйте факторы не входящие в данную модель, с помощью которых можно повысить надежность ПО.

2) Составьте план для повышения надежности ПО и проанализируйте – как его реализация скажется на изменении надежности в рамках простой интуитивной модели.

Отчет должен содержать схему алгоритма и листинг разработанной программы экспоненциальной модели надежности, результаты работы программы, а также расчет надежности программного комплекса с помощью Марковской модели.

Контрольные вопросы

1. Перечислите основные характеристики надежности ПО.

2. Укажите основные классы скрытых ошибок программного обеспечения.

3. Существует ли модель, позволяющая точно определить количество обнаруженных ошибок ПО на определенном промежутке времени?

4. Укажите особенности использования модели надежности с дискретным увеличением времени наработки на отказ и экспоненциальной модели.

5. Какими способами можно повысить надежность разрабатываемого ПО?

6. Опишите способы оценки надежности больших программных комплексов.

7. Расскажите о современных технологиях повышения надежности программного обеспечения.

Задания для самостоятельной работы

1) Изучите теоретический материал о повышении надежности ПО.

2) Составьте краткий реферат (презентацию) о возможностях повышения надежности ПО и реализации данных возможностей в рамках модели своего варианта из задания 1 лабораторной работы 1.

3) Приведите пример организационного мероприятия по повышению надежности ПО и реализуйте вычисление изменения надежности в рамках модели вашего варианта, при успешной реализации мероприятия.

ЛАБОРАТОРНАЯ РАБОТА № 2
Основные понятия и принципы организации тестирования

Цель и задачи работы: Получить навыки организации практического тестирования ПО.

Теоретический материал

Цель и содержание отладки программы. Классификация ошибок. Уровни корректности программы в процессе отладки.

Цель и содержание отладки:

Отладка представляет собой процесс поиска, локализации и устранения ошибок в программе.

Наличие ошибки в программе проявляется разными путями:

- программа не завершается (зацикливается или ее обработка прерывается до выдачи результата);
- программа завершается, но результат выдается неверный;
- выдается неверный результат спустя некоторое время после эксплуатации программы;

В первых двух случаях, когда ошибка явно существует, продолжается отладка. В третьем случае ошибки наиболее неприятные и могут привести к серьезным последствиям, так как обнаруживают себя тогда, когда программа уже сдана в эксплуатацию. Ошибки такого рода возникают в больших программных системах и имеют место в тех логических ветвях программы, которые не были проверены в процессе отладки и редко активизируются при реализации программы.

Следовательно, отладка является важным этапом в процессе разработки программы, влияющим на успешную реализацию программы, причём отладка плохо спроектированной программы может занять более 70% времени разработки. Использование структурных методов разработки программ, включающих правильную организацию отладки, значительно ускоряет процесс отладки и облегчает нахождение ошибок.

Условно ошибки можно разделить на синтаксические и логические.

Синтаксические ошибки состоят в нарушении формальных правил написания программы и появляются в результате недостаточного знания пользователем языка программирования, а также невнимательности при технической подготовке программы к обработке в машине. К синтаксическим ошибкам можно отнести неправильную запись ключевого олова, отсутствие описания массива, пропуск скобки в арифметическом выражении либо инструкции, задающей формат, и т.д.

Логические ошибки подразделяются на ошибки алгоритма и семантические ошибки.

Ошибки алгоритма возникают при несоответствии алгоритма поставленной задаче. Это прежде всего ошибки спецификации, неверная запись расчетной формулы, расходимость итерационного процесса и т.д.

Ошибки семантические (смысловые) являются следствием неправильного понимания программистом смысла инструкций языка программирования или недостаточного знания математического обеспечения (операционной системы). Например, неправильное обращение к устройствам ввода-вывода, неправильное обращение к процедуре, наложение массивов или переменных и т.д.

Отладка состоит из 3-х взаимосвязанных действий:

- контроль правильности программы;
- локализация ошибок, обнаруженных в процессе контроля;
- исправление ошибок.

Перечисленные действия могут многократно повторяться.

Основные действия при отладке. Контроль программы. Фазы контроля.

Действия при отладке:

Отладка состоит из 3-х взаимосвязанных действий:

- контроль правильности программы;
- локализация ошибок, обнаруженных в процессе контроля;

- исправление ошибок.

Перечисленные действия могут многократно повторяться.

Контроль программы - важнейший этап отладки; цель его – обнаружение ошибок. Методика отладки отражает последовательность применения различных методов контроля и состоит из следующих фаз:

1) визуальный контроль текста программы;
2) синтаксический контроль;
3) контроль ограничений структурного программирования;
4) статический семантический контроль;
5) тестирование программы на специально подбираемых тестах.

С точки зрения использования ЭВМ, первая и четвертая фазы относятся к ручному контролю (без ЭВМ), вторая и пятая - к автоматическому (с использованием ЭВМ), а третья - к ручному контролю, если специальных инструментальных средств нет, и к автоматическому - в противном случае. Важно, что приблизительно 55% ошибок выявляется без применения ЭВМ в результате отладки "за столом".

Первые четыре фазы контроля являются контролем текста на конкретном языке программирования. Тестирование является контролем результатов, который базируется на спецификации задачи и логике алгоритма ее решения.

ВИЗУАЛЬНЫЙ КОНТРОЛЬ:

Визуальный контроль осуществляется путем просмотра текста алгоритма или программы с целью определения неправдоподобных или сомнительных конструкций. Этот контроль проводится по перечню шаблонных конструкций и ситуаций, которые следует проверять в программе:

1) Обращение к данным.
2) Описание данных.
3) Вычисления.
4) Операции сравнения.
5) Передачи управления.
6) Межмодульный интерфейс.
7) Инструкция ввода-вывода.

Визуальный контроль существенно сокращает время отладки и уменьшает стоимость отладки, так как помогает выявить и исправить значительную часть ошибок без выхода на машину. Часть этих ошибок предупреждается при анализе аномалий на этапе спецификации.

СИНТАКСИЧЕСКИЙ КОНТРОЛЬ:

Задачей синтаксического контроля является проверка текста программы на соответствие формальному описанию синтаксиса языка программирования. Синтаксический контроль производится с помощью системных средств отладки. Результатом работы системных средств могут быть ин-

формационные и диагностические сообщения, а также дампы (печать состояния памяти).

КОНТРОЛЬ ОГРАНИЧЕНИЙ СТРУКТУРНОГО ПРОГРАММИРОВАНИЯ:

Эта фаза контроля может быть осуществлена автоматически, с помощью ЭВМ, если в ЭВМ существуют специальные инструментальные средства. Этот контроль включает проверку правильности применения управляющих конструкций и проверку стиля программы: рельеф, выбор меток, имен переменных, наличие комментариев и т.д.

СЕМАНТИЧЕСКИЙ КОНТРОЛЬ:

Задачей семантического контроля является проверка правильности применения конструкций языка программирования и выявление в тексте программы конструкций, не формализованных в синтаксисе языка. Статический семантический контроль состоит в исследовании синтаксически правильной программы, основанном на анализе управляющих а информационных связей и выявлении в программе конструкций, сознательное использование которых маловероятно. Такой контроль обычно выявляет ошибки следующих видов:

- недостижимая инструкция, т.е. инструкция, к которой не ведет ни один путь в программе;
- неправильный порядок инструкций ввода-вывода;
- неинициированная переменная, т.е. переменная, которой не было присвоено значение хотя бы на одном пути;
- наличие переменных, которые были описаны, но не используются ни в одной инструкции;
- отсутствие изменения переменных, которые определяют условие завершения цикла.

Статический семантический контроль может быть совмещен с визуальным контролем, если нет специальных инструментальных средств для его автоматизации.

Наиболее эффективным методом тестирования является детерминированное тестирование, при котором известны и контролируются каждая комбинация исходных данных и соответствующие ей результаты исполнения программы.

Детерминированное тестирование основывается на двух подходах: структурное тестирование (СТ) и функциональное тестирование (ФТ).

***Метод трассировки при визуальном и компьютерном способах отладки*.**

МЕТОД ТРАССИРОВКИ:

Цель метода - локализация ошибки, т.е. обнаружение точного места в программе, где находится источник ошибки. Суть этого метода состоит в пошаговом выполнении всех действий, которые предписаны программой.

Трассировка может являться способом визуального контроля и выполняться без помощи компьютера, а также может выполняться с помощью компьютера.

При визуальном контроле все действия в программе выполняются вручную. Значения всех промежуточных переменных записывают в таблицу. Столбцы таблицы соответствуют переменным программы, их имена указывают в заголовках столбцов. Ниже, по мере выполнения программы, записываются значения, которые эти переменные получают на каждом шаге. В конце концов, в таблице окажется весь протокол состояния переменных в ходе выполнения программы.

ПРАКТИЧЕСКИЕ ЗАДАНИЯ.

Задание 1. Ознакомиться с теоретическим материалом по теме.

Задание 2. Определите номер Вашего варианта по формуле:

№ варианта = 1+ остаток от деления на 17 (Целая часть (Число*Месяц*Год Рождения / № в журнале));

Задание 3a. Составить спецификацию для задачи задания 2.

Задание 3b. Написать программу для решения задачи задания 2, согласно спецификации.

Задание 4. Составить план тестирования и отладки для программы из задания 3.

Задание 5. Детализировать и реализовать план отладки из задания 4.

Задание 6. Выбрать программы для тестирования, согласно своего варианта, и изучить их спецификации.

Задание 7. Оформить и сдать отчет по лабораторной работе.

ЛАБОРАТОРНАЯ РАБОТА № 3
Структурные и функциональные методы тестирования

Цель и задачи работы: *Получить навыки структурного и функционального тестирования ПО.*

Теоретический материал

Тестирование программ

Процесс тестирования состоит из трёх этапов:

1. Проектирование тестов.
2. Исполнение тестов.
3. Анализ полученных результатов.

На первом этапе решается вопрос о выборе некоторого подмножества множества тестов, которое сможет найти наибольшее количество ошибок за наименьший промежуток времени. На этапе исполнения тестов проводят, запуск тестов и отлавливают ошибки в тестируемом программном продукте.

Виды тестов

Функциональные тесты составляются на уровне спецификации, до решения задачи. Будущий алгоритм рассматривается как «черный ящик» - функция с неизвестной (или не рассматриваемой) структурой, преобразующая входы в выходы. Суть функциональных тестов: каким бы способом ни решалась задача, при заданных входных значениях должны получиться соответствующие выходные значения.

Структурные тесты составляются для проверки логики решения, или логики работы уже готового алгоритма. Логика определяется последовательностью операций, их условным выполнением или повторением (т. е. композицией базовых конструкций). Совокупность структурных тестов должна обеспечить проверку каждой из таких конструкций.

Чаще всего совокупность тщательно составленных функциональных тестов покрывает множество структурных тестов.

Приведенные понятия различаются тем, что первое рассматривает программу только с точки зрения входов и выходов, тогда как второе относится к ее структуре; но оба понятия не касаются процесса организации тестирования.

Общая последовательность разработки тестов

Наиболее рациональная процедура заключается в том, что сначала разрабатываются функциональные тесты, а затем – структурные.

Функциональное тестирование (тестирование «черного ящика»)

При функциональном тестировании выявляются следующие категории ошибок:

· некорректность или отсутствие функций;
· ошибки интерфейса;
· ошибки в структурах данных;
· ошибки машинных характеристик (нехватка памяти и др.);
· ошибки инициализации и завершения.

Техника тестирования ориентирована:

· на сокращение необходимого количества тестовых вариантов;
· на выявление классов ошибок, а не отдельных ошибок.

Способы функционального тестирования

Разбиение на классы эквивалентности

Это самый популярный способ. Его суть заключается в разделении области входных данных программы на классы эквивалентности и разработке для каждого класса одного тестового варианта.

Класс эквивалентности – набор данных с общими свойствами, в силу чего при обработке любого набора данных этого класса задействуется один и тот же набор операторов [1].

Классы эквивалентности определяются по спецификации программы. Тесты строятся в соответствии с классами эквивалентности, а именно: вы-

бирается вариант исходных данных некоторого класса и определяются соответствующие выходные данные.

Самыми общими классами эквивалентности являются классы допустимых и недопустимых (аномальных) исходных данных. Описание класса строится как комбинация условий, описывающих каждое входное данное.

Условия допустимости или недопустимости данных задают возможные значения данных и могут описывать:

· некоторое конкретное значение; определяется один допустимый и два недопустимых класса эквивалентности: заданное значение, множество значений меньше заданного, множество значений больше заданного;

· диапазон значений; определяется один допустимый и два недопустимых класса эквивалентности: множество значений в границах диапазона; множество значений, выходящих за левую границу диапазона; множество значений, выходящих за правую границу диапазона;

· множество конкретных значений; определяется один допустимый и один недопустимый класс эквивалентности: заданное множество и множество значений, в него не входящих.

Такие классы можно описать языком логики, например, языком исчисления предикатов. Описания более сложных условий и соответствующих классов (например, элементы массива должны находиться в некотором диапазоне и при этом массив не должен содержать нулевых элементов) могут быть построены на основании приведенных выше условий.

В примере, приводимом в вопросе 2 темы 3, при построении тестов неформально использовался описанный метод. В методических целях были выделены только основные класс тестов. Кроме того, исходя из условия задачи, были выделены классы эквивалентности внутри класса правильных данных.

Анализ граничных значений

Этот способ построения тестов дополняет предыдущий и предполагает анализ значений, лежащих на границе допустимых и недопустимых данных. Построение таких тестов часто диктуется интуицией.

Основные правила построения тестов:

· если условие правильности данных задает диапазон, то строятся тесты для левой и правой границы диапазона; для значений чуть левее левой и чуть правее правой границы;

· если условие правильности данных задает дискретное множество значений, то строятся тесты для минимального и максимального значений; для значений чуть меньше минимума и чуть больше максимума;

· если используются структуры данных с переменными границами (массивы), то строятся тесты для минимального и максимального значения границ.

Диаграммы причин-следствий

Взаимосвязь классов эквивалентности и соответствующих им действий описывается формально в виде графа на основе автоматного подхода. Граф преобразуется в таблицу решений, столбцы которой в свою очередь преобразуются в тестовые варианты.

Структурное тестирование (тестирование программ как "белого ящика") предполагает детальное изучение текста (логики) программы и построение (подбор) таких входных наборов данных, которые позволили бы при многократном выполнении программы на ЭВМ обеспечить выполнение максимально возможного количества маршрутов, логических ветвлений, циклов и т.д.

Функциональное тестирование (тестирование программ как "черного ящика") полностью абстрагируется от логики программы, предполагается, что программа - "черный ящик", а тестовые наборы выбираются на основании анализа входных функциональных спецификаций.

Для успешного и качественного проведения детерминированного тестирования необходимо разработать эффективные тестовые наборы данных.

Подмножество всех возможных тестов, которое обладает этим свойством, т.е. имеет наивысшую вероятность обнаружения большинства ошибок, называется **эффективным.**

Чтобы разработать эффективный тестовый набор, необходимо знать ряд методов его построения и придерживаться определенных правил и рекомендаций.

В соответствии с методом детерминированного тестирования при структурном тестировании ориентируются на построение тестовых наборов по принципу "белого ящика", а при функциональном тестировании - по принципу "черного ящика".

При построении тестовых наборов данных по принципу "белого ящика" руководствуются следующими критериями:

Покрытие операторов. Этот критерий предполагает выбор такого тестового набора данных, который вызывает выполнение каждого оператора в программе хотя бы один раз.

Покрытие узлов ветвления (покрытие решений). Этот критерий предполагает разработку такого количества тестов, чтобы в каждом узле ветвления был обеспечен переход по веткам "истина" и "ложь" хотя бы один раз.

Покрытие условий. Если узел ветвления содержит более одного условия, тогда нужно разработать число тестов, достаточное для того, чтобы возможные результаты каждого условия в решении выполнялись, по крайней мере один раз.

Комбинаторное покрытие условий. Этот критерий требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении по крайней мере один раз.

При построении тестов по стратегии "черного ящика" программа рассматривается как "черный ящик", а исходной информацией для тестовых

наборов служат ее спецификации. К стратегии "черного ящика" относятся методы:

Метод эквивалентного разбиения. Построение тестов методов эквивалентного разбиения осуществляется в 2 этапа: 1) выделение классов эквивалентности; 2) построение тестов. Класс эквивалентности множество входных значений, каждое из которых имеет одинаковую вероятность обнаружения конкретного типа ошибки.

Анализ граничных значений. Этот метод предполагает исследование ситуаций, возникающих на границах и вблизи границ эквивалентных разбиений.

Метод функциональных диаграмм. Метод заключается в преобразовании входной спецификации программы в функциональную диаграмму (диаграмму причинно-следственных связей) с помощью простейших булевских отношений, построения таблицы решений, которая является основой для написания эффективных тестовых наборов данных.

ПРАКТИЧЕСКИЕ ЗАДАНИЯ.

Задание 1. Ознакомиться с теоретическим материалом по теме.

Задание 2. Для задач из задания 6 лабораторной работы № 2 написать или найти программы их решения.

Задание 3. Составить планы структурного и функционального тестирования для программ из задания 2.

Задание 4. Детализировать и реализовать планы из задания 3.

Задание 5. Оформить и сдать отчет по лабораторной работе.

Задания для самостоятельной работы

1) Изучите теоретический материал о нефункциональных видах тестирования.
2) Составьте краткий реферат (презентацию) об одном из нефункциональных видов тестирования.
3) Реализуйте тестирование рассмотренное в задании 2 на одной из задач из лабораторной работы № 3.

ЛАБОРАТОРНАЯ РАБОТА № 4
Модульное и интеграционное тестирование

Цель и задачи работы: Получить навыки модульного и интеграционного тестирования.

Теоретический материал

Модульное тестирование – это тестирование программы на уровне отдельно взятых модулей, функций или классов. Цель модульного тестирования состоит в выявлении локализованных в модуле ошибок в реализации

алгоритмов, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования. Модульное тестирование проводится по принципу «белого ящика», то есть основывается на знании внутренней структуры программы, и часто включает те или иные методы анализа покрытия кода.

Интеграционное тестирование предназначено для проверки связи между компонентами, а также взаимодействия с различными частями системы (операционной системой, оборудованием либо связи между различными системами).

Уровни интеграционного тестирования:

- **Компонентный интеграционный уровень** (*ComponentIntegrationtesting*)
  Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.
- **Системный интеграционный уровень** (*SystemIntegrationTesting*)
  Проверяется взаимодействие между разными системами после проведения системного тестирования.

Подходы к интеграционному тестированию:

- **Снизу вверх** (*Bottom Up Integration*)
  Все низкоуровневые модули, процедуры или функции собираются воедино и затем тестируются. После чего собирается следующий уровень модулей для проведения интеграционного тестирования. Данный подход считается полезным, если все или практически все модули, разрабатываемого уровня, готовы. Также данный подход помогает определить по результатам тестирования уровень готовности приложения (см. также Integrationtesting - BottomUp)
- **Сверху вниз** (*Top Down Integration*)
  Вначале тестируются все высокоуровневые модули, и постепенно один за другим добавляются низкоуровневые. Все модули более низкого уровня симулируются заглушками с аналогичной функциональностью, затем по мере готовности они заменяются реальными активными компонентами. Таким образом мы проводим тестирование сверху вниз. (см. также TopDownIntegration)
- **Большой взрыв** (*"Big Bang" Integration*)
  Все или практически все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится интеграционное тестирование. Такой подход очень хорош для сохранения времени. Однако если тест кейсы и их результаты записаны не верно, то сам процесс интеграции сильно осложнится, что станет преградой для команды тестирования при достижении основной цели интеграционного тестирования (см. также Integrationtesting - BigBang)

ПРАКТИЧЕСКИЕ ЗАДАНИЯ

Задание 1. Ознакомиться с теоретическим материалом по теме.

Задание 2. Подберите задачу, программа решения которой состоит из нескольких модулей (от 3 до 7).

Задание 3. Составить спецификацию для задачи задания 2.

Задание 4. Написать или найти программу для решения задачи задания 2, согласно спецификации.

Задание 5. Составить план тестирования 1-2 модулей программы из задания 4.

Задание 6. Детализировать и реализовать план тестирования модулей предыдущего задания.

Задание 7. Составить план интеграционного тестирования программы из задания 4.

Задание 8. Детализировать и реализовать план тестирования предыдущего задания.

Задание 9. Оформить и сдать отчет по лабораторной работе.

ЛАБОРАТОРНАЯ РАБОТА № 5
Системное регрессионное тестирование

Цель и задачи работы: *Получить навыки системного регрессионного тестирования.*

Теоретический материал

Основной задачей системного тестирования является **проверка как функциональных, так и не функциональных требований в системе в целом**. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д. Для минимизации рисков, связанных с особенностями поведения в системы в той или иной среде, **во время тестирования рекомендуется использовать окружение максимально приближенное к тому, на которое будет установлен продукт после выдачи**.

Можно выделить два подхода к системному тестированию:

- **на базе требований** (*requirementsbased*)

  Для каждого требования пишутся тестовые случаи (testcases), проверяющие выполнение данного требования.

- **на базе случаев использования** (*usecasebased*)

  На основе представления о способах использования продукта создаются случаи использования системы (**UseCases**). По конкретному случаю использования можно определить один или более сценариев. На проверку каждого сценария пишутся тест кейсы (testcases), которые должны быть протестированы.

Регрессионное тестирование — цикл тестирования, который производится при внесении изменений на фазе системного тестирования или сопровождения продукта. Главная проблема регрессионного тестирования — выбор между полным и частичным перетестированием и пополнение тес-

товых наборов. При частичном перетестировании контролируются только те части проекта, которые связаны с измененными компонентами. На ГМП это пути, содержащие измененные узлы, и, как правило, это методы и классы, лежащие выше модифицированных по уровню, но содержащие их в своем контексте. Пропуск огромного объема тестов, характерного для этапа системного тестирования, удается осуществить без потери качественных показателей продукта только с помощью регрессионного подхода.

ПРАКТИЧЕСКИЕ ЗАДАНИЯ.

Задание 1. Ознакомиться с теоретическим материалом по теме.

Задание 2. В программу задачи лабораторной работы № 4 внесите мутационные изменения и документируйте их.

Задание 3. Обменяйтесь спецификациями и программами (с изменениями) с Вашим напарником.

Задание 4. Составить план регрессионного тестирования полученной программы из задания 3.

Задание 5. Детализировать и реализовать план тестирования из задания 4, при этом можно пользоваться тестами, как новыми, так и созданные напарником в предыдущей работе.

Задание 7. Сравнить результаты тестирования программы с данными, задокументированными напарником.

Задание 8. Оформить и сдать отчет по лабораторной работе.

ЛАБОРАТОРНАЯ РАБОТА № 6
Автоматизация тестирования

Цель и задачи работы: Получить навыки автоматизации тестирования.

Теоретический материал

Автоматизированное тестирование программного обеспечения (**SoftwareAutomationTesting**) - это процесс верификации программного обеспечения, при котором основные функции и шаги теста, такие как запуск, инициализация, выполнение, анализ и выдача результата, выполняются автоматически при помощи инструментов для автоматизированного тестирования.

Специалист по автоматизированному тестированию программного обеспечения (**SoftwareAutomationTester**) - это технический специалист (тестировщик или разработчик программного обеспечения), обеспечивающий создание, отладку и поддержку работоспособного состояния тест скриптов, тестовых наборов и инструментов для автоматизированного тестирования.

Инструмент для автоматизированного тестирования (**AutomationTestTool**) - это программное обеспечение, посредством которого

специалист по автоматизированному тестированию осуществляет создание, отладку, выполнение и анализ результатов прогона тест скриптов.

Тест Скрипт (**TestScript**) - это набор инструкций, для автоматической проверки определенной части программного обеспечения.

Тестовый набор (**TestSuite**) - это комбинация тест скриптов, для проверки определенной части программного обеспечения, объединенной общей функциональностью или целями, преследуемыми запуском данного набора.

Тесты для запуска (**TestRun**) - это комбинация тест скриптов или тестовых наборов для последующего совместного запуска (последовательного или параллельного, в зависимости от преследуемых целей и возможностей инструмента для автоматизированного тестирования).

Автоматизированное функциональное тестирование ПО (**FunctionalAutomationTesting**)
- это процесс верификации **функциональных требований** и особенностей тестируемого приложения, посредством инструментов для автоматизированного тестирования (см. также Функциональное тестирование).

С автоматизацией тестирования, как и со многими другими узконаправленными IT - дисциплинами, связано много неверных представлений. Для того, чтобы избежать неэффективного применения автоматизации, следует обходить ее недостатки и максимально использовать преимущества. Далее мы перечислим и дадим небольшое описание для основных нюансов автоматизации и дадим ответ на основной вопрос данной статьи – когда автоматизацию все-таки стоит применять.

Преимущества автоматизации тестирования:

- Повторяемость – все написанные тесты всегда будут выполняться однообразно, то есть исключен «человеческий фактор». Тестировщик не пропустит тест по неосторожности и ничего не напутает в результатах.
- Быстрое выполнение – автоматизированному скрипту не нужно сверяться с инструкциями и документациями, это сильно экономит время выполнения.
- Меньшие затраты на поддержку – когда автоматические скрипты уже написаны, на их поддержку и анализ результатов требуется, как правило, меньшее время чем на проведение того же объема тестирования вручную.
- Отчеты – автоматически рассылаемые и сохраняемые отчеты о результатах тестирования.
- Выполнение без вмешательства – во время выполнения тестов инженер-тестировщик может заниматься другими полезными делами, или тесты могут выполняться в нерабочее время (этот метод предпочтительнее, так как нагрузка на локальные сети ночью снижена).

Недостатки автоматизации тестирования (их тоже немало):

- Повторяемость – все написанные тесты всегда будут выполняться однообразно. Это одновременно является и недостатком, так как тестировщик, выполняя тест вручную, может обратить внимание на некоторые детали и, проведя несколько дополнительных операций, найти дефект. Скрипт этого сделать не может.
- Затраты на поддержку – несмотря на то, что в случае автоматизированных тестов они меньше, чем затраты на ручное тестирование того же функционала – они все же есть. Чем чаще изменяется приложение, тем они выше.
- Большие затраты на разработку – разработка автоматизированных тестов это сложный процесс, так как фактически идет разработка приложения, которое тестирует другое приложение. В сложных автоматизированных тестах также есть фреймворки, утилиты, библиотеки и прочее. Естественно, все это нужно тестировать и отлаживать, а это требует времени.
- Стоимость инструмента для автоматизации – в случае если используется лицензионное ПО, его стоимость может быть достаточно высока. Свободно распространяемые инструменты как правило отличаются более скромным функционалом и меньшим удобством работы.
- Пропуск мелких ошибок - автоматический скрипт может пропускать мелкие ошибки на проверку которых он не запрограммирован. Это могут быть неточности в позиционировании окон, ошибки в надписях, которые не проверяются, ошибки контролов и форм с которыми не осуществляется взаимодействие во время выполнения скрипта.

ПРАКТИЧЕСКИЕ ЗАДАНИЯ.

Задание 1. Ознакомиться с теоретическим материалом по теме.

Задание 2. Для программ задач лабораторной работы № 3 составить планы автоматизации их тестирования.

Задание 3. Детализировать и реализовать, насколько это возможно, планы тестирования из задания 2, используя при этом наработанный ранее материал.

Задание 4. Провести автоматизированное тестирование, используя имеющиеся готовые средства автоматизации, для программы из задания 2.

Задание 5. Сравнить результаты тестирования программ в заданиях 3 и 4.

Задание 6*. Используя имеющиеся готовые средства автоматизации и спецификацию, для одной программы из задания 2, постарайтесь найти не соответствие программы и спецификации.

Задание 7. Оформить и сдать отчет по лабораторной работе.

Контрольные вопросы

1.Что такое тестирование ПС?

2.Чем тестирование отличается от отладки ПС?

3.Для чего проводится функциональное тестирование?

4.Что такое комплексное тестирование?

5.Каковы правила тестирования программы «как черного ящика»?

6.Как проводится тестирования программы по принципу «белого ящика»?

7.Что такое модульное тестирование?

8.Как осуществляется сборка программы при модульно тестировании?

Задания для самостоятельной работы

1) Изучите теоретический материал автоматизации тестирования.

2) Составьте краткий реферат (презентацию) о возможностях и методах автоматизации тестирования.

3) Подберите пример программы, для автоматизации тестирования которой используется программа – чекер, реализуйте пример на практике.

Учебное издание

ШЕДЬКО Василий Викторович

НАДЕЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Методические рекомендации
к выполнению лабораторных работ

Технический редактор *Г.В. Разбоева*
Компьютерный дизайн *Е.А. Барышева*

Подписано в печать 2017. Формат 60x84 $^{1}/_{16}$. Бумага офсетная.
Усл. печ. л. 1,51. Уч.-изд. л. 1,19. Тираж экз. Заказ .

Издатель и полиграфическое исполнение – учреждение образования
«Витебский государственный университет имени П.М. Машерова».

Свидетельство о государственной регистрации в качестве издателя,
изготовителя, распространителя печатных изданий
№ 1/255 от 31.03.2014 г.

Отпечатано на ризографе учреждения образования
«Витебский государственный университет имени П.М. Машерова».
210038, г. Витебск, Московский проспект, 33.