5. Тестирование информационной системы (ИС)

Содержание темы:

5.1. Общая характеристика процесса тестирования

5.2. Типы тестирования

5.3. Программные ошибки

5.4. Разработка и выполнение тестов

5.1. Общая характеристика процесса тестирования

Тестирование представляет собой деятельность по проверке программного кода и документации. Она должна заранее планироваться и систематически проводиться специально назначенным сотрудником – ***тестировщиком***. Работа тестировщика начинается до утверждения требований к программному продукту (ПП).

5.1.1. Этапы процесса тестирования:

1) Проверка требований к ПП на полноту.
2) Определение методов тестирования.
3) Разработка стратегии тестирования.
4) Разработка плана тестирования.
5) Создание наборов тестов для проведения интеграционного и системного тестирований *(о них см. далее)*.
6) Создание отчета о тестировании, в котором представляются все результаты проведения тестирования.

5.1.2. Цикл тестирования

Уровни (типы) тестирования:

- модульное;
- интеграционное;
- системное;
- выходное;

} проводятся внутри организации-разработчика ПП

- приемочное – выполняется совместно с представителями заказчика.

5.1.2. Цикл тестирования (продолжение)

Цикл тестирования – совокупность действий, выполняемых тестировщиком с момента передачи базовой версии ПП тестировщику для интеграционного, системного или приемочного тестирования до момента успешного завершения тестирования.

Каждый проход цикла тестирования состоит из следующих этапов:

1) Создание базовой версии ПП.
2) Проведение тестирования.
3) Анализ результатов.
4) Создание отчета о тестировании.
5) Если в процессе тестирования выявлены ошибки, то выполняется их исправление и переход к п.1 (см. выше).
6) Если в процессе тестирования ошибок не выявлено, то цикл тестирования завершается.

5.2. Типы тестирования

1) **Модульное тестирование** представляет собой процесс проверки отдельных подпрограмм, входящих в состав ПП. Оно производится разработчиком ПП и позволяет проверить все структуры и потоки данных в каждом модуле. Включает в себя выявление синтаксических ошибок в программном коде.
2) **Интеграционное тестирование** проводится для проверки совместной работы отдельных модулей и предшествует тестированию всей системы как единого целого. В ходе интеграционного тестирования проверяются связи между модулями, их совместимость и функциональность. Оно осуществляется тестировщиком.

5.2. Типы тестирования (продолжение)

Элементы интеграционного тестирования:

- проверка соответствия отдельных функций, выполняемых совокупностями модулей, функциям, заданным в требованиях к ПП;
- проверка всех промежуточных результатов и файлов на наличие и корректность;
- проверка того, что модули передают друг другу информацию корректно.

Результаты интеграционного тестирования включаются в отчет о ходе тестирования при завершении цикла тестирования.

5.2. Типы тестирования (продолжение)

3) **Системное тестирование** предназначено для проверки функционирования всего ПП на соответствие требованиям заказчика. Системное тестирование проводит тестировщик после успешного завершения интеграционного тестирования.

Элементы системного тестирования:

- тестирование в граничных условиях;
- тестирование всех функциональных характеристик реальной работы системы;
- проверка пользовательской документации на корректность;
- другие тесты, определяемые тестировщиком.

Результаты системного тестирования включаются в отчет о ходе тестирования.

5.2. Типы тестирования (продолжение)

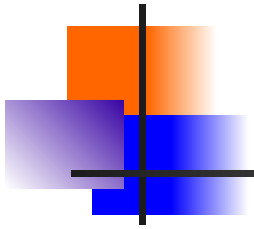
4) **Выходное тестирование** – это завершающий этап тестирования, на котором проверяется готовность ПП к поставке заказчику. Данный вид тестирования проводит тестировщик.

Элементы выходного тестирования:

- проверка на ясность и корректность инструкций по инсталляции;
- проверка того, что вся необходимая документация полностью готова к передаче заказчику.

При успешном завершении выходного тестирования ПП поставляется заказчику вместе с отчетом о результатах тестирования.

5) **Приемочное тестирование** проводится организацией, отвечающей за инсталляцию, сопровождение ПП и обучение пользователей.



5.3. Программные ошибки

Требования к ПП фиксируются в специальном документе – **спецификации требований**.

Два определения программной ошибки:

1) **Программная ошибка** – это расхождение между программой и ее спецификацией.
2) **Программная ошибка** – это ситуация, когда программа не делает того, чего пользователь от нее вполне обоснованно ожидает.

5.3. Программные ошибки (продолжение)

Категории программных ошибок:

1) **Функциональные недостатки.** Программа не делает того, что должна, выполняет некоторые свои функции плохо или не полностью. Функции программы должны быть подробно описаны в ее спецификации, и именно на основе утвержденной спецификации тестировщик строит свою работу.
2) **Недостатки пользовательского интерфейса.** Оценить удобство и правильность работы пользовательского интерфейса можно только в процессе работы с ним. Желательно, чтобы в этой работе принимал участие сам пользователь. Этого можно добиться с помощью разработки прототипа ПП, на котором проводятся «обкатка» и согласование всех требований к пользовательскому интерфейсу с дальнейшей фиксацией их в спецификации требований. После утверждения спецификации требований любые отклонения от неё или невыполнение требований являются ошибкой.

5.3. Программные ошибки (продолжение)

3) **Недостаточная производительность.** Иногда скорость работы ПП задается в требованиях заказчика. Если ПП не удовлетворяет характеристикам производительности, заданным в спецификации требований, это является программной ошибкой.
4) **Некорректная обработка ошибок.** Правильно определив ошибку, программа должна выдать сообщение о ней. Отсутствие такого сообщения является ошибкой в работе программы.
5) **Некорректная обработка граничных условий.** Программа должна быть проверена на исходных данных, находящихся на границах допустимых диапазонов. Для данных, находящихся внутри диапазонов, программа может работать правильно, а на границах диапазонов при этом могут возникать ошибки.
6) **Ошибки вычислений.** Это ошибки, вызванные неправильным выбором алгоритма вычислений, неправильными формулами, а также формулами, неприменимыми к обрабатываемым данным.
7) **Ошибки управления потоком.** Какие-либо действия в программе выполняются в неправильном порядке.

5.3. Программные ошибки (продолжение)

8) **Ситуация гонок.** Предположим, в системе ожидаются два события: А и Б. Если первым наступит событие А, то выполнение программы продолжится, а если событие Б, то в работе программы произойдет сбой. Разработчики предполагают, что первым всегда должно быть событие А, и не ожидают, что Б может выиграть гонки и наступить раньше. Ситуации гонок наиболее типичны для многопользовательских систем реального времени.
9) **Перегрузки.** Сбои в работе программы могут происходить из-за нехватки памяти или отсутствия других системных ресурсов. Программа может не справляться, например, со слишком большими объемами данных.
10) **Некорректная работа с аппаратурой.** Программа может посылать аппаратным устройствам неверные данные, игнорировать сообщения устройств об ошибках, пытаться использовать устройства, которые заняты или отсутствуют. Если нужное устройство сломано, программа должна понять это, а не «зависать» при попытке обратиться к нему.

5.4. Разработка и выполнение тестов

Требования к хорошему тесту:

1) **Должна быть достаточной вероятность выявления тестом ошибки.** Разрабатывая тестовые примеры, необходимо проанализировать все возможные варианты сбоев программы или ее некорректной работы.
2) **Набор тестов не должен быть избыточным.** Если два теста предназначены для выявления одной и той же ошибки, то достаточно выполнить только один из них.
3) **Тест должен быть наилучшим в своей категории.** В группе похожих тестов одни могут быть эффективнее других. Поэтому нужно выбирать тот тест, который с наибольшей вероятностью выявит ошибку.
4) **Тест не должен быть слишком простым или слишком сложным.** Большой и сложный тест трудно понять, выполнить и долго создавать. Поэтому лучше всего разрабатывать простые, но всё же не совсем элементарные тестовые примеры.

5.4.1. Классы эквивалентности

Класс эквивалентности – это набор тестов, от выполнения которых ожидается один и тот же результат. В простейшем случае тест представляет собой набор входных данных, вводимых в тестируемую программу. В случае эквивалентных тестов эти данные обладают общими свойствами.

Группа тестов представляет собой класс эквивалентности, если выполняются следующие условия:

- все тесты предназначены для выявления одной и той же ошибки;
- если один из тестов выявит ошибку, то остальные тоже это сделают;
- если один из тестов не выявит ошибку, то остальные тоже этого не сделают.

Необходимо стремиться выявить как можно больше классов эквивалентности. Это сэкономит время тестирования и сделает тестирование более эффективным, избавив тестировщика от повторения эквивалентных тестов. Разбив все предполагаемые тесты на классы, можно затем выделить в каждом из них один или несколько наиболее эффективных тестов; остальные тесты выполнять незачем.

5.4.1. Классы эквивалентности (продолжение)

Рекомендации для поиска классов эквивалентности:

- *Классы эквивалентности должны охватывать заведомо недопустимые входные данные.* Часто такие входные данные вызывают в программе разнообразные ошибки. Поэтому чем больше типов неверного ввода выделить, тем больше ошибок можно найти.
- *Следует определить допустимые диапазоны числовых значений.* Обычно для числовых значений имеются три недопустимых класса эквивалентности: все числа, которые меньше нижнего граничного значения диапазона; все числа, которые больше его верхнего граничного значения; нечисловые данные. Иногда один из этих классов отсутствует. Например, возможен ввод любого числа. В этом случае при тестировании необходимо убедиться, что это на самом деле так. Следует попробовать ввести очень большое число.

5.4.1. Классы эквивалентности (продолжение)

- *Перечень классов эквивалентности лучше организовать в виде таблицы.* Обычно классов эквивалентности оказывается много, поэтому нужен удобный и продуманный способ организации собранной информации. Пример перечня классов эквивалентности:

Входное событие	Допустимые классы эквивалентности	Недопустимые классы эквивалентности
Ввод числа	Числа от 1 до 99	Число 0 Числа больше 99 Выражение, результатом которого является недопустимое число (например: 5 – 5 = 0) Отрицательные числа Буквы и др. нечисловые символы
Ввод первой буквы наименования	Заглавная буква Прописная буква	Не буква

5.4.1. Классы эквивалентности (продолжение)

- *Следует проанализировать возможные результаты выбора из списков и меню.* Любой элемент предложенного программой списка опций может представлять собой отдельный класс эквивалентности. Каждый элемент меню или списка опций обрабатывается программой особым образом, поэтому все они подлежат проверке.
- *Следует выявить группы переменных, совместно участвующих в определенных вычислениях, результат которых ограничивается конкретным диапазоном значений.* Например, к классу допустимых относятся величины трех углов треугольника, в сумме дающие 180°. Недопустимые значения можно разделить на два класса эквивалентности: с суммарным значением менее 180° и более 180°.
- *Следует продумать варианты операционного окружения.* Бывает, что программа хорошо работает только при определенных типах мониторов, принтеров, модемов, дисковых устройств или любого другого подключенного к системе оборудования. Поэтому важно определить классы эквивалентных конфигураций системы.

5.4.2. Границы классов эквивалентности

Для каждого класса эквивалентности достаточно провести один-два теста. Лучшими из них будут те, которые проверяют значения, лежащие на границах класса. Неправильные операторы сравнения (например, > вместо ≥) вызывают ошибки только при граничных значениях аргументов. В то же время программа, которая сбоит при промежуточных значениях диапазона, почти наверняка будет сбоить и при его граничных значениях.

Необходимо протестировать каждую границу класса эквивалентности с обеих сторон. Программа, которая пройдет эти тесты, скорее всего, пройдет и все остальные, относящиеся к данному классу.

5.4.2. Границы классов эквивалентности (продолжение)

Примеры:

- Если допустимы значения от **1** до **99**, то для тестирования допустимых данных можно выбрать **1** и **99**, а для тестирования недопустимых – **0** и **100**.
- Если программа ожидает ввода заглавной латинской буквы, лучше ввести **A** и **Z**. Также следует проверить символ **@** (т.к. его код предшествует коду символа **A**) и символ **[** (код которого следует за кодом символа **Z**). Кроме того, следует проверить символы **a** и **z**.
- Если сумма входных значений должна равняться **180**, следует попробовать ввести значения, дающие в сумме **179**, **180** и **181**.
- Следует попробовать отправить на печать файл непосредственно перед тем, как принтер напечатает еще чье-либо задание, и сразу после этого.

5.4.3. Тестирование переходов между состояниями

Пример переходов между состояниями: меню. После запуска программы в нем имеется один перечень команд. После выбора одной из них состояние программы меняется и в меню появляются команды, доступные в этом новом состоянии. Необходимо протестировать каждую команду меню. Например, команда 10 может быть доступна в режиме, открываемом по команде 9 или по команде 22. В этом случае команду 10 придется протестировать дважды – в обоих режимах.

Содержимое и структура очередной формы ввода данных могут зависеть от информации, введенной в предыдущей форме, значения одних полей могут определять допустимые значения других, ввод определенной информации может инициировать серию дополнительных запросов. Например, при вводе чисел от 1 до 99 программа выводит одну форму запроса, обращенного к пользователю, а при вводе любых других чисел – другую. В этом случае следует проанализировать возможные пути выполнения программы, чтобы составить полноценный набор тестов.

5.4.3. Тестирование переходов между состояниями (продолжение)

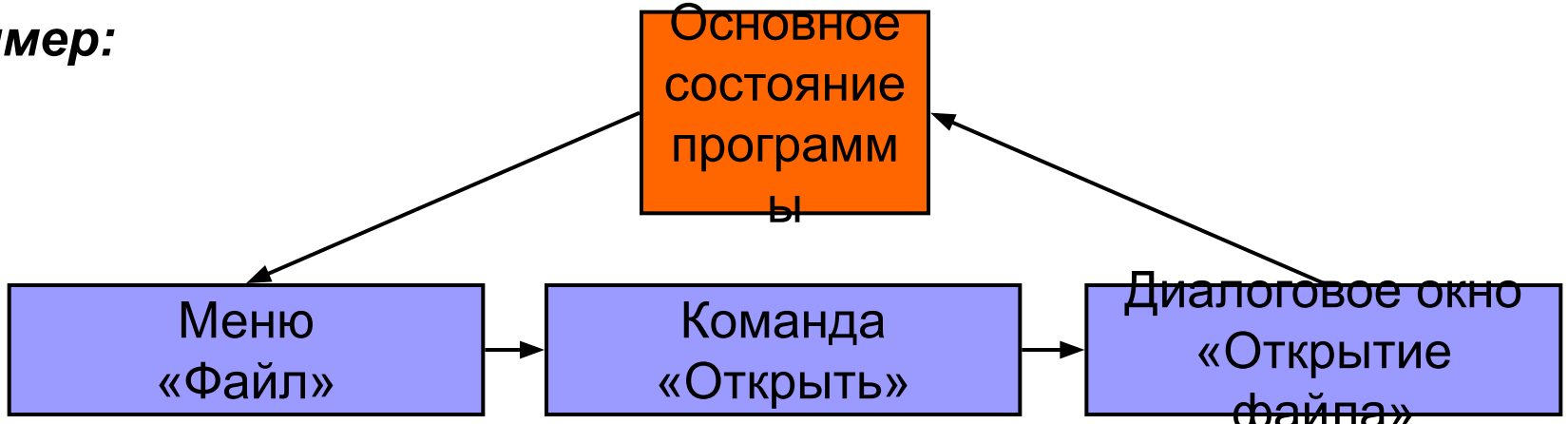
Команд меню, режимов программы и путей перехода в эти режимы может быть так много, что протестировать их все просто нереально. Поэтому, ***отбирая тесты для проверки путей выполнения программы, лучше всего руководствоваться следующими принципами:***

- тестировать все наиболее вероятные последовательности действий пользователей;
- если можно предположить, что действия пользователя в одном режиме могут влиять на представление данных или набор предоставляемых программой возможностей в другом режиме, тестировать эти действия;
- поработать с программой в произвольном режиме, случайным образом выбирая путь ее выполнения.

5.4.3. Тестирование переходов между состояниями (продолжение)

Очень полезно составление ***схем меню***. На такой схеме отражаются все состояния программы и команды, вызывающие переходы между этими состояниями. В нее включаются команды, активизируемые через меню, кнопки, а также через нажатия определенных клавиш на клавиатуре.

Пример:



Особенно удобны подобные схемы в случае, если определенное диалоговое окно можно открыть несколькими способами и выйти из него в несколько различных режимов. В этом случае можно изобразить на схеме все направления переходов и по ним протестировать программу.

5.4.4. Условия гонок и другие временны*е* зависимости

Способы тестирования:

- Вмешательство в работу программы, когда она выполняет операции обработки данных или ввода-вывода (например, нажатия на различные клавиши, выбор каких-либо пунктов меню, попытка параллельного ввода или вывода еще какой-либо информации).
- Проверка реакции программы на действия пользователя или наступление ожидаемого события на границах интервала тайм-аута. **Ситуация тайм-аута** – это ситуация, когда программа ждет определенного события в течение заданного времени, а затем переходит в другое состояние. Следует проверить, что будет, если событие произойдет за секунду до того, как программа должна прекратить ожидать его, или через секунду после этого.

5.4.4. Условия гонок и другие временные зависимости (продолжение)

- Тестирование при повышенной нагрузке:
  - запуск нескольких других программ;
  - отправка большого файла на принтер;
  - одновременная работа различных внешних устройств.

  В результате тестирования при повышенной нагрузке программа будет выполняться медленнее, и, быстро вводя данные, можно превысить ее возможности приема этих данных, что приведет к сбою программы.

  Главная задача тестирования при повышенной нагрузке: обеспечить такую надежность разрабатываемого ПП, чтобы он работал (пусть медленно, но без сбоев) при любых дополнительных нагрузках. Необходимо выяснить, какие конфигурации системы являются предельными для эксплуатации ПП.

5.4.5. Нагрузочные испытания

Важно протестировать те ограничения возможностей ПП, которые определены в его документации.

Примеры нагрузочных испытаний:

- открытие максимального количества файлов, с которыми программа может работать; эксплуатация ее в таком состоянии;
- проверка логически допустимых значений каких-либо параметров (например, ввод очень больших и очень маленьких числовых значений), даже если в документации эти ограничения не описаны;
- проверка того, как ведет себя программа, когда исчерпываются различные аппаратные ресурсы (например, переполняется диск, в принтере заканчивается бумага, остается очень мало свободной оперативной памяти).

5.4.5. Нагрузочные испытания (продолжение)

Нагрузочное тестирование – это один из видов тестирования граничных условий. Схема его проведения аналогична: сначала программу запускают в условиях, в которых она должна работать, а затем – в условиях, для которых она не предназначена. Имеет смысл проверить и различные комбинации условий: возможно, что, справившись с различными повышенными нагрузками по-отдельности, программа не выдержит их всех вместе. Нагрузив систему, следует провести достаточно длительное и обстоятельное тестирование.

5.4.6. Прогнозирование ошибок

Иногда тестировщик предполагает, что определенный класс тестов вызовет сбой программы, хотя и не может это логически обосновать. Следует доверять своей интуиции и использовать подобные тесты. Существует целый ряд ситуаций и значений, которые часто вызывают программные сбои, хотя и не являются граничными (например, число 0).

5.4.7. Тестирование функциональной эквивалентности

Предположим, что тестируется программа, которая вычисляет некоторую математическую функцию (например, тригонометрическую, обращающую матрицу, возвращающую коэффициенты для построения кривой, отражающей некоторый набор данных). Обычно можно найти другую программу, вычисляющую ту же функцию, и при этом достаточно надежную и проверенную временем. С помощью обеих программ обрабатывают одинаковые наборы входных данных. Если результаты совпадают, значит, тестируемая программа работает правильно.

Можно автоматизировать весь процесс тестирования: от ввода входных данных до сравнения выходных. Результаты расчетов можно выводить в файлы и сравнивать их с помощью специально написанной программы. При этом можно определить и допустимые расхождения результатов, например, погрешности округлений.

5.4.8. Регрессионное тестирование

***Два значения термина «*Регрессионное тестирование»**, которые объединяет идея повторного использования разработанных тестов:

1) повторное проведение теста после исправления ошибки в программе, чтобы убедиться, что ошибки больше нет; тестируется только исправленный фрагмент программы;
2) проведение стандартной серии тестов после исправления ошибки в программе, чтобы убедиться, что, исправив одну часть программы, программист не испортил другую; тестируется вся программа целиком.

5.4.8. Регрессионное тестирование (продолжение)

Последовательность действий программиста при исправлении ошибки:

1) найти причину ошибки, указанной в отчете о тестировании; для этого следует тщательно проанализировать исходный код программы;
2) исправить ошибку;
3) протестировать результат.

5.4.8. Регрессионное тестирование (продолжение)

Задачи тестировщика, проверяющего исправления, внесенные программистом:

1) *Убедиться, что ошибка исправлена.* Для этого следует выполнить тот же тест, в котором проявилась ошибка и который описан в отчете о тестировании. Если программа не пройдет этот тест, дальнейшее тестирование не имеет смысла.
2) *Постараться найти связанные ошибки.* Возможно, программист устранил описанные в отчете симптомы ошибки, но саму ее не исправил. Можно попробовать спровоцировать симптомы ошибки каким-нибудь другим способом.
3) *Протестировать оставшуюся часть программы.* Последствия исправления ошибки могут проявиться где-нибудь еще. Необходимо подумать, какие части программы могут быть затронуты внесенными исправлениями, и проверить их.

5.4.8. Регрессионное тестирование (продолжение)

Библиотека регрессионных тестов – это полный набор тестов, охватывающий всю программу и выполняющийся каждый раз, когда программисты сдают ее очередную версию.

Рекомендации по регрессионному тестированию:

- *Уменьшить число тестов, объектом которых является уже исправленная ошибка.* Если ошибка проявляется в течение ряда циклов тестирования, то в библиотеку следует добавить достаточное число тестов для выявления этой ошибки. Соответствующую часть программы необходимо тестировать до тех пор, пока в ней не останется ошибки. Однако после этого большую часть тестов, направленных на поиск исправленной ошибки, можно удалить из библиотеки.
- *Автоматизировать тестирование.* Если определенная группа тестов будет выполняться в течение 5 – 10 циклов тестирования, то стоит потратить время на автоматизацию.

5.4.8. Регрессионное тестирование (продолжение)

- *Выделить часть тестов для периодического выполнения.* Все тесты регрессионной библиотеки не обязательно выполнять после каждого изменения программы. Это можно делать реже: на каждом 2 – 3 цикле. На последней стадии тестирования лучше выполнить максимально возможное число тестов, чтобы убедиться, что программа готова к выпуску. На других циклах достаточно 1/2 или 1/3 всех тестов.

5.4.8. Регрессионное тестирование (продолжение)

Основное правило выполнения тестов: процесс тестирования должен заставить программу использовать введенные данные и подтвердить, что они используются правильно.

Примеры выполнения тестов:

- Если при установке программы на компьютер у пользователя есть возможность выбора конфигурации, то при тестировании недостаточно просто запустить программу установки и посмотреть, предоставляет ли она необходимые опции. Следует выполнить каждый вариант установки, всякий раз запуская программу и проверяя, действительно ли установлена выбранная конфигурация. Следует убедиться, что программа при этом полностью работоспособна.
- Если программа позволяет указать размеры печатаемой страницы, полей и др. параметры страницы, то при тестировании недостаточно просто увидеть, что документ правильно выглядит на экране. Его необходимо распечатать.

5.4.9. Технологии тестирования:

1) **Тестирование «черного ящика» (black box).** Программа рассматривается как объект, внутренняя структура которого неизвестна. Тестировщик вводит данные и анализирует результат, но он не знает, как именно работает программа. Подбирая тесты, тестировщик ищет входные данные, при которых с наибольшей вероятностью могут проявиться ошибки тестируемой программы.
2) **Тестирование «белого ящика» (white box)**, *другое название* – **тестирование «стеклянного ящика» (glass box).** Представляет собой часть процесса программирования. Программист тестирует каждый модуль после его написания *(модульное тестирование; о нем см. подраздел 5.2)*. Он разрабатывает тесты, основываясь на знании исходного кода, к которому имеет полный доступ.

5.4.9. Технологии тестирования (продолжение)

Преимущества тестирования «белого ящика»:

- *Направленность тестирования.* Программист может тестировать программу по частям, разрабатывать специальные тестовые подпрограммы, которые вызывают тестируемый модуль и передают ему интересующие программиста данные.
- *Полный охват кода.* Программист всегда может определить, какие именно фрагменты кода работают в каждом тесте. Он видит, какие еще ветви кода остались непротестированными, и может подобрать условия, в которых они будут протестированы.
- *Возможность управления потоком команд.* Чтобы выяснить, работает ли программа так, как он думает, программист может включить в нее отладочные команды, отображающие информацию о ходе ее выполнения, или воспользоваться для этого специальным программным средством – отладчиком (режим «Отладчик» в среде «1С:Предприятие»). С помощью отладчика можно отслеживать последовательность выполнения команд программы, показывать содержимое ее переменных.

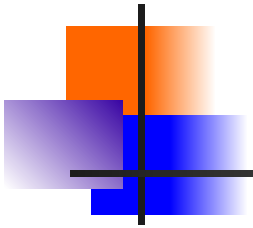
5.4.9. Технологии тестирования (продолжение)

Последовательность разработки тестов для тестирования «белого ящика»:

- Разрабатываются тесты для каждого класса эквивалентности, а также для граничных и особых значений входных данных *(см. пункты 5.4.1 и 5.4.2)*.
- Контролируется, все ли классы выходных данных при этом проверены; при необходимости добавляются новые тесты.
- Разрабатываются тесты для тех подпрограмм, которые не проверяются в 1-ом пункте.
- По тексту программы проверяется, все ли условные переходы выполнены в каждом направлении (ТО…, ИНАЧЕ…); при необходимости добавляются новые тесты.
- По тексту программы проверяется, проходятся ли пути для каждого цикла: без выполнения тела цикла, с однократным повторением, с максимальным числом повторений; при необходимости добавляются новые тесты.
- Разрабатываются тесты, проверяющие исключительные ситуации, недопустимые входные данные.

Литература
(имеется в библиотеке КТИ)

1. Технология разработки программных продуктов: учеб. пособие для студентов учреждений сред. проф. образования / А.В. Рудаков. – М.: Издательский центр «Академия», 2010. – 208 с.
2. Технология разработки программных продуктов. Практикум: учеб. пособие для студентов учреждений сред. проф. образования / А.В. Рудаков, Г.Н. Федорова. – М.: Издательский центр «Академия», 2010. – 192 с.



Контрольные вопросы

1. Дать общую характеристику процессу тестирование.
2. Из каких этапов состоит тестирование?
3. Какие существуют уровни тестирования?
4. Что такое цикл тестирования?
5. Из каких этапов состоит каждый цикл тестирования?
6. Какие типы тестирования существуют?
7. Дать определения программной ошибке.
8. Какие существуют категории программных ошибок?
9. Какие существуют требования к хорошему тесту?
10. Что такое класс эквивалентности?