Методика тестирования программных систем

Процесс тестирования объединяет различные способы тестирования в спланированную последовательность шагов, которые приводят к успешному построению программной системы (ПС) [3], [13], [64], [69]. Методика тестирования ПС может быть представлена в виде разворачивающейся спирали (рис. 8.1).

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

Охарактеризуем каждый шаг процесса тестирования.

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

Рис. 8.1. Спираль процесса тестирования ПС

2. *Тестирование интеграции.* Цель — тестирование сборки модулей в программную систему. В основном применяют способы тестирования «черного ящика».
3. *Тестирование правильности.* Цель — проверить реализацию в программной системе всех функциональных и поведенческих требований, а также требования эффективности. Используются исключительно способы тестирования «черного ящика».
4. *Системное тестирование.* Цель — проверка правильности объединения и взаимодействия всех элементов компьютерной системы, реализации всех системных функций.

Организация процесса тестирования в виде эволюционной разворачивающейся спирали обеспечивает максимальную эффективность поиска ошибок. Однако возникает вопрос — когда заканчивать тестирование?

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

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

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

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

устраняемых ошибок; *t* —время тестирования.

С помощью уравнения (8.1) можно предсказать снижение ошибок в ходе тестирования, а также время, требующееся для достижения допустимо низкой интенсивности отказов.

Тестирование элементов

Объектом тестирования элементов является наименьшая единица проектирования ПС — модуль. Для обнаружения ошибок в рамках модуля тестируются его важнейшие управляющие пути. Относительная сложность тестов и ошибок определяется как результат ограничений области тестирования элементов. Принцип тестирования — «белый ящик», шаг может выполняться для набора модулей параллельно.

Тестированию подвергаются:

- интерфейс модуля;
- внутренние структуры данных;
- независимые пути;
- пути обработки ошибок;
- граничные условия.

Интерфейс модуля тестируется для проверки правильности ввода-вывода тестовой информации. Если нет уверенности в правильном вводе-выводе данных, нет смысла проводить другие тесты.

Исследование внутренних структур данных гарантирует целостность сохраняемых данных.

Тестирование независимых путей гарантирует однократное выполнение всех операторов модуля. При тестировании путей выполнения обнаруживаются следующие категории ошибок: ошибочные вычисления, некорректные сравнения, неправильный поток управления [3].

Наиболее общими ошибками вычислений являются:

1) неправильный или непонятый приоритет арифметических операций;
2) смешанная форма операций;
3) некорректная инициализация;
4) несогласованность в представлении точности;
5) некорректное символическое представление выражений.

Источниками ошибок сравнения и неправильных потоков управления являются:

1) сравнение различных типов данных;
2) некорректные логические операции и приоритетность;
3) ожидание эквивалентности в условиях, когда ошибки точности делают эквивалентность невозможной;
4) некорректное сравнение переменных;
5) неправильное прекращение цикла;
6) отказ в выходе при отклонении итерации;
7) неправильное изменение переменных цикла.

Обычно при проектировании модуля предвидят некоторые ошибочные условия. Для защиты от ошибочных условий в модуль вводят пути обработки ошибок. Такие пути тоже должны тестироваться. Тестирование путей обработки ошибок можно ориентировать на следующие ситуации:

1) донесение об ошибке невразумительно;
2) текст донесения не соответствует, обнаруженной ошибке;
3) вмешательство системных средств регистрации аварии произошло до обработки ошибки в модуле;
4) обработка исключительного условия некорректна;
5) описание ошибки не позволяет определить ее причину.

И, наконец, перейдем к граничному тестированию. Модули часто отказывают на

«границах». Это означает, что ошибки часто происходят:

1) при обработке *n*-го элемента *n*-элементного массива;

2) при выполнении *m*-й итерации цикла с *m* проходами;

3) при появлении минимального (максимального) значения.

Тестовые варианты, ориентированные на данные ситуации, имеют высокую вероятность обнаружения ошибок.

Тестирование элементов обычно рассматривается как дополнение к этапу кодирования. Оно начинается после разработки текста модуля. Так как модуль не является автономной системой, то для реализации тестирования требуются дополнительные средства, представленные на рис. 8.2.

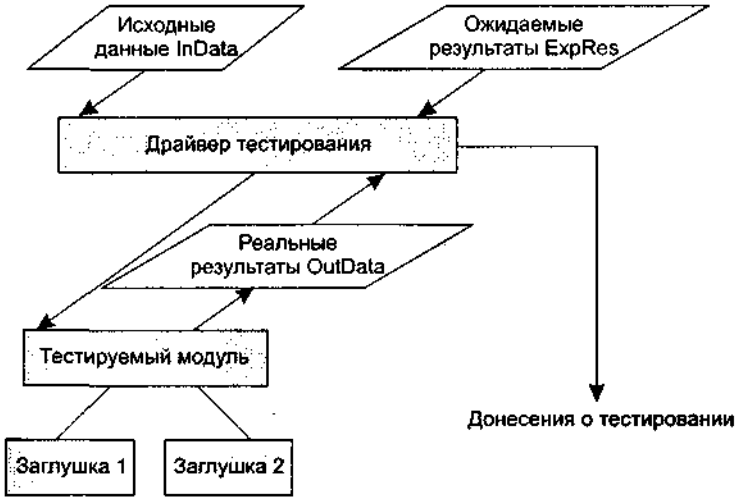

Рис. 8.2. Программная среда для тестирования модуля

Дополнительными средствами являются драйвер тестирования и заглушки. Драйвер — управляющая программа, которая принимает исходные данные (InData) и ожидаемые результаты (ExpRes) тестовых вариантов, запускает в работу тестируемый модуль, получает из модуля реальные результаты (OutData) и формирует донесения о тестировании. Алгоритм работы тестового драйвера приведен на рис. 8.3.

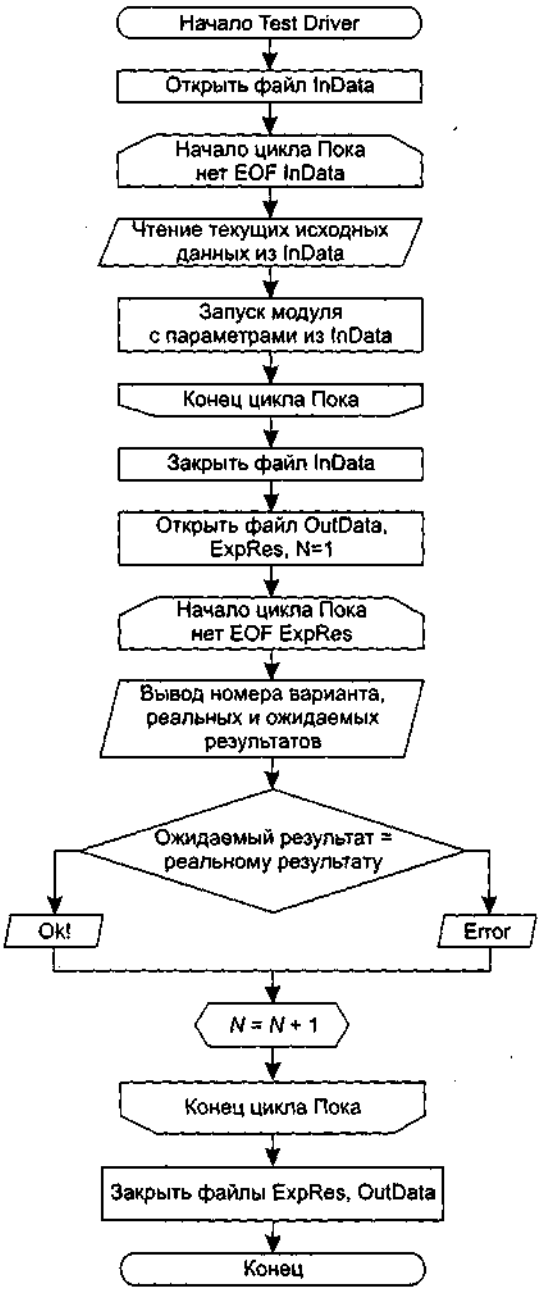

Рис. 8.3. Алгоритм работы драйвера тестирования

Заглушки замещают модули, которые вызываются тестируемым модулем. Заглушка, или «фиктивная подпрограмма», реализует интерфейс подчиненного модуля, может выполнять минимальную обработку данных, имитирует прием и возврат данных.

Создание драйвера и заглушек подразумевает дополнительные затраты, так как они не поставляются с конечным программным продуктом.

Если эти средства просты, то дополнительные затраты невелики. Увы, многие модули не могут быть адекватно протестированы с помощью простых дополнительных средств. В этих случаях полное тестирование может быть отложено до шага тестирования интеграции (где драйверы или заглушки также используются).

Тестирование элемента просто осуществить, если модуль имеет высокую связность. При реализации модулем только одной функции количество тестовых вариантов уменьшается, а ошибки легко предсказываются и обнаруживаются.

Тестирование интеграции

Тестирование интеграции поддерживает сборку цельной программной системы.

Цель сборки и тестирования интеграции: взять модули, протестированные как элементы, и построить программную структуру, требуемую проектом [3].

Тесты проводятся для обнаружения ошибок интерфейса. Перечислим некоторые категории ошибок интерфейса:

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

Существует два варианта тестирования, поддерживающих процесс интеграции: нисходящее тестирование и восходящее тестирование. Рассмотрим каждый из них.

Нисходящее тестирование интеграции

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

Рассмотрим пример (рис. 8.4). Интеграция поиском в глубину будет подключать все модули, находящиеся на главном управляющем пути структуры (по вертикали). Выбор главного управляющего пути отчасти произволен и зависит от характеристик, определяемых приложением. Например, при выборе левого пути прежде всего будут подключены модули Ml, M2, M5. Следующим подключается модуль M8 или Мб (если это необходимо для правильного функционирования M2). Затем строится центральный или правый управляющий путь.

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

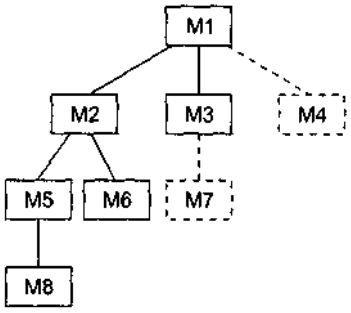

Рис. 8.4. Нисходящая интеграция системы

Опишем возможные шаги процесса нисходящей интеграции.

1. Главный управляющий модуль (находится на вершине иерархии) используется как тестовый драйвер. Все непосредственно подчиненные ему модули временно замещаются заглушками.
2. Одна из заглушек заменяется реальным модулем. Модуль выбирается поиском в ширину или в глубину.
3. После подключения каждого модуля (и установки на нем заглушек) проводится набор тестов, проверяющих полученную структуру.
4. Если в модуле-драйвере уже нет заглушек, производится смена модуля-драйвера (поиском в ширину или в глубину).
5. Выполняется возврат на шаг 2 (до тех пор, пока не будет построена целая

структура).

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

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

Существуют 3 возможности борьбы с этим недостатком:

1) откладывать некоторые тесты до замещения заглушек модулями;

2) разрабатывать заглушки, частично выполняющие функции модулей;

3) подключать модули движением снизу вверх.

Первая возможность вызывает сложности в оценке результатов тестирования.

Для реализации второй возможности выбирается одна из следующих категорий заглушек:

- заглушка A — отображает трассируемое сообщение;
- заглушка B — отображает проходящий параметр;
- заглушка C — возвращает величину из таблицы;
- заглушка D — выполняет табличный поиск по ключу (входному параметру) и возвращает связанный с ним выходной параметр.

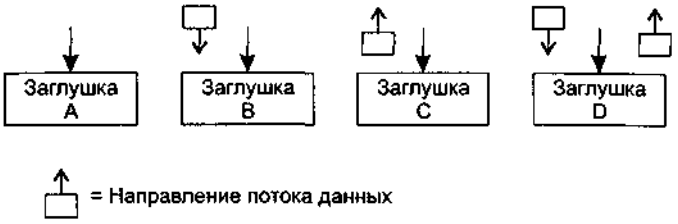

Рис. 8.5. Категории заглушек

Категории заглушек представлены на рис. 8.5.

Очевидно, что заглушка A наиболее проста, а заглушка D наиболее сложна в реализации.

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

Третью возможность обсудим отдельно.

Восходящее тестирование интеграции

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

Рассмотрим шаги методики восходящей интеграции.

1. Модули нижнего уровня объединяются в кластеры (группы, блоки), выполняющие определенную программную подфункцию.
2. Для координации вводов-выводов тестового варианта пишется драйвер, управляющий тестированием кластеров.
3. Тестируется кластер.
4. Драйверы удаляются, а кластеры объединяются в структуру движением вверх. Пример восходящей интеграции системы приведен на рис. 8.6.

Модули объединяются в кластеры 1,2,3. Каждый кластер тестируется драйвером. Модули в кластерах 1 и 2 подчинены модулю Ma, поэтому драйверы D1 и D2 удаляются и кластеры подключают прямо к Ma. Аналогично драйвер D3 удаляется перед подключением кластера 3 к модулю Mb. В последнюю очередь к модулю Mc подключаются модули Ma и Mb.

Рассмотрим различные типы драйверов:

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

Очевидно, что драйвер A наиболее прост, а драйвер D наиболее сложен в реализации. Различные типы драйверов представлены на рис. 8.7.

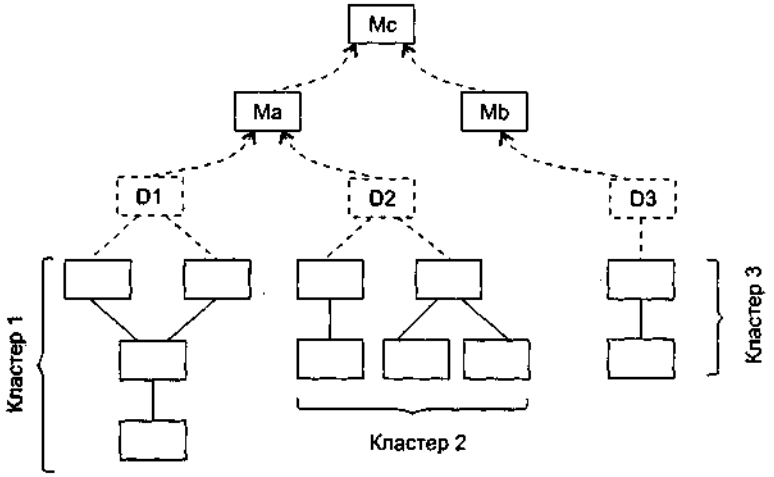

Рис. 8.6. Восходящая интеграция системы

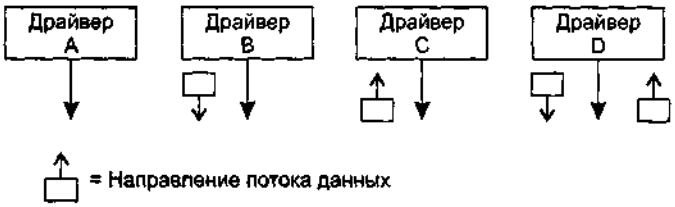

Рис. 8.7. Различные типы драйверов

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

Сравнение нисходящего и восходящего тестирования интеграции

Нисходящее тестирование:

1) *основной недостаток*— необходимость заглушек и связанные с ними трудности тестирования;

2) *основное достоинство* — возможность раннего тестирования главных управляющих функций.

Восходящее тестирование:

1) *основной недостаток* — система не существует как объект до тех пор, пока не будет добавлен последний модуль;
2) *основное достоинство* — упрощается разработка тестовых вариантов, отсутствуют заглушки.

Возможен комбинированный подход. В нем для верхних уровней иерархии применяют нисходящую стратегию, а для нижних уровней — восходящую стратегию тестирования [3], [13].

При проведении тестирования интеграции очень важно выявить критические модули. Признаки критического модуля:

1) реализует несколько требований к программной системе;
2) имеет высокий уровень управления (находится достаточно высоко в программной структуре);
3) имеет высокую сложность или склонность к ошибкам (как индикатор может

использоваться цикломатическая сложность — ее верхний разумный предел составляет 10);

4) имеет определенные требования к производительности обработки.

Критические модули должны тестироваться как можно раньше. Кроме того, к ним должно применяться регрессионное тестирование (повторение уже выполненных тестов в полном или частичном объеме).

Тестирование правильности

После окончания тестирования интеграции программная система собрана в единый корпус, интерфейсные ошибки обнаружены и откорректированы. Теперь начинается последний шаг программного тестирования — *тестирование правильности.* Цель — подтвердить, что функции, описанные в спецификации требований к ПС, соответствуют ожиданиям заказчика [64], [69].

Подтверждение правильности ПС выполняется с помощью тестов «черного ящика», демонстрирующих соответствие требованиям. При обнаружении отклонений от спецификации требований создается список недостатков. Как правило, отклонения и ошибки, выявленные при подтверждении правильности, требуют изменения сроков разработки продукта.

Важным элементом подтверждения правильности является проверка конфигурации ПС. Конфигурацией ПС называют совокупность всех элементов информации, вырабатываемых в процессе конструирования ПС. Минимальная конфигурация ПС включает следующие базовые элементы:

1) системную спецификация;
2) план программного проекта;
3) спецификацию требований к ПС; работающий или бумажный макет;
4) предварительное руководство пользователя;
5) спецификация проектирования;
6) листинги исходных текстов программ;
7) план и методику тестирования; тестовые варианты и полученные результаты;
8) руководства по работе и инсталляции;
9) exe-код выполняемой программы;
10) описание базы данных;
11) руководство пользователя по настройке;
12) документы сопровождения; отчеты о проблемах ПС; запросы сопровождения; отчеты о конструкторских изменениях;
13) стандарты и методики конструирования ПС.

Проверка конфигурации гарантирует, что все элементы конфигурации ПС правильно разработаны, учтены и достаточно детализированы для поддержки этапа сопровождения в жизненном цикле ПС.

Разработчик не может предугадать, как заказчик будет реально использовать ПС. Для обнаружения ошибок, которые способен найти только конечный пользователь, используют процесс, включающий альфа- и бета-тестирование.

Альфа-тестирование проводится заказчиком в организации разработчика. Разработчик фиксирует все выявленные заказчиком ошибки и проблемы использования.

Бета-тестирование проводится конечным пользователем в организации заказчика. Разработчик в этом процессе участия не принимает. Фактически, бета-тестирование — это реальное применение ПС в среде, которая не управляется разработчиком. Заказчик сам записывает все обнаруженные проблемы и сообщает о них разработчику. Бета-тестирование проводится в течение фиксированного срока (около года). По результатам выявленных проблем разработчик изменяет ПС и тем самым подготавливает продукт полностью на базе заказчика.

Системное тестирование

Системное тестирование подразумевает выход за рамки области действия программного проекта и проводится не только программным разработчиком. Классическая проблема системного тестирования — указание причины. Она возникает, когда разработчик одного системного элемента обвиняет разработчика другого элемента в причине возникновения дефекта. Для защиты от подобного обвинения разработчик программного элемента должен:

1) предусмотреть средства обработки ошибки, которые тестируют все вводы информации от других элементов системы;
2) провести тесты, моделирующие неудачные данные или другие потенциальные ошибки интерфейса ПС;
3) записать результаты тестов, чтобы использовать их как доказательство невиновности в случае «указания причины»;
4) принять участие в планировании и проектировании системных тестов, чтобы гарантировать адекватное тестирование ПС.

В конечном счете системные тесты должны проверять, что все системные элементы правильно объединены и выполняют назначенные функции. Рассмотрим основные типы системных тестов [13], [52].

Тестирование восстановления

Многие компьютерные системы должны восстанавливаться после отказов и возобновлять обработку в пределах заданного времени. В некоторых случаях система должна быть отказоустойчивой, то есть отказы обработки не должны быть причиной прекращения работы системы. В других случаях системный отказ должен быть устранен в пределах заданного кванта времени, иначе заказчику наносится серьезный экономический ущерб.

Тестирование восстановления использует самые разные пути для того, чтобы заставить ПС отказать, и проверяет полноту выполненного восстановления. При автоматическом восстановлении оцениваются правильность повторной инициализации, механизмы копирования контрольных точек, восстановление данных, перезапуск. При ручном восстановлении оценивается, находится ли среднее время восстановления в допустимых пределах.

Тестирование безопасности

Компьютерные системы очень часто являются мишенью незаконного проникновения. Под проникновением понимается широкий диапазон действий: попытки хакеров проникнуть в систему из спортивного интереса, месть рассерженных служащих, взлом мошенниками для незаконной наживы.

Тестирование безопасности проверяет фактическую реакцию защитных механизмов, встроенных в систему, на проникновение.

В ходе тестирования безопасности испытатель играет роль взломщика. Ему разрешено все:

- попытки узнать пароль с помощью внешних средств;
- атака системы с помощью специальных утилит, анализирующих защиты;
- подавление, ошеломление системы (в надежде, что она откажется обслуживать других клиентов);
- целенаправленное введение ошибок в надежде проникнуть в систему в ходе восстановления;

- просмотр несекретных данных в надежде найти ключ для входа в систему.

Конечно, при неограниченном времени и ресурсах хорошее тестирование безопасности взломает любую систему. Задача проектировщика системы — сделать цену проникновения более высокой, чем цена получаемой в результате информации.

Стрессовое тестирование

На предыдущих шагах тестирования способы «белого» и «черного ящиков» обеспечивали полную оценку нормальных программных функций и качества функционирования. Стрессовые тесты проектируются для навязывания программам ненормальных ситуаций. В сущности, проектировщик стрессового теста спрашивает, как сильно можно расшатать систему, прежде чем она откажет?

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

Примеры:

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

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

Тестирование производительности

В системах реального времени и встроенных системах недопустимо ПО, которое реализует требуемые функции, но не соответствует требованиям производительности.

Тестирование производительности проверяет скорость работы ПО в компьютерной системе. Производительность тестируется на всех шагах процесса тестирования. Даже на уровне элемента при проведении тестов «белого ящика» может оцениваться производительность индивидуального модуля. Тем не менее, пока все системные элементы не объединятся полностью, не может быть установлена истинная производительность системы. Иногда тестирование производительности сочетают со стрессовым тестированием. При этом нередко требуется специальный аппаратный и программный инструментарий. Например, часто требуется точное измерение используемого ресурса (процессорного цикла и т. д.). Внешний инструментарий регулярно отслеживает интервалы выполнения, регистрирует события (например, прерывания) и машинные состояния. С помощью инструментария испытатель может обнаружить состояния, которые приводят к деградации и возможным отказам системы.

Искусство отладки

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

Итак, процессу отладки предшествует выполнение тестового варианта. Его результаты оцениваются, регистрируется несоответствие между ожидаемым и реальным результатами. Несоответствие является симптомом скрытой причины. Процесс отладки пытается сопоставить симптом с причиной, вследствие чего приводит к исправлению ошибки. Возможны два исхода процесса отладки:

1) причина найдена, исправлена, уничтожена;
2) причина не найдена.

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

Возможные разные способы проявления ошибок:

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

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

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

В ходе отладки мы встречаем ошибки в широком диапазоне: от мелких неприятностей до катастроф. Следствием увеличения ошибок является усиление давления на отладчика — «найди ошибки быстрее!!!». Часто из-за этого давления разработчик устраняет одну ошибку и вносит две новые ошибки.

Английский термин *debugging* (отладка) дословно переводится как «ловля блох», который отражает специфику процесса — погоню за объектами отладки, «блохами». Рассмотрим, как может быть организован этот процесс «ловли блох» [3], [64].

Различают две группы методов отладки:

- аналитические;
- экспериментальные.

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

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

В простейшем случае место проявления симптома и ошибочный фрагмент совпадают. Но чаще всего они далеко отстоят друг от друга.

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

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

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

В экспериментальных методах для прослеживания выполняется:

1. Выдача значений переменных в указанных точках.
2. Трассировка переменных (выдача их значений при каждом изменении).
3. Трассировка потоков управления (имен вызываемых процедур, меток, на которые передается управление, номеров операторов перехода).

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

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