1. Понятие качества программного обеспечения.

Стандарт ГОСТ 2844-94 дает определение **качества программного обеспечения** как совокупность свойств (показателей качества) ПО, которые обеспечивают его способность удовлетворять потребности заказчика в соответствии с назначением.

Стандарт ISO/IEC 25000:2014 даёт следующее определение понятия **качество программного обеспечения** – способность программного продукта при заданных условиях удовлетворять установленным или предполагаемым потребностям.

Согласно стандарту на этапах ЖЦ должен проводиться контроль качества ПО:

- проверка соответствия требований проектируемому продукту и критериев их достижения;
- верификация и аттестация (валидация) промежуточных результатов ПО на этапах ЖЦ и измерение степени удовлетворения достигаемых отдельных показателей;
- тестирование готовой ПС, сбор данных об отказах, дефектах и других ошибках, обнаруженных в системе;
- подбор моделей надежности для оценивания надежности по полученным результатам тестирования (дефекты, отказы и др.);
- оценка показателей качества, заданных в требованиях на разработку ПС.

2. Показатели качества программного обеспечения.

Модель качества ПО содержит 4 уровня:

1. **Первый уровень** содержит 6 х-р:

- функциональность
- надежность
- удобство
- эффективность
- сопровождаемость

- переносимость

2. **Второй уровень** - атрибуты для каждой х-р качества, которые детализируют разные аспекты конкретной х-р

3. **Третий уровень** - предназначен для измерения качества с помощью метрик, каждая из них согласно стандарту [10.1] определяется как комбинация метода измерения атрибута и шкалы измерения значений атрибутов.

4. **Четвертый уровень** - это оценочный элемент метрики, который используется для оценки количественного или качественного значения отдельного атрибута показателя ПО

Функциональность. Группа свойств ПО, обуславливающая его способность выполнять определенный перечень функций, которые удовлетворяют потребностям в соответствии с назначением.

К атрибутам функциональности относятся:

- функциональная полнота - свойство компонента, которое показывает степень достаточности основных функций для решения задач в соответствии с назначением ПО;
- **правильность** (точность) - атрибут, который показывает степень достижения правильных результатов;
- **интероперабельность** - атрибут, который показывает возможность взаимодействовать на ПО специальными системами и средами (ОС, сеть);
- **защищенность** - атрибут, который показывает на способность ПО предотвращать несанкционированный доступ (случайный или умышленный) к программам и данным.

Надёжность. Группа свойств, обусловливающая способность ПО сохранять работоспособность и преобразовывать исходные данные в результат за установленный период времени, характер отказов которого является следствием внутренних дефектов и условий его применения.

К атрибутам надежности ПО относятся:

- **безотказность** - атрибут, который определяет способность ПО функционировать без отказов (как программы, так и оборудования);

• **устойчивость к ошибкам** - атрибут, который показывает на способность ПО выполнять функции при аномальных условиях (сбой аппаратуры, ошибки в данных и интерфейсах, нарушение в действиях оператора и др.);

• **восстанавливаемость** - атрибут, который показывает на способность программы к перезапуску для повторного выполнения и восстановления данных после отказов.

Дополнительные атрибуты надёжности ПО:

• готовность к использованию (availability);

• готовностью к непрерывному функционированию (reliability);

• безопасность для окружающей среды, т.е. способность системы не вызывать катастрофических последствий в случае отказа (safety);

• секретность и сохранность информации (confidential);

• способность к сохранению системы и устойчивости к самопроизвольному ее изменению (integrity);

• способность к эксплуатации ПО, простота выполнения операций обслуживания, а также устранения ошибок, восстановление системы после их устранения и т.п. (maintainability);

• готовность и сохранность информации (security) и др.

Удобство применения. Совокупность свойств ПО для предполагаемого круга пользователей и отражающих легкость его освоения и адаптации к изменяющимся условиям эксплуатации, стабильность работы и подготовки данных, понимаемость результатов, удобства внесения изменений в программную документацию и в программы.

К атрибутам удобства применения относятся:

• **понимаемость** - атрибут, который определяет усилия, затрачиваемые на распознавание логических концепций и условий применения ПО;

• **изучаемость** (легкость изучения) - атрибут, который определяет усилия пользователей на определение применимости ПО путем использования операционного контроля, диагностики, а также процедур, правил и документации;

- **оперативность** - атрибут, который показывает на реакцию системы при выполнении операций и операционного контроля;
- **согласованность** - атрибут, который показывает соответствие разработки требованиям стандартов, соглашений, правил, законов и предписаний.

Эффективность - множество атрибутов, которые определяют взаимосвязь уровней выполнения ПО, использования ресурсов (средства, аппаратура, материалы - бумага для печатающего устройства и др.) и услуг, выполняемых штатным обслуживающим персоналом и др.

К атрибутам эффективности ПО относятся:

- **реактивность** - атрибут, который показывает время отклика, обработки и выполнения функций;
- **эффективность ресурсов** - атрибут, показывающий количество и продолжительность используемых ресурсов при выполнении функций ПО;
- **согласованность** - атрибут, который показывает соответствие данного атрибута с заданными стандартами, правилами и предписаниями.

Сопровождаемость – множество свойств, которые показывают на усилия, которые надо затратить на проведение модификаций, включающих корректировку, усовершенствование и адаптацию ПО при изменении среды, требований или функциональных спецификаций.

Сопровождаемость включает атрибуты:

- **анализируемость** - атрибут, определяющий необходимые усилия для диагностики отказов или идентификации частей, которые будут модифицироваться;
- **изменяемость** - атрибут, который определяет удаление ошибок в ПО или внесение изменений для их устранения, а также введение новых возможностей в ПО или в среду функционирования;
- **стабильность** - атрибут, указывающий на постоянство структуры и риск ее модификации;
- **тестируемость** - атрибут, показывающий на усилия при проведении валидации и верификации с целью обнаружения несоответствий

требованиям, а также на необходимость проведения модификации ПО и сертификации;

- **согласованность** - атрибут, который показывает соответствие данного атрибута соглашениям, правилам и предписаниям стандарта.

Переносимость - множество показателей, указывающих на способность ПО адаптироваться к работе в новых условиях среды выполнения. Среда может быть организационной, аппаратной и программной. Атрибуты переносимости:

- **адаптивность** - атрибут, определяющий усилия, затрачиваемые на адаптацию к различным средам;
- **настраиваемость** (простота инсталляции) - атрибут, который определяет необходимые усилия для запуска данного ПО в специальной среде;
- **сосуществование** - атрибут, который определяет возможность использования специального ПО в среде действующей системы;
- **заменяемость** - атрибут, который обеспечивают возможность интероперабельности при совместной работе с другими программами с необходимой инсталляцией или адаптацией ПО;
- **согласованность** - атрибут, который показывает на соответствие стандартам или соглашениями по обеспечению переноса ПО.

Совместимость: пригодность продукции, процессов или услуг к совместному, но не вызывающему нежелательных взаимодействий использованию при заданных условиях для выполнения установленных требований.

3. Метрики качества программного обеспечения.

Для набора характеристик качества ПО, приведенных в требованиях, определяются соответствующие метрики, модели их оценки и диапазон значений мер для измерения отдельных атрибутов качества.

Существует три типа метрик:

- **метрики программного продукта**, которые используются при измерении его характеристик - свойств;

• **метрики процесса**, которые используются при измерении свойства процесса ЖЦ создания продукта;

• **метрики использования**.

Типы метрик:

1. Метрики программного продукта:
   1. Внутренние
      - метрики размера
      - метрики сложности
      - метрики стиля
   2. Внешние
      - метрики надежности
      - метрики функциональности
      - метрики сопровождения
      - метрики стоимости
2. Метрики процесса
   - общее время разработки
   - время модификации моделей
   - время выполнения работ на процессе
   - число найденных ошибок
   - стоимость проверки качества
   - стоимость процесса разработки
3. Метрики использования
   - точность и полнота реализации задач пользователя
   - трудозатраты

Метрики Холстеда.

В основе вычисления метрик Холстеда лежит концепция, согласно которой алгоритм состоит только из операторов и операндов (проверяется рассмотрением простых вычислительных машин с форматом команд, содержащим две части: код операции и адрес операнда). Операнды - переменные или константы, используемые в данной реализации алгоритма.

Операторы - комбинации символов, влияющие на значение или порядок операндов.

4. Управление качеством программного обеспечения.

Под **управлением качеством** понимается совокупность организационной структуры и ответственных лиц, а также процедур, процессов и ресурсов для планирования и управления достижением качества ПС.

Управление качеством - SQM (Software Quality Management) базируется на применении стандартных положений по гарантии качества - SQA (Software Quality Assurance).

Цель процесса SQA состоит в гарантировании того, что продукты и процессы согласуются с требованиями, соответствуют планам и включают следующие виды деятельности:

- внедрение стандартов и соответствующих процедур разработки ПС на этапах ЖЦ;
- оценка соблюдения положений этих стандартов и процедур. Гарантия качества состоит в следующем:
- проверка непротиворечивости и выполнимости планов;
- согласование промежуточных рабочих продуктов с плановыми показателями;
- проверка изготовленных продуктов заданным требованиям;
- анализ применяемых процессов на соответствие договору и планам;
- согласование с заказчиком среды и методов разработки продукта;
- проверка принятых метрик продуктов, процессов и приемов их измерения в соответствии с утвержденным стандартом и процедурами измерения.

Цель процесса управления SQM - мониторинг (систематический контроль) качества для гарантии того, что продукт будет удовлетворять потребителю и предполагает выполнение следующего:

- определение количественных свойств качества, основанных на выявленных и предусмотренных потребностях пользователей;

• управление реализацией поставленных целей для достижения качества.

Процесс SQM основывается на гарантии того, что:

• цели достижения требуемого качества установлены для всех рабочих продуктов в контрольных точках продукта;

• определена стратегия достижения качества, метрики, критерии, приемы, требования к процессу измерения и др.;

• определены и выполняются действия, связанные с предоставлением продуктам свойств качества;

• проводится контроль качества (SQA, верификация и валидация) и целей;

• выполняются процессы измерения и оценивания конечного продукта на достижение требуемого качества.

5. Методы обеспечения критериев качества ПО.

Основные стандартные положения по созданию качественного продукта и оценки достигнутого его уровня позволяют выделить два процесса обеспечения качества на этапах ЖЦ:

• гарантия (подтверждение) качества ПС как результат определенной деятельности на каждом этапе ЖЦ с проверкой соответствия системы стандартам и процедурам, ориентированным на достижении качества;

• инженерия качества как процесс предоставления продуктам ПО свойств функциональности, надежности, сопровождения и других характеристик качества.

Процессы достижения качества предназначены для:

• управления, разработки и обеспечения гарантий в соответствии с указанными стандартами и процедурами;

• управления конфигурацией (идентификация, учет состояния и действий по аутентификации), риском и проектом в соответствии со стандартами и процедурами;

• контроль базовой версии ПС и реализованных в ней характеристик качества.

Выполнение указанных процессов включает такие действия:

- оценка стандартов и процедур, которые выполняются при разработке программ;
- ревизия управления, разработки и обеспечение гарантии качества ПО, а также проектной документации (отчеты, графики разработки, сообщения и др.);
- контроль проведения формальных инспекций и просмотров;
- анализ и контроль проведения приемочного тестирования (испытания) ПС.

Инженерия качества включает набор методов и мероприятий, с помощью которых программные продукты проверяются на выполнение требований к качеству и снабжаются характеристиками, предусмотренными в требованиях на ПО.

Методы обеспечения критериев качества программного обеспечения:

- Обеспечение завершенности программного средства
- Обеспечение точности программного средства
- Обеспечение автономности программного средства
- Обеспечение устойчивости программного средства
- Обеспечение защищенности программных средств

6. Понятие надёжности программного обеспечения.

Надежность программного обеспечения - способность программного продукта безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью.

Существует 4 основные составляющие функциональной надежности программных систем:

- **безотказность** - свойство программы выполнять свои функции во время эксплуатации;
- **работоспособность** - свойство программы корректно (так как ожидает пользователь) работать весь заданный период эксплуатации;
- **безопасность** - свойство программы быть не опасной для людей и окружающих систем;

• **защищенность** - свойство программы противостоять случайным или умышленным вторжениям в нее.

7. Основные показатели надёжности программного обеспечения.

1. **Вероятность безотказной работы P(tз)** – это вероятность того, что в пределах заданной наработки отказ системы не возникает.

2. **Вероятность отказа** – вероятность того, что в пределах заданной наработки отказ системы возникает. Это показатель, обратный предыдущему.

$$Q(t\,з) = 1 - P(tз)$$

где tз – заданная наработка, ч.;

Q(tз) – вероятность отказа.

3. **Интенсивность отказов системы** – это условная плотность вероятности возникновения отказа ПИ в определенный момент времени при условии, что до этого времени отказ не возник.

$$\chi(t) = \frac{f(t)}{P(t)}$$

где f(t) – плотность вероятности отказа в момент времени t.

$$f(t) = \frac{d}{dt}Q(t) = \frac{d}{dt}(1 - P(t)) = \frac{d}{dt}P(t)$$

Существует следующая связь между интенсивностью отказов системы и вероятностью безотказной работы

$$P(t) = \exp\left(-\int_0^t \chi(t)dt\right)$$

4. **Средняя наработка на отказ Ti** - математическое ожидание времени работы ПИ до очередного отказа:

$$T_i = \int_0^t t \cdot f(t)dt$$

Иначе среднюю наработку на отказ Ti можно представить:

$$T_i = \frac{t_1 + t_2 + t_3 \ldots t_n}{n} = \left(\frac{i}{n}\right) \cdot \sum_{i=1}^{n} t_i$$

где t – время работы ПИ между отказами, с.

n – количество отказов.

5. **Среднее время восстановления T** - математическое ожидание времени восстановления - t; времени, затраченного на обнаружение и локализацию отказа - t ; времени устранения отказа - t ; времени пропускной проверки работоспособности - t : t = t + t + t

6. **Коэффициент готовности K** - вероятность того, что ПИ ожидается в работоспособном состоянии в произвольный момент времени его использования по назначению:

K = T1 / (T1 + T2)

Где T1 – средняя наработка на отказ

T2 – среднее время восстановления

8. Модели надёжности программного обеспечения.

1. **Аналитические модели** дают возможность рассчитывать количественные показатели надежности, основываясь на данных о поведении программы в процессе тестирования (измеряющие и оценивающие модели).

2. **Эмпирические модели** базируются на анализе структурных особенностей программ. Они рассматривают зависимость показателей

надёжности от числа межмодульных связей, количества циклов в модулях и т.д.

Модель Шумана

Модель La Padula

Модель Миллса

Модель Липова

9. Основные понятия и принципы тестирования программного обеспечения.

Тестирование — процесс выполнения программы с целью обнаружения ошибок. Шаги процесса задаются тестами.

Каждый тест определяет:

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

Другое название теста — тестовый вариант. Полную проверку программы гарантирует *исчерпывающее тестирование.* Оно требует проверить все наборы исходных данных, все варианты их обработки и включает большое количество тестовых вариантов. Увы, но исчерпывающее тестирование во многих случаях остается только мечтой — срабатывают ресурсные ограничения (прежде всего, ограничения по времени).

Хорошим считают тестовый вариант с высокой вероятностью обнаружения еще не раскрытой ошибки. Успешным называют тест, который обнаруживает до сих пор не раскрытую ошибку.

Целью проектирования тестовых вариантов является систематическое обнаружение различных классов ошибок при минимальных затратах времени и стоимости.

Тестирование обеспечивает:

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

Рассмотрим информационные потоки процесса тестирования. Они показаны на рис. 6.1.

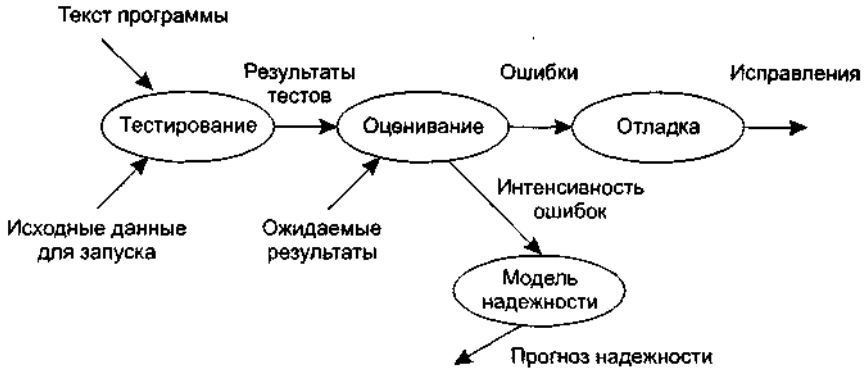


Рис. 6.1. Информационные потоки процесса тестирования

На входе процесса тестирования три потока:

- текст программы;
- исходные данные для запуска программы;
- ожидаемые результаты.

Выполняются тесты, все полученные результаты оцениваются. Это значит, что реальные результаты тестов сравниваются с ожидаемыми результатами. Когда обнаруживается несовпадение, фиксируется ошибка — начинается отладка. Процесс отладки непредсказуем по времени. На поиск места дефекта и исправление может потребоваться час, день, месяц. Неопределенность в отладке приводит к большим трудностям в планировании действий.

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

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

В конечном счете, если тесты не обнаруживают ошибок, появляется сомнение в том, что тестовые варианты достаточно продуманы и что в ПО нет

скрытых ошибок. Такие ошибки будут, в конечном итоге, обнаруживаться пользователями и корректироваться разработчиком на этапе сопровождения (когда стоимость исправления возрастает в 60-100 раз по сравнению с этапом разработки).

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

10. Понятие структурного и функционального тестирования. Тестирование «черного ящика»

Известны: функции программы.

Исследуется: работа каждой функции на всей области определения.

Как показано на рис. 6.2, основное место приложения тестов «черного ящика» — интерфейс ПО.

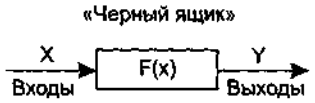


Рис. 6.2. Тестирование «черного ящика»

Эти тесты демонстрируют:

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

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

Тестирование «белого ящика»

Известна: внутренняя структура программы.

Исследуются: внутренние элементы программы и связи между ними (рис. 6.3).

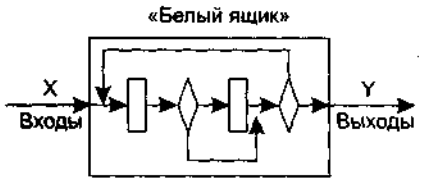


Рис. 6.3. Тестирование «белого ящика»

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

В основе структурного тестирования лежит концепция максимально полного тестирования всех маршрутов, предусмотренных алгоритмом (последовательности операторов программы, выполняемых при конкретном варианте исходных данных). Недостатки: построенные тестовые наборы не обнаруживают пропущенных маршрутов и ошибок, зависящих от заложенных данных; не дают гарантии, что программа правильна.

Другим способом проверки программ является функциональное тестирование: программа рассматривается как «черный ящик», целью тестирования является выяснение обстоятельств, когда поведение программы не соответствует спецификации.

11. Способ тестирования базового пути.

Тестирование базового пути — это способ, который основан на принципе «белого ящика». Автор этого способа — Том МакКейб (1976) [49].

Способ тестирования базового пути дает возможность:

- получить оценку комплексной сложности программы;

- использовать эту оценку для определения необходимого количества тестовых вариантов.

Тестовые варианты разрабатываются для проверки базового множества путей (маршрутов) в программе. Они гарантируют однократное выполнение каждого оператора программы при тестировании.

Для представления программы используется потоковый граф. Перечислим его особенности.

1. Граф строится отображением управляющей структуры программы. В ходе отображения закрывающие скобки условных операторов и операторов циклов (end if; end loop) рассматриваются как отдельные (фиктивные) операторы.

2. Узлы (вершины) потокового графа соответствуют линейным участкам программы, включают один или несколько операторов программы.

3. Дуги потокового графа отображают поток управления в программе (передачи управления между операторами). Дуга — это ориентированное ребро.

4. Различают операторные и предикатные узлы. Из операторного узла выходит одна дуга, а из предикатного — две дуги.

5. Предикатные узлы соответствуют простым условиям в программе. Составное условие программы отображается в несколько предикатных узлов. Составным называют условие, в котором используется одна или несколько булевых операций (OR, AND).

6. Замкнутые области, образованные дугами и узлами, называют регионами.

Цикломатическая сложность — метрика ПО, которая обеспечивает количественную оценку логической сложности программы. В способе тестирования базового пути Цикломатическая сложность определяет:

- количество независимых путей в базовом множестве программы;
- верхнюю оценку количества тестов, которое гарантирует однократное выполнение всех операторов.

Независимым называется любой путь, который вводит новый оператор обработки или новое условие. В терминах потокового графа независимый путь должен содержать дугу, не входящую в ранее определенные пути.

Свойства базового множества:

1) тесты, обеспечивающие его проверку, гарантируют:

- однократное выполнение каждого оператора;
- выполнение каждого условия по True-ветви и по False-ветви;

2) мощность базового множества равна цикломатической сложности потокового графа.

Значение 2-го свойства трудно переоценить — оно дает априорную оценку количества независимых путей, которое имеет смысл искать в графе.

Цикломатическая сложность вычисляется одним из трех способов:

1) цикломатическая сложность равна количеству регионов потокового графа;

2) цикломатическая сложность определяется по формуле

V(G)-E-N+2,

где *E* — количество дуг, *N* — количество узлов потокового графа;

12. Способы тестирования условий.

Цель этого семейства способов тестирования — строить тестовые варианты для проверки логических условий программы. При этом желательно обеспечить охват операторов из всех ветвей программы.

Простое условие — булева переменная или выражение отношения.

Выражение отношения имеет вид

E1 <оператор отношения> E2,

где El, E2 — арифметические выражения, а в качестве оператора отношения используется один из следующих операторов: <, >, =, $\neq$, $\leq$, $\geq$.

Составное условие состоит из нескольких простых условий, булевых операторов и круглых скобок. Будем применять булевы операторы OR, AND (&), NOT. Условия, не содержащие выражений отношения, называют булевыми выражениями.

Таким образом, элементами условия являются: булев оператор, булева переменная, пара скобок (заключающая простое или составное условие), оператор отношения, арифметическое выражение. Эти элементы определяют типы ошибок в условиях.

Если условие некорректно, то некорректен по меньшей мере один из элементов условия. Следовательно, в условии возможны следующие типы ошибок:

- ошибка булева оператора (наличие некорректных / отсутствующих / избыточных булевых операторов);
- ошибка булевой переменной;
- ошибка булевой скобки;
- ошибка оператора отношения;
- ошибка арифметического выражения.

Способ тестирования условий ориентирован на тестирование каждого условия в программе. Методики тестирования условий имеют два достоинства. Во-первых, достаточно просто выполнить измерение тестового покрытия условия. Во-вторых, тестовое покрытие условий в программе — это фундамент для генерации дополнительных тестов программы.

Целью тестирования условий является определение не только ошибок в условиях, но и других ошибок в программах. Если набор тестов для программы А эффективен для обнаружения ошибок в условиях, содержащихся в А, то вероятно, что этот набор также эффективен для обнаружения других ошибок в А. Кроме того, если методика тестирования эффективна для обнаружения ошибок в условии, то вероятно, что эта методика будет эффективна для обнаружения ошибок в программе.

Существует несколько методик тестирования условий.

Простейшая методика — *тестирование ветвей.* Здесь для составного условия *С* проверяется:

- каждое простое условие (входящее в него);
- True-ветвь;
- False-ветвь.

Другая методика — *тестирование области определения.* В ней для выражения отношения требуется генерация 3-4 тестов. Выражение вида

E1 <оператор отношения> E2

проверяется тремя тестами, которые формируют значение E1 большим, чем E2, равным E2 и меньшим, чем E2.

Если оператор отношения неправилен, а E1 и E2 корректны, то эти три теста гарантируют обнаружение ошибки оператора отношения.

Для определения ошибок в E1 и E2 тест должен сформировать значение E1 большим или меньшим, чем E2, причем обеспечить как можно меньшую разницу между этими значениями.

Для булевых выражений с n переменными требуется набор из 2^n тестов. Этот набор позволяет обнаружить ошибки булевых операторов, переменных и скобок, но практичен только при малом n. Впрочем, если в булево выражение каждая булева переменная входит только один раз, то количество тестов легко уменьшается.

13. Способ тестирования потоков данных.

В предыдущих способах тесты строились на основе анализа управляющей структуры программы. В данном способе анализу подвергается информационная структура программы.

Работу любой программы можно рассматривать как обработку потока данных, передаваемых от входа в программу к ее выходу.

Пусть потоковый граф программы имеет вид, представленный на рис. 6.8. В нем сплошные дуги — это связи по управлению между операторами в программе. Пунктирные дуги отмечают информационные связи (связи по потокам данных). Обозначенные здесь информационные связи соответствуют следующим допущениям:

- в вершине 1 определяются значения переменных a, b;
- значение переменной *a* используется в вершине 4;
- значение переменной *b* используется в вершинах 3, 6;
- в вершине 4 определяется значение переменной c, которая используется в вершине 6.

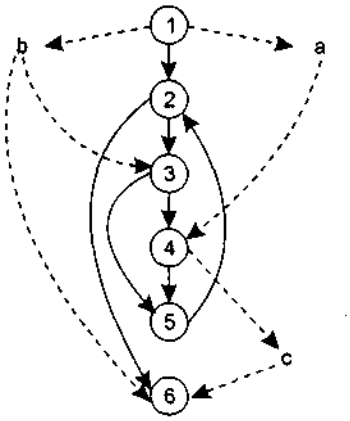


Рис. 6.8. Граф программы с управляющими и информационными связями

В общем случае для каждой вершины графа можно записать:

- множество определений данных

DEF(i) = { x | i -я вершина содержит определение x};

- множество использований данных:

USE (i) = { x | i -я вершина использует x}.

Под *определением данных* понимают действия, изменяющие элемент данных. Признак определения — имя элемента стоит в левой части оператора присваивания:

x:=*f*(…).

Использование данных — это применение элемента в выражении, где происходит обращение к элементу данных, но не изменение элемента. Признак использования — имя элемента стоит в правой части оператора присваивания:

?:=*f*(*x*).

Здесь место подстановки другого имени отмечено прямоугольником (прямоугольник играет роль метки-заполнителя).

Назовём *DU-цепочкой (цепочкой определения-использования)* конструкцию *[x, i,j],* где *i,j* — имена вершин; x определена в *i*-й вершине *(x* ∈ DEF(*i*)) и используется в *j* -й вершине *(x* ∈ USE(*j*)).

В нашем примере существуют следующие DU-цепочки:

[a,1,4],[b, 1,3], [b, 1,6], [c, 4, 6].

Способ *DU-тестирования* требует охвата всех DU-цепочек программы. Таким образом, разработка тестов здесь проводится на основе анализа жизни всех данных программы.

Очевидно, что для подготовки тестов требуется выделение маршрутов — путей выполнения программы на управляющем графе. Критерий для выбора пути — покрытие максимального количества DU-цепочек.

Шаги способа DU-тестирования:

1) построение управляющего графа (УГ) программы;

2) построение информационного графа (ИГ);

3) формирование полного набора DU-цепочек;

4) формирование полного набора отрезков путей в управляющем графе (отображением набора DU-цепочек информационного графа, рис. 6.9);

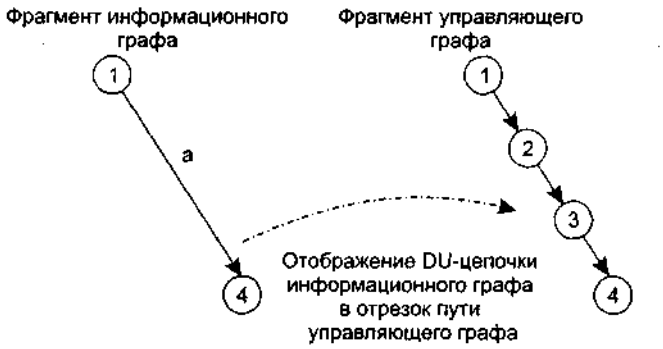


Рис. 6.9. Отображение DU-цепочки *в* отрезок пути

5) построение маршрутов — полных путей на управляющем графе, покрывающих набор отрезков путей управляющего графа;

6) подготовка тестовых вариантов.

Достоинства DU-тестирования:

- простота необходимого анализа операционно-управляющей структуры программы;
- простота автоматизации.

Недостаток DU-тестирования: трудности в выборе минимального количества максимально эффективных тестов.

Область использования DU-тестирования: программы с вложенными условными операторами и операторами цикла.

14. Тестирование циклов.

Цикл — наиболее распространенная конструкция алгоритмов, реализуемых в ПО. Тестирование циклов производится по принципу «белого ящика», при проверке циклов основное внимание обращается на правильность конструкций циклов.

Различают 4 типа циклов: простые, вложенные, объединенные, неструктурированные. Структура циклов приведена на рис. 6.10.

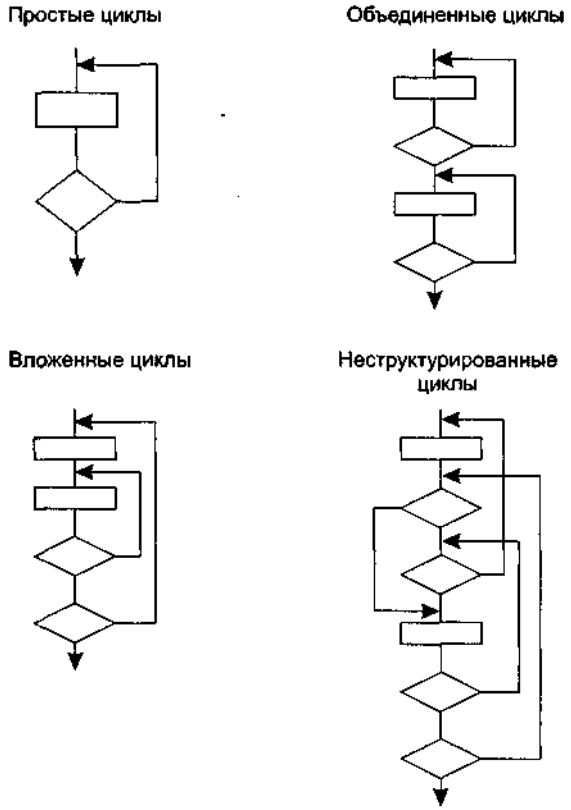


Рис. 6.10. Типовые структуры циклов

Простые циклы

Для проверки простых циклов с количеством повторений n может использоваться один из следующих наборов тестов:

1) прогон всего цикла;

2) только один проход цикла;

3) два прохода цикла;

4) m проходов цикла, где $m<n$;

5) *n* - 1, *n, n* + 1 проходов цикла.

Вложенные циклы

С увеличением уровня вложенности циклов количество возможных путей резко возрастает. Это приводит к нереализуемому количеству тестов [13]. Для сокращения количества тестов применяется специальная методика, в которой используются такие понятия, как объемлющий и вложенный циклы (рис. 6.11).

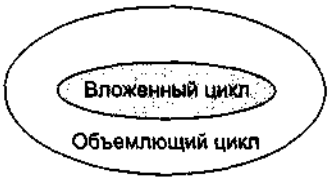


Рис. 6.11. Объемлющий и вложенный циклы

Порядок тестирования вложенных циклов иллюстрирует рис. 6.12.

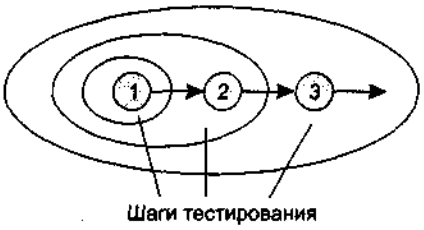


Рис. 6.12. Шаги тестирования вложенных циклов

Шаги тестирования.

1. Выбирается самый внутренний цикл. Устанавливаются минимальные значения параметров всех остальных циклов.

2. Для внутреннего цикла проводятся тесты простого цикла. Добавляются тесты для исключенных значений и значений, выходящих за пределы рабочего диапазона.

3. Переходят в следующий по порядку объемлющий цикл. Выполняют его тестирование. При этом сохраняются минимальные значения параметров для всех внешних (объемлющих) циклов и типовые значения для всех вложенных циклов.

4. Работа продолжается до тех пор, пока не будут протестированы все циклы.

Объединенные циклы

Если каждый из циклов независим от других, то используется техника тестирования простых циклов. При наличии зависимости (например, конечное значение счетчика первого цикла используется как начальное значение счетчика второго цикла) используется методика для вложенных циклов.

Неструктурированные циклы

Неструктурированные циклы тестированию не подлежат. Этот тип циклов должен быть переделан с помощью структурированных программных конструкций

15. Способ тестирования по методу разбиения по эквивалентности.

Разбиение по эквивалентности — самый популярный способ тестирования «черного ящика» [3], [14].

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

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

На рис. 7.2 каждый класс эквивалентности показан эллипсом. Здесь выделены входные классы эквивалентности допустимых и недопустимых исходных данных, а также классы результатов.

Классы эквивалентности могут быть определены по спецификации на программу.

Рис. 7.2. Разбиение по эквивалентности

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

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

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

Условие ввода может задавать:

1) определенное значение;

2) диапазон значений;

3) множество конкретных величин;

4) булево условие.

Сформулируем *правила формирования классов эквивалентности.*

1. Если условие ввода задает диапазон *n...m,* то определяются один допустимый и два недопустимых класса эквивалентности:

- V_Class={*n*.. .*m*} — допустимый класс эквивалентности;
- Inv_C1ass1={*x*|для любого *x: x* < *n*} — первый недопустимый класс эквивалентности;
- Inv_C1ass2={*y*|для любого *y: y* > *m*} — второй недопустимый класс эквивалентности.

2. Если условие ввода задает конкретное значение *a,* то определяется один допустимый и два недопустимых класса эквивалентности:

- V_Class={*a*};
- Inv_Class1 ={*x*|для любого *x: x* < *a*};
- Inv_C1ass2={*y*|для любого *y: y* > *a*}.

3. Если условие ввода задает множество значений *{a, b, c},* то определяются один допустимый и один недопустимый класс эквивалентности:

- V_Class={*a, b, c*};
- Inv_C1ass={*x*|для любого *x: (x* ≠ *a)&(x* ≠ *b)&(x* ≠ *c)}.*

4. Если условие ввода задает булево значение, например true, то определяются один допустимый и один недопустимый класс эквивалентности:

- V_Class={true};
- Inv_Class={false}.

После построения классов эквивалентности разрабатываются тестовые варианты. Тестовый вариант выбирается так, чтобы проверить сразу наибольшее количество свойств класса эквивалентности.

16. Способ тестирования по методу анализа граничных значений.

Как правило, большая часть ошибок происходит на границах области ввода, а не в центре. Анализ граничных значений заключается в получении тестовых вариантов, которые анализируют граничные значения [3], [14], [69]. Данный способ тестирования дополняет способ разбиения по эквивалентности.

Основные отличия анализа граничных значений от разбиения по эквивалентности:

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

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

Сформулируем **правила анализа граничных значений.**

1. Если условие ввода задает диапазон *n...m*, то тестовые варианты должны быть построены:

- для значений *n* и *m;*

- для значений чуть левее *n* и чуть правее *m* на числовой оси.

Например, если задан входной диапазон -1,0...+1,0, то создаются тесты для значений - 1,0, +1,0, - 1,001, +1,001.

2. Если условие ввода задает дискретное множество значений, то создаются тестовые варианты:

- для проверки минимального и максимального из значений;
- для значений чуть меньше минимума и чуть больше максимума.

Так, если входной файл может содержать от 1 до 255 записей, то создаются тесты для О, 1, 255, 256 записей.

3. Правила 1 и 2 применяются к условиям области вывода.

Рассмотрим пример, когда в программе требуется выводить таблицу значений. Количество строк и столбцов в таблице меняется. Задается тестовый вариант для минимального вывода (по объему таблицы), а также тестовый вариант для максимального вывода (по объему таблицы).

4. Если внутренние структуры данных программы имеют предписанные границы, то разрабатываются тестовые варианты, проверяющие эти структуры на их границах.

5. Если входные или выходные данные программы являются упорядоченными множествами (например, последовательным файлом, линейным списком, таблицей), то надо тестировать обработку первого и последнего элементов этих множеств.

Большинство разработчиков используют этот способ интуитивно. При применении описанных правил тестирование границ будет более полным, в связи с чем возрастет вероятность обнаружения ошибок.

Рассмотрим применение способов разбиения по эквивалентности и анализа граничных значений на конкретном примере. Положим, что нужно протестировать программу бинарного поиска. Нам известна *спецификация* этой программы. Поиск выполняется в массиве элементов *M,* возвращается индекс *I* элемента массива, значение которого соответствует ключу поиска *Key.*

Предусловия:

1) массив должен быть упорядочен;

2) массив должен иметь не менее одного элемента;

3) нижняя граница массива (индекс) должна быть меньше или равна его верхней границе.

Постусловия:

1) если элемент найден, то флаг Result=True, значение I — номер элемента;

2) если элемент не найден, то флаг Result=False, значение I не определено.

Для формирования классов эквивалентности (и их ребер) надо произвести разбиение области ИД — построить дерево разбиений. Листья дерева разбиений дадут нам искомые классы эквивалентности. Определим стратегию разбиения. На первом уровне будем анализировать выполнимость предусловий, на втором уровне — выполнимость постусловий. На третьем уровне можно анализировать специальные требования, полученные из практики разработчика. В нашем примере мы знаем, что входной массив должен быть упорядочен. Обработка упорядоченных наборов из четного и нечетного количества элементов может выполняться по-разному. Кроме того, принято выделять специальный случай одноэлементного массива. Следовательно, на уровне специальных требований возможны следующие эквивалентные разбиения:

1) массив из одного элемента;

2) массив из четного количества элементов;

3) массив из нечетного количества элементов, большего единицы.

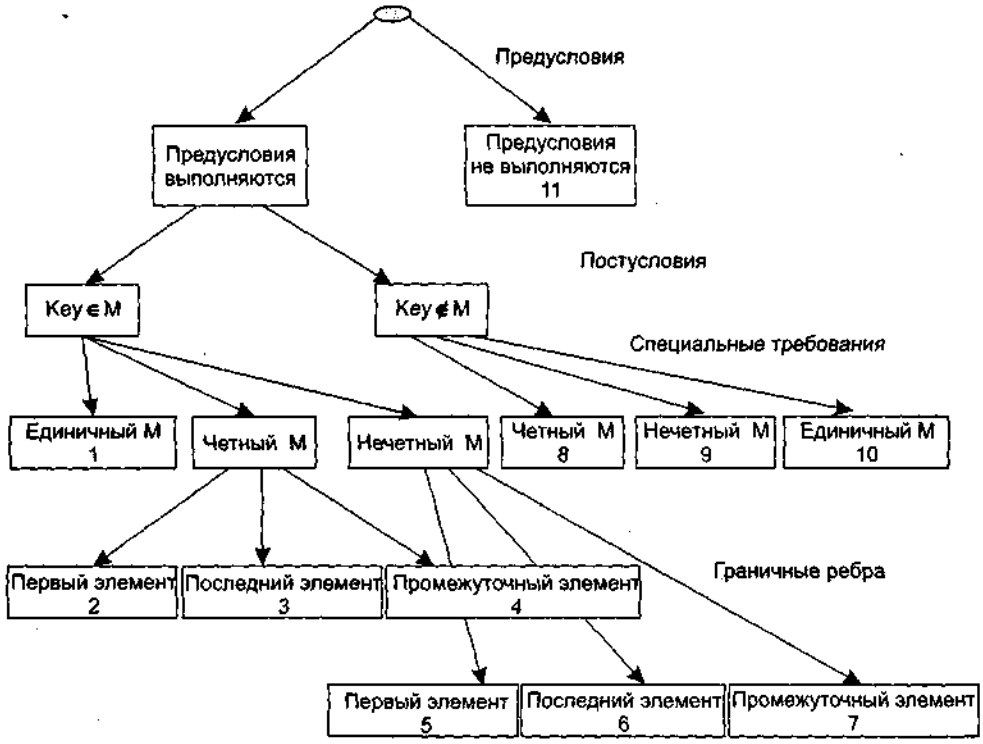
Наконец на последнем, 4-м уровне критерием разбиения может быть анализ ребер классов эквивалентности. Очевидно, возможны следующие варианты:

1) работа с первым элементом массива;

2) работа с последним элементом массива;

3) работа с промежуточным (ни с первым, ни с последним) элементом массива.

Структура дерева разбиений приведена на рис. 7.3.



17. Способ диаграмм причин-следствий.

Способ обеспечивает формальное выведение высокорезультативных тестовых вариантов, основанное на анализе причинно-следственных связей.

Причина – отдельное входное условие или класс эквивалентности.

Следствие – выходное условие или действие системы.

Дополнительный эффект – обнаружение неполноты или неоднозначности спецификаций.

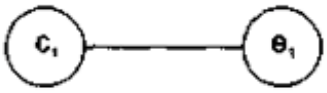
Шаги способа:

1. Разбиение спецификаций на «рабочие» участи и выделение групп причин и следствий.
2. В каждой группе выделяются причины и следствия, им присваиваются идентификаторы.
3. Разрабатывается граф причинно-следственных связей.
4. Граф преобразуется в таблицу решений.
5. Столбцы таблицы решений преобразуются в тестовые варианты.

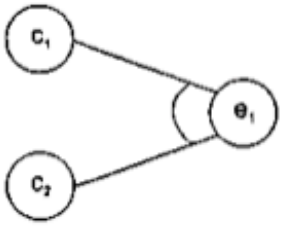
Граф причин следствий

- Обозначения узлов: c_i – причина, e_i – следствие.
- Узел может находится в двух состояниях: 0 и 1.
- Базовые символы:

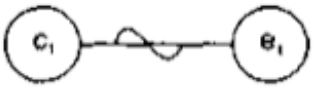
Тождество



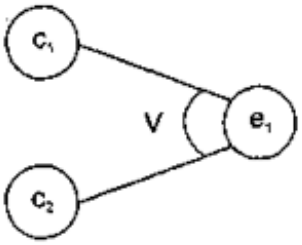
Функция И



Функция НЕ



Функция ИЛИ

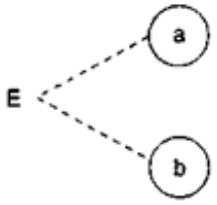


Граф причин следствий

3. Базовые символы:

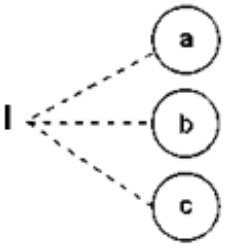
Ограничение E (Exclusive - исключение):

только одна из величин может принимать значение 1



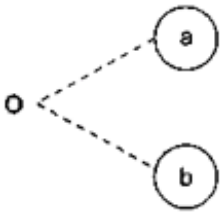
Ограничение I (Inclusive - включение):

по крайней мере одна из величин должна быть равной 1



Ограничение O (Only one - только одно):

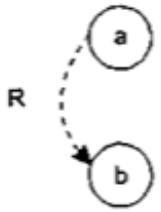
одна и только одна из величин может быть равна 1



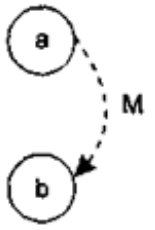
Граф причин следствий

3. Базовые символы:

Ограничение R (Requires - требование):
если a=1, то b должно принимать значение 1



Ограничение M (Masks - скрытие):
если следствие *a*=1, то следствие *b* должно принять значение 0



Генерация таблицы решений.

1. Выбирается некоторое следствие, которое должно быть в состоянии «1».

2. Находятся все комбинации причин (с учетом ограничений), которые устанавливают это следствие в состояние «1». Для этого из следствия прокладывается обратная трасса через граф.

3. Для каждой комбинации причин, приводящих следствие в состояние «1», строится один столбец.

4. Для каждой комбинации причин доопределяются состояния всех других следствий. Они помещаются в тот же столбец таблицы решений.

5. Действия 1-4 повторяются для всех следствий графа.

18. Методика тестирования программного обеспечения.

Общий процесс тестирования объединяет различные способы тестирования в спланированную последовательность шагов, которые приводят к успешному построению программной системы.

19. Тестирование элементов и интеграции.

Тестирование элементов

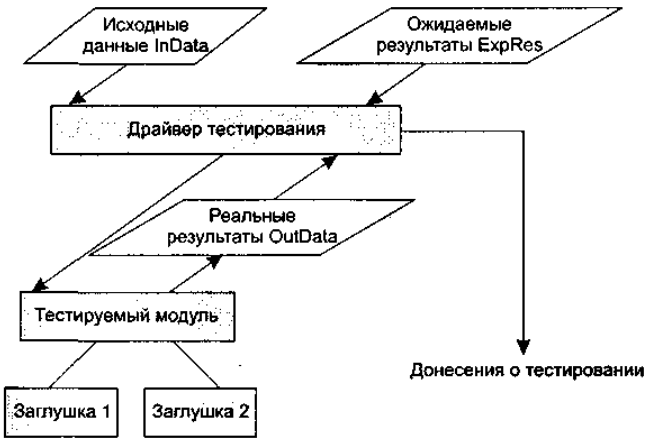
Объект тестирования – наименьшая единица проектирования ПС – модуль.

Цель – индивидуальная проверка каждого модуля.

Основной способ тестирования – структурное.

Последовательно подвергаются тестированию:

- интерфейс модуля (ошибки ввода-вывода);
- внутренние структуры данных (целостность данных);
- независимые пути (ошибочные вычисления, некорректные сравнения, неправильный поток управления);
- пути обработки ошибок (некорректность работы операторов обработки ошибок);
- граничные условия (анализ граничных значений).



Тестирование интеграции

Объект тестирования – сборка программной системы.

Цель – проверка корректности сборки модулей в программную систему.

Основной способ тестирования – функциональное.

Конкретный объект тестирования – межмодульные интерфейсы.

Основные ошибки:

- потеря данных при прохождении интерфейса;
- отсутствие необходимого вызова;
- влияние одного модуля на другой (недопустимое сцепление);
- композиция подфункций не дает нужную функцию;
- некорректная работа с глобальными данными.

Способы тестирования интеграции: нисходящее и восходящее.

1. Нисходящее тестирование.

Модули объединяются движением сверху вниз по управляющей иерархии, начиная от главного управляющего модуля. Выбор модулей осуществляется по принципу обхода в глубину или в ширину (по уровням).

Общая методика тестирования интеграции нисходящим способом:

1. Главный управляющий модуль используется как тестовый драйвер. Все непосредственно подчиненные ему модули временно замещаются заглушками.

2. Одна из заглушек драйвера заменяется реальным модулем.

3. На этом модуле устанавливаются заглушки и проводится набор тестов, проверяющих полученную структуру.

4. Если в модуле-драйвере уже нет заглушек, производится смена модуля-драйвера.

5. Если есть неподключенные модули, выполняется возврат на шаг 2.

Достоинство – ошибки в главной, управляющей части выявляются в первую очередь.

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

Способы устранения недостатков:

1. Откладывать некоторые тесты до замещения заглушек модулями.

2. Разработка более сложных заглушек, частично выполняющие функции модулей.

3. Использовать восходящее тестирование интеграции

2. Восходящее тестирование.

Модули подключаются движением снизу вверх. Подчиненные модули всегда доступны, и нет необходимости в заглушках.

Общая методика тестирования интеграции восходящим способом:

1. Модули нижнего уровня объединяются в кластеры, выполняющие определенную программную подфункцию.

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

3. Тестируется кластер.

4. Драйверы удаляются, а кластеры объединяются в структуру движением вверх, осуществляется тестирование частичной сборки до объединения всех модулей.

Недостаток – система не существует как целое до тех пор, пока не будет добавлен последний модуль.

Достоинство – упрощается разработка тестовых вариантов, отсутствуют заглушки.

3. Комбинированный подход:

- для верхних уровней иерархии применяют нисходящую стратегию;
- для нижних уровней – восходящую стратегию тестирования;
- выделяются критические модули, они должны тестироваться как можно раньше, тестироваться не один раз.

Признаки критического модуля:

- реализует несколько требований к программной системе;
- имеет высокий уровень управления;
- имеет высокую сложность или склонность к ошибкам (как индикатор может использоваться цикломатическая сложность – больше 10).

20. Тестирование правильности и системное тестирование.

Тестирование правильности

Объект тестирования – единая программная система.

Цель тестирования – проверка и подтверждение реализации в ПС функциональных требований заказчика.

Способ тестирования – функциональное.

Тестирование правильности включает также тестирование конфигурации ПС, т.е. совокупности всех элементов информации, полученных в процессе разработки ПС. Базовые элементы:

- план программного проекта;
- спецификация требований к ПС;
- листинги исходных текстов программ;
- план и методика тестирования; тестовые варианты и полученные результаты;
- руководства по работе и инсталляции;
- описание базы данных;
- руководство пользователя;
- и др.

В ходе тестирования правильности реализуются альфа- и бета-тестирование.

Системное тестирование

Объект тестирования – система в целом.

Цель тестирования – проверка правильности объединения и взаимодействия всех элементов компьютерной системы, реализации всех системных функций.

Способ тестирования – Функциональное.

Системное тестирование включает в себя:

1. Тестирование восстановления (время восстановления, правильность повторной инициализации, восстановление данных и др.)
2. Тестирование безопасности (проверка функционирования защитных механизмов системы в случае проникновения).

3. Стрессовое тестирование (тестовые варианты направлены на оценку стабильности работы ПС при «ненормальных» нагрузках: по частоте запросу, объему данных). Частный случай – тестирование чувствительности.

4. Тестирование производительности (проверяется скорость работы ПО в компьютерной системе).

21. Методы отладки программного обеспечения. Классификация ошибок.

Отладка – это локализация и устранение ошибок. Она следует за успешным тестированием программного обеспечения.

Основной сложностью в процессе отладки является именно обнаружение оператора, содержащего ошибку.

Способы отслеживания ошибки – аналитические и экспериментальные.

Классификация ошибок по принадлежности к этапу обработки программы

Способы проявления ошибок выполнения:

- появление системного сообщения об ошибке низкого уровня;
- появление системного сообщения об ошибке;
- «зависание» компьютера;
- несовпадение полученных результатов с ожидаемыми.

Группы причин ошибок выполнения:

- неверное определение исходных данных;
- логические ошибки;
- накопление погрешностей результатов вычислений.

Классификация ошибок выполнения по возможным причинам

22. Общая методика отладки программного обеспечения. Способы отладки.

Аналитические методы отладки:

1. Метод ручного тестирования.

2. Метод индукции.

Основан на тщательном анализе симптомов ошибки. Информацию организуют и тщательно изучают. Выдвигают гипотезы об ошибках, каждую из которых проверяют.

3. Метод дедукции.

Формируют множество возможных причин. Анализируя причины, исключают невозможные. Наиболее вероятную гипотезу пытаются доказать.

4. Метод обратного прослеживания.

Эффективен для небольших программ. Для точки вывода некорректного результата строится гипотеза о значениях основных переменных, которые могли бы привести к получению имеющегося результата. Делают предложения о значениях переменных в предыдущей точке и т.д. до обнаружения причины ошибки.

1. Отладочный вывод

Основная идея: включение дополнительных операторов вывода в узловых точках.

В настоящее время используется редко. Недостаток: обычно большой объем вывода.

2. Интегрированные средства отладки

Основные функции:

- пошаговое выполнение программы;
- поддержка точек останова;
- выполнение программы «до курсора»;
- отображение содержимого переменных при пошаговом выполнении;
- отслеживание потока сообщений;
- и т.п.

3. Использование независимых отладчиков

1 этап – изучение проявления ошибки (используются индуктивные и дедуктивные методы). Выдвигаются и проверяются версии (могут примениться методы и средства получения дополнительной информации).

2 этап – локализация ошибки (путем отсечения частей программы или с использованием отладочных средств).

3 этап – (если ошибка не в том месте, где она проявилась) определение причины ошибки - изучение результатов второго этапа и формирование версий возможных причин ошибки.

4 этап – исправление ошибки.

5 этап - повторное тестирование.

23. Регрессионное тестирование.

Под регрессионным тестированием понимают те виды тестов, которые проводятся с каждой новой версией программы. Иными словами, тесты регрессии - это своего рода "старые песни о главном". Цель проведения этих тестов проста - убедиться, что новая версия программы не содержит ошибок в уже протестированных участках кода. По данным зарубежных авторов количество ошибок, возникающих в процессе изменения кода (исправления багов, внедрения новой функциональности и т.п.) колеблется от 50% до 80%. Выявить эти ошибки и помогают тесты регрессии.

Таким образом регрессионное тестирование - понятие комплексное. Рассмотрим основные виды тестов регрессии:

1.Верификационные тесты (Verification Test).

•Тесты верификация багов (Bug Verification Test). Представляют собой тесты проверки исправления багов. Приведем пример. Допустим, что тест с номером 3 выявил баг, что было зафиксровано и передано разработчику для исправления. Через определенное время Вы получили от разработчика новую версию программы, с информацией о том, что описнный баг исправлен. Ваша задача - провести тест с номером 3 повторно - для того, чтобы убедиться, что баг действительно больше не проявляется. В случае успешного прохождения теста такой баг помечается как Verified, в противном случае - как re-do, о чем сообщается разработчику и передается на доработку. Проведение таких тестов является обязательным. Так как причин, из-за которых исправленный баг может сохраниться в программе - множество (от ошибочного описания, а, возможно, и понимания проблемы, до ошибочного утверждения о том, что исправление имело место).

•Тесты верификации версии (Build Verification Test; Build Acceptance Test, smoke test, quick check). Представляют собой набор тестов для проверки сохранности основной функциональности в каждой новой версии программы. Иными словами - это краткое тестирование всех основных функций разрабатываемого ПО, цель которого - убедится, что программа "работает нормально", что основная функциональность программы не нарушена. Если

хотя бы один из тестов верификации версии выявляет баг - то тестер возвращается к предыдущей (последней "рабочей"), дальнейшей тестирование новой версии не проводится, а информация об ошибке вносится в базу и отправляется разработчику. Т.о. тесты верификации версии представляют собой краткий набор основных тестов функциональности.

2.Собственно Тесты Регрессии (или Regression Test Pass). Под этим понятием объединяют те тесты, которые уже проводились с предыдущими версиями программы, притом успешно, т.е. не выявили багов и были отмечены (например в TCM) как pass (passed). Необходимость проведения таких тестов очевидна. Приведем пример. Допустим, что ранее проведенный тест № 2, который обеспечивал проверку в программе участка кода (назовем его условно кодом-А) не выявил ошибок в программе, и был отмечен как pass. В ходе разработки возникла необходимость изменить участок кода-А (например, при исправлении какого либо иного бага или же придания программе новой функциональности). В результате этот участок кода требует дополнительной проверки, что и будет сделано при повторном проведении теста № 2. Среди Собственно Тестов Регрессии можно выделить две группы. Первая - тесты, входящие в набор (т.н. Regression Test Pass with Regression Test Suit), другие - тесты не входящие в набор (т.н. Regression Test Pass without Regression Test Suit). Существенные отличия между ними в следующем: первые - вносятся в базу и описываются, для них могут и должны быть созданы скрипты, которые позволяют автоматизировать процесс тестирования; вторые - существуют только "в голове" тестировщика и проводятся в ручную, причин этого может быть много - от малых сроков тестирования, до отсутствия необходимого ПО, для автоматизации процесса.

3.Тесты регрессии на "закрытых" багах. Рассмотрим пример. Допустим, что тест № 3, выявивший баг, после исправления этого бага разработчиком был проведен повторно, при том успешно. Тест был отмечен как pass, а баг - как Verified. Такой баг откладывается "на полочку", "дело" закрыто. Такой баг и будет "закрытым". Допустим теперь, что в ходе разработки, участок кода, где был исправлен этот баг был изменен, или сменился разработчик, который

случайно удалил "нашлепку" в коде исправлявшую этот глюк и показавшуюся ему лишней и т.п. В этом случае баг проявится снова. Что бы не допустить подобного бета-тестеру время от времени необходимо проводить тесты, выявлявшие ранее баги в измененном участке кода, исправление которых уже было проверено ранее и зафиксировано в базе. Это и есть Тесты регрессии на "закрытых" багах.

Далее описаны лишь общие положения:

1. Регрессионное тестирование проводится в каждой новой версии.
2. Начинают регрессионное тестирование с Тестов верификации версии. Если программа приходит от разработчика в виде полноценной инсталляции, то Тесты верификации начинаются с проверки инсталляции, после чего проводится краткий набор тестов функциональности. Если хотя бы один из тестов failed, версия передается на доработку, регрессионное тестирование прекращается, а тестер возвращается к тестированию последней "рабочей" версии.
3. После успешного прохождения тестов верификации версии, проводят серию Тестов верификации багов.
4. Из Собственно тестов регрессии проводят лишь те, которые сопряжены с измененным в новой версии участком кода.
5. Аналогичным образом (см. пункт 4) отбираются тесты в группу регрессии на "закрытых" багах.
6. Тесты регрессии, выполненные успешно (pass) дважды считаются "закрытыми". Дальнейшее их использование производится так как описано в пункте 4.
7. Для тестов регрессии, которые предполагается проводить более 3-5 раз рекомендуется писать скрипты для автоматизации процесса. Это относится ко всемгруппам тестов регрессии.
8. Отбор тестов для Финального регрессионного тестирования осуществляется по следующим принципам:

- В первую очередь отбирают тесты забракованные (failed) два и более раз. В том числе и те, которые выявляли баги, требующие доработки (re-do).
- Во вторую очередь отбираются тесты забракованные один раз, и успешно пройденные повторно.
- Далее отбираются все тесты, которые были пройдены успешно (pass), но проводились только один раз.
- Затем проводятся все остальные тесты, в зависимости от поставленной задачи.

Для наглядности при проведении Регрессионного тестирования можно использовать следущую таблицу:

№ теста	№ версии	№ бага	№ версии	№ бага		

Количество столбцов соответствует количеству версий.

24. Нагрузочное тестирование. Понятие, основные виды, термины.

Нагрузочное тестирование (Load Testing) или **тестирование производительности** (Performance Testing) - это **автоматизированное тестирование**, имитирующее работу определенного количества бизнес пользователей на каком-либо общем (разделяемом ими) ресурсе.

Начиная работу в области нагрузочного тестирования, следует четко понимать, что это не просто запись и прогон (Record and Playback) скриптов, а более сложный процесс:

- **Во-первых**, нагрузочное тестирование - это серьезная исследовательская и аналитическая работа
- **Во-вторых** - это реальное автоматизированное тестирование, требующее серьезных навыков программирования, а также знания сетевых протоколов и различных серверов приложений и баз данных
- **В-третьих** - существуют разные **виды нагрузочного тестирования**, ставящие перед собой **разные цели**

Рассмотрим **основные виды** нагрузочного тестирования, также задачи стоящие перед ними.

Тестирование производительности (Performance testing)

Задачей тестирования производительности является определение масштабируемости приложения под нагрузкой, при этом происходит:

- измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций
- определение количества пользователей, одновременно работающих с приложением
- определение границ приемлемой производительности при увеличении нагрузки (при увеличении интенсивности выполнения этих операций)
- исследование производительности на высоких, предельных, стрессовых нагрузках

Стрессовое тестирование (Stress Testing)

Стрессовое тестирование позволяет проверить насколько приложение и система в целом работоспособны в условиях стресса и также оценить способность системы к регенерации, т.е. к возвращению к нормальному состоянию после прекращения воздействия стресса. Стрессом в данном контексте может быть повышение интенсивности выполнения операций до очень высоких значений или аварийное изменение конфигурации сервера. Также одной из задач при стрессовом тестировании может быть оценка деградации производительности, таким образом цели стрессового тестирования могут пересекаться с целями тестирования производительности.

Объемное тестирование (Volume Testing)

Задачей объемного тестирования является получение оценки производительности при увеличении объемов данных в базе данных приложения, при этом происходит:

- измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций
- может производиться определение количества пользователей, одновременно работающих с приложением

Тестирование стабильности или надежности (Stability / Reliability Testing)

Задачей тестирования стабильности (надежности) является проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки. Время выполнения операций может играть в данном виде тестирования второстепенную роль. При этом на первое место выходит отсутствие **утечек памяти**, перезапусков серверов под нагрузкой и другие аспекты влияющие именно на стабильность работы.

Терминология

Нагрузочное тестирование или **Тестирование производительности** - это **автоматизированное тестирование**, имитирующее работу определенного количества бизнес пользователей на каком-либо общем (разделяемом ими) ресурсе. В качестве примера можно привести работу сотрудников современного банка, в котором все работают с одними и теми же программными приложениями, установленными на банковских серверах. Или использование программного приложения веб магазин, в данном случае посетителями, нагружающими сервера, будут пользователи интернета.

1. **Виртуальный пользователь (Virtual User)** - программный процесс, циклически выполняющий моделируемые операции
2. **Итерация (Iteration)** – это один повтор выполняемой в цикле операции

3. **Интенсивность выполнения операции** (**Operation Intensity**) - частота выполнения операции в единицу времени, в тестовом скрипте задается интервалом времени между итерациями
4. **Нагрузка** (**Loading**) - совокупное выполнение операций на общем ресурсе (тр./сек, хитов/сек)
5. **Производительность** (**Performance**) - количество выполняемых операций за период времени (N операций за M часов)
6. **Масштабируемость приложения** (**Application Scalability**) - пропорциональный рост производительности при увеличении нагрузки
7. **Профиль нагрузки** (**Performance Profile**) - это набор операций с заданными интенсивностями, полученный на основе сбора статистических данных либо определенный путем анализа требований к тестируемой системе
8. **Нагрузочной точкой** называется рассчитанное (либо заданное Заказчиком) количество виртуальных пользователей в группах, выполняющих операции с определенными интенсивностями

25. Нагрузочное тестирование. Этапы и методика проведения.

Рассматривая **этапы** проведения **нагрузочного тестирования**, хотелось бы отметить следующие, на наш взгляд обязательные:

1. **Анализ требований и сбор информации о тестируемой системе**
2. **Конфигурация тестового стенда для нагрузочного тестирования**
3. **Разработка модели нагрузки**
4. **Выбор инструмента для нагрузочного тестирования**
5. **Создание и отладка тестовых скриптов**
6. **Проведение тестирования**
7. **Анализ результатов**
8. **Подготовка, отправка и публикация отчета по проведенному нагрузочному тестированию**

5.1 Анализ требований

При анализе требований основной упор необходимо сделать на определение основных критериев успешности проведенных тестов. Для этого Вам необходимо будет выделить следующие характеристики:

- **Время отклика** (время необходимое для открытия страницы или получения ожидаемого результата)
- **Интенсивность** (число запросов в секунду – (Qps)
- **Используемые ресурсы** (загрузка процессора, кол-во используемой памяти, дисковое и сетевой I/O)
- **Максимальное количество пользователей** (определяет число пользователей, способных работать с системой в условиях заданной конфигурации)

А также некоторые метрики связанные с работой бизнес сценариев (например, количество бизнес операций в единицу времени, время выполнения бизнес операции и т.д.)

Заданные в требованиях характеристики, будут являться **базовыми нагрузочными точками** тестируемого приложения. Все получаемые результаты будут сравниваться с ними для принятия решения о завершении тестирования либо дальнейшем профилировании производительности.

При анализе требований необходимо учесть разрабатывается ли новое (**startup project**) или же проект направлен на профилирование нагрузки для уже находящегося в эксплуатации приложения (**profiling project**).

5.1.1 Анализ целевой аудитории

Общение непосредственно с пользователями, наблюдение за их действиями, помогает нам более точно понять как будет происходить работа с ПО, какие части приложения наиболее критичны и требуют особого внимания.

Например:

Утром операторы проверяют журналы посещения, обрабатывают сообщения клиентов, после обеда вводят новые товары в базу, и в конце рабочего дня снова анализируют журналы и сообщения. Значит основная нагрузка на журналы посещения и базы сообщений будет в начале и в конце рабочего дня.

5.1.2 Определение базового профиля нагрузки

Выделив, на предыдущем шаге группы пользователей, бизнес сценарии, требования к системе можем приступить к созданию профиля нагрузки – набор операций и интенсивность их выполнения для каждой пользовательской группы.

Зная количество пользователей в каждой группе, мы должны будем распределить их для выполнения требуемых операций, сохранив необходимую интенсивность.

Проведя аналогичный перерасчет для каждой группы пользователей, мы получим небольшую сводную таблицу со списком операций и интенсивностью их выполнения:

группа	**кол-во юзеров**	**операция**	**интенсивность опер./ сек**
Админы	Z	добавить пользователя	N
Админы	Z	забанить пользователя	N
...			
покупатели	Z	вход/выход в систему	N
покупатели	Z	поиск товара	N
покупатели	Z	покупка товара	N
...			
Продавцы	Z	просмотр посещения	N
Продавцы	Z	добавление/удаление товара	N

Именно эта таблица и будет являться полным **базовым профилем нагрузки** (**performance baseline profile**) для тестируемого приложения.

5.1.3 Разработка моделей нагрузки

В зависимости от вида проводимого тестирования и целей преследуемых им, нам придется разрабатывать разные модели нагрузки.

Так разным может быть:

- количество пользователей и интенсивность запросов
- старт самих пользователей: последовательно по одному, ступенчато группами или же запуск всех сразу.
- длительность сценария от 10 минут до нескольких часов или даже дней.

Проще всего изменения проводить, варьируя параметрами в базовом профиле нагрузки и инструменте для тестирования производительности (например, в HP LoadRunner Controller).

Модель тестирования производительности

Постепенное увеличение нагрузки, добавляя новых пользователей с некоторым интервалом времени, позволяет нам определить:

- измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций
- количество пользователей, способных одновременно работать с приложением
- границы приемлемой производительности при увеличении нагрузки
- производительность при разных нагрузках

Модель стрессового тестирования

Увеличивая интенсивность операций выше пиковых (максимально разрешенных) значений либо увеличивая количество пользователей, до тех пор пока нагрузка не станет выше максимально допустимых значений, проверяем, что система работоспособна в условиях стресса. Далее, опустив нагрузку до средних значений, проверяем (способность системы к регенерации), что система вернулась к нормальному состоянию (основные нагрузочные характеристики не превышают базовые).

Модель объемного тестирования

Можно использовать ту же модель что и для тестирования производительности однако целью будет проверка работы системы с прогнозом на будущий рост объема данных. Следовательно одним и самым важным предусловием теста будет увеличение объемов базы данных приложения до

требуемых размеров. Таким образом мы сможем проверить и оценить производительность, прогнозируя рост системы на год, два или три вперед.

Модель тестирования стабильности или надежности

Используя базовый нагрузочный профиль, запускаем тест длительностью от нескольких часов до нескольких дней, с целью выявления утечек памяти, перезапуска серверов и других аспектов влияющих на нагрузку.

5.2 Конфигурация тестового стенда

Отметим те части конфигурации, которые требуют особого внимания:

- **Hardware**
  - процессор (тип, частота, количество ядер и т.д)
  - оперативная память (тип, объем, тайминг, эффективная частота и т.д.)
  - жесткие диски (тип, скорость и т.д.)
- **Software**
  - Операционная система
  - Драйвера
- **Network**
  - топология сети
  - пропускная способность
  - протокол передачи данных
- **Application**
  - Архитектура
  - База данных (структура + данные)
  - программное обеспечение, необходимое для работы приложения (например, для Java приложений - JVM)

В самом идеальном случае **тестовый стенд один к одному дублирует конфигурацию реального сервера**, на котором работает или же будет работать приложение. Однако, как мы с вами знаем, идеальных случаев практически не бывает (то памяти мало, то процессора такой частоты нет в наличии, то операционная система не той версии, то стоимость некоторого серверного ПО

не укладывается в бюджете). Перечислим основные причины, по которым не всегда получается продублировать конфигурацию системы на тестовом стенде:

1. Сложность дублирования дорогого серверного железа для тестовых нужд
2. Ограничения на использование лицензий требуемого программного обеспечения
3. Закрытость архитектуры приложения со стороны заказчика по соображениям безопасности
4. Трудность воссоздания или транспортировки базы данных приложения
5. Сложность воссоздания требуемой архитектуры сети
6. и многое другое (всё перечислить крайне сложно из-за большого количества нюансов, влияющих на конфигурацию системы)

Целесообразность же воссоздания инфраструктуры необходимо оценить с учетом выделенных ресурсов, времени и усилий, так как не всегда результат будет оправдывать средства.

5.2.1 Анализ результатов при разных конфигурациях

Как же быть с тем фактом, что приложение будет протестировано на одной платформе, а работать ему придется на другой? Как провести анализ результатов? Как сделать окончательные выводы и спрогнозировать работу приложения под нагрузкой в будущем?

Использование переходного коэффициента

В своей статье **[1]** Вячеслав Берзин, предложил при анализе результатов использовать “переходный коэффициент”, показывающий численно отличие производительности тестовой платформы от промышленной:

"Если существует отличие в конфигурациях тестового стенда и тестируемой системы, необходима оценка переходного коэффициента для производительности тестовой и промышленной систем. На основании этого коэффициента, проводится масштабирование полученных результатов. Для получения погрешности оценки, не превышающей 5-10%, рекомендуется проведение оценки переходного коэффициента несколькими независимыми

способами, например, на основании анализа независимых данных о производительности используемых аппаратных платформ."

Использование данного подхода, значительно облегчает создание и конфигурацию тестового стенда, но прибавляет работы на этапе анализа результатов. Так что вам придется решать самим: либо пытаться один к одному (100%) скопировать промышленную систему, либо сделать её максимально приближенной и с помощью переходного коэффициента анализировать и делать прогнозы на будущий рост производительности.

Однако, не всегда возможно точно получить значение переходного коэффициента из-за нелинейных зависимостей между компонентами системы, а примерный расчет может дать слишком большую погрешность. Поэтому использование данного подхода видится реальным, когда тестовая и реальная конфигурации системы практически идентичны.

Использование экспертной оценки

Рассмотрим мнение Андрея Широбокова **[2]** (сертифицированный специалист в области нагрузочного тестирования, имеющий более 15 лет рабочего опыта в IT):

"Идеально ключевые моменты должны совпадать, пишу по убыванию важности:

1. конфигурация (кол-во и тип процессоров х память)
2. версия ОС и статс паки
3. версия и тип СУБД
4. версия серверов приложений и патчей

Естественно, что версия тестируемого приложения должна быть актуальна.

Если совпадает платформа (ОС и тип процессоров), то допускается "пропустить" конфигурацию, например, в случае если проведены эксперименты на конфигурациях 8х16 (8 прц х 16 Гб) и 32х64 (32 прц х 64 Гб), то можно "аппроксимировать" недостающую конфигурацию 24х48 (24 прц х 48 Гб). Обратим внимание, что памяти для такой методики должно пропорционально быть в 2 раза больше во всех случаях.

Если не совпадает платформа, ОС, типы процессоров, объемы памяти, то можно делать только очень общие выводы, наподобие: "В реальности будет не хуже". Но времена откликов и другие детали могут быть малозначимы."

5.3 Разработка модели нагрузки

Определившись с **видами нагрузочного тестирования**, **целями** и **терминологией**, перейдем к основной задаче нагрузочного тестировании - **разработке модели нагрузки**.

Для этого необходимо определить следующее:

- список тестируемых операций
- интенсивность выполнения операций
- зависимость изменения интенсивности выполнения операций от времени

В список тестируемых задач должны войти операции, критичные с точки зрения бизнеса, а также с технической точки зрения:

- Критичными с точки зрения бизнеса являются операции, скорость выполнения которых, реально влияет на производительность бизнес процесса. *Например, увеличение длительности обслуживания клиентов в банке, невозможность выполнения необходимого количества операций в течение дня и так далее*.
- Критичными с технической точки зрения являются ресурсоемкие операции, требующие большое количество памятьи, серьезно задействующие процессор, создающие значительный сетевой трафик. *Как правило, это операции выполняемые одновременно большим количеством бизнес пользователей или создание сложных отчетов, в которые входят так называемые "тяжелые" запросы к базе данных*.

Хотим еще раз подчеркнуть, что *под степенью критичности операции мы подразумеваем её влияние на бизнес процесс и работоспособность системы*. Например, создание какого-нибудь отчета, полностью загружающего сервер базы данных в ночное время, не будет носить высокий приоритет для оптимизации, а в рабочие часы будет иметь максимальный приоритет.

5.3.1 Изучение Приложения

Будем называть тестируемое прикладное программное обеспечение "приложением". Чтобы выделить части приложения, а именно операции, которые будут тестироваться, необходимо провести работу связанную с изучением приложения. Очень большую пользу при этом должны оказать разработчики приложения, если речь идет о тестировании в процессе разработки, либо бизнес пользователи и системные администраторы, если приложение находится в процессе эксплуатации. В ходе этой работы разумно сделать такие шаги:

- Описать компоненты приложения и составить схемы взаимодействия между ними
- Выделить критические с точки зрения предполагаемого тестирования операции. В качестве таковых могут быть выбраны:

1. Операции с "тяжелыми" запросами к базе данных, процессы генерации отчетов
2. Операции, выполняемые большим количеством пользователей или с высокой интенсивностью
3. Операции критичные с точки зрения бизнеса, и к тому же удовлетворяющие условиям двух верхних пунктов

Еще раз хочется заметить, что опрос бизнес пользователей или совместное исследование с разработчиками и администраторами системы может значительно облегчить задачу. Если приложение находится в эксплуатации, то можно провести мониторинг загрузки компонентов аппаратных серверов (процессора, память, диски) и проанализировать системные журналы веб серверов (снять stats pack, если в качестве сервера базы данных, например, используется Oracle). Системные журналы могут показать пики высокой активности пользователей в течение дня и дать количественное оценки того сколько транзакций (хитов) выполняется в единицу времени. Согласно **закону Паретто или принципу 20/80**, 20% операций приложения

генерируют 80% нагрузки в системе, поэтому нужно стараться выбрать для моделирования именно эти 20% операций.

5.3.2 Определение профиля нагрузки

Ключевым моментом в модели нагрузки являются выбранные для тестирования операции или **профиль нагрузки**. Естественно выполняться эти операции в тесте должны одновременно. **Профилей нагрузки для приложения может быть несколько** и это оправдано. Ведь бизнес пользователи могут выполнять разные наборы операций в разное время. Например, начало операционного дня и конец дня, начало месяца (квартала) и соответственно завершение могут отличаться. Таким образом получаем различные наборы операций приложения, выполняющиеся одновременно и соответственно создающие различную нагрузку. Кстати, **меняться могут не только сами операции но и их интенсивности**. В первом приближении моделью нагрузки является набор профилей нагрузки, где каждый профиль отличается от другого или набором операций или интенсивностями выполнения этих операций.

Пример профиля нагрузки, в который входит 5 операций, значение n может быть различным для каждой операции:

<Профиль нагрузки>

1. Операция_1 - интенсивность выполнения n раз / ед. времени
2. Операция_2 - интенсивность выполнения n раз / ед. времени
3. Операция_3 - интенсивность выполнения n раз / ед. времени
4. Операция_4 - интенсивность выполнения n раз / ед. времени
5. Операция_5 - интенсивность выполнения n раз / ед. времени

5.3.3 Расчет нагрузочных точек

Поскольку **в профиле нагрузки как правило присутствует несколько операций** - это означает, что у нас будет несколько групп пользователей. Желательно моделировать каждую операцию отдельной группой виртуальных пользователей (хотя в жизни часто бывает наоборот, один бизнес пользователь может отвечать за выполнение нескольких операций). Тем не менее, если назначить одному виртуальному пользователю выполнение одной операции, то

так легче выдержать определенную интенсивность (и соответственно производительность) для этой операции в тесте, чем в случае, когда виртуальному пользователю назначается последовательная цепочка операций. Зная интенсивность выполнения операции нужно определить количество виртуальных пользователей в группе, выполняющих эту операцию. **Идеальный случай** когда работа с приложением аналогична работе заводского конвейера и **есть точные оценки сколько операций в день делает один пользователь**. Чаще всего бывает не так и известно только общее количество операций выполняемое в течение дня. Так же может оказаться, что интенсивность выполнения операции каждым пользователем очень низкая, например, один пользователь выполняет операцию раз в день или раз в два дня.

5.3.4 Baseline нагрузочная точка

Хочется заметить что расчет нагрузочной точки, описанный в разделе **Расчет точек нагрузки**, основанный на собранной для работающего приложения статистике (или на ожидаемом объеме работы для вновь разработанного приложения), является исходным для дальнейшего роста нагрузки, а сама **рассчитанная нагрузочная точка может считаться базовой или baseline точкой**. Теперь можно увеличивать нагрузку, двигаясь с некоторым шагом, увеличивая при этом только количество виртуальных пользователей в группах, не изменяя интенсивности выполнения операций для одного виртуального пользователя.

Полная модель нагрузки - это набор профилей нагрузки со всеми нагрузочными точками для каждого профиля. При разработке тестовых сценариев должны быть корректно реализованы все нагрузочные точки. Еще хотелось бы добавить, что нагрузочных точек для каждого профиля должно быть не меньше трех, чтобы можно было оценить зависимость времен отклика выполняемых операций от роста нагрузки. Очевидно, что чем линейнее такая зависимость тем лучше масштабируемость приложения и выше предсказуемость его поведения под нагрузкой.

26. Объектно-ориентированное тестирование программного обеспечения. Особенности и принципы.

Разработка объектно-ориентированного ПО начинается с создания визуальных моделей, отражающих статические и динамические характеристики будущей системы.

На конструирование моделей приходится большая часть затрат объектно-ориентированного процесса разработки. Если к этому добавить, что цена устранения ошибки стремительно растет с каждой итерацией разработки, то совершенно логично требование тестировать объектно-ориентированные модели анализа и проектирования.

Критерии тестирования моделей: правильность, полнота, согласованность

О **синтаксической правильности** судят по правильности использования языка моделирования (например, UML).

О **семантической правильности** судят по соответствию модели реальным проблемам.

О **полноте** судят по наличию в модели всех существенных элементов моделируемой предметной области с точки зрения решаемой задачи.

О **согласованности** судят путем рассмотрения противоречий между элементами в модели. Несогласованная модель имеет в одной части представления, которые противоречат представлениям в других частях модели.

Для оценки согласованности моделей применяется модель Класс – Обязанность – Сотрудник CRC (Class — Responsibility — Collaboration). Основной элемент этой модели — **CRC-карта.**

CRC-карта — это расчерченная карточка размером 6 на 10 сантиметров. Она помогает установить задачи класса и выявить его окружение (классы-собеседники). Для каждого класса создается отдельная карта.

В каждой CRC-карте указывается имя класса, его обязанности (операции) и его сотрудничества (другие классы, в которые он посылает сообщения и от которых он зависит при выполнении своих обязанностей). Сотрудничества

подразумевают наличие ассоциаций и зависимостей между классами. Они фиксируются в других моделях — диаграмме сотрудничества (последовательности) объектов и диаграмме классов.

Шаги оценки модели с помощью CRC-карт.

1. Выполняется перекрестный просмотр CRC-карты и диаграммы сотрудничества (последовательности) объектов. Цель — проверить наличие сотрудников, согласованность информации в обеих моделях.
2. Исследуются обязанности CRC-карты. Цель — определить, предусмотрена ли в карте сотрудника обязанность, которая делегируется ему из данной карты.
3. Организуется проход по каждому соединению CRC-карты. Проверяется корректность запросов, выполняемых через соединения.
4. Определяется, требуются ли другие классы, или правильно ли распределены обязанности по классам. Для этого используют проходы по соединениям, исследованные на шаге 3.
5. Определяется, нужно ли объединять часто запрашиваемые обязанности.
6. Шаги 1-5 применяются итеративно, к каждому классу и на каждом шаге эволюции объектно-ориентированной модели.

Особенности тестирования объектно-ориентированных «модулей»:

При рассмотрении объектно-ориентированного ПО меняется понятие модуля. Наименьшим тестируемым элементом теперь является класс (объект). Класс содержит несколько операций и свойств. Поэтому сильно изменяется содержание тестирования модулей.

В данном случае нельзя тестировать отдельную операцию изолированно, как это принято в стандартном подходе к тестированию модулей. Любую операцию приходится рассматривать как часть класса.

27. Способы объектно-ориентированного тестирования ПО.

- тестированию модулей традиционного ПО соответствует тестирование классов объектно-ориентированного ПО;

• тестирование традиционных модулей ориентировано на поток управления внутри модуля и поток данных через интерфейс модуля;

• тестирование классов ориентировано на операции, инкапсулированные в классе, и состояния в пространстве поведения класса.

Тестирование объектно-ориентированной интеграции.

Объектно-ориентированное ПО не имеет иерархической управляющей структуры, поэтому здесь неприменимы методики как восходящей, так и нисходящей интеграции. Мало того, классический прием интеграции (добавление по одной операции в класс) зачастую неосуществим.

Методики тестирования:

1.Тестирование, основанное на потоках.

2.Тестирование, основанное на использовании.

Тестирование, основанное на потоках

Объектом интеграции является набор классов, обслуживающий единичный ввод данных в систему. Для проверки отсутствия побочных эффектов применяют регрессионное тестирование.

Тестирование, основанное на использовании.

Вначале интегрируются и тестируются независимые классы. Далее работают с первым слоем зависимых классов (которые используют независимые классы), со вторым слоем и т. д. В отличие от стандартной интеграции, везде, где возможно, избегают драйверов и заглушек.

Кластерное тестирование.

Кластер сотрудничающих классов определяется исследованием CRC-модели или диаграммы сотрудничества объектов. Тестовые варианты для кластера ориентированы на обнаружение ошибок сотрудничества.

Объектно-ориентированное тестирование правильности

При проверке правильности исчезают подробности отношений классов. Как и традиционное подтверждение правильности, подтверждение правильности объектно-ориентированного ПО ориентировано на видимые действия пользователя и распознаваемые пользователем выводы из системы.

К операциям класса применимы классические способы тестирования «белого ящика», которые гарантируют проверку каждого оператора и их управляющих связей. При большом количестве операций от тестирования по принципу «белого ящика» приходится отказываться. Меньших затрат потребует тестирование на уровне классов.

Способы тестирования «черного ящика» также применимы к объектно-ориентированным системам. Полезную входную информацию для тестирования «черного ящика» и тестирования состояний обеспечивают элементы Use Case.

Тестирование поверхностной и глубинной структуры

Поверхностная структура — это видимая извне структура объектно-ориентированной системы. Она отражает взгляд пользователя, который видит не функции, а объекты для обработки. Тестирование поверхностной структуры основывается на задачах пользователя. Главное — выяснить задачи пользователя. Для разработчика это нетривиальная проблема, поскольку требует отказа от своей точки зрения.

Глубинная структура отражает внутренние технические подробности объектно-ориентированной системы (на уровне проектных моделей и программного текста). Тесты глубинной структуры исследуют зависимости, поведение и механизмы взаимодействия, которые создаются в ходе проектирования подсистем и объектов.

Стохастическое тестирование класса.

При стохастическом тестировании исходные данные для тестовых вариантов генерируются случайным образом.

Рассмотрим класс Счет, который имеет следующие операции: Открыть, Установить, Положить, Снять, Остаток, Итог, ОграничитьКредит, Закрыть.

Каждая из этих операций применяется при определенных ограничениях:

1. счет должен быть открыт перед применением других операций;
2. счет должен быть закрыт после завершения всех операций.

Минимальная работа экземпляра Счета включает следующую последовательность операций:

Открыть ► Установить ►Положить ► Снять ► Закрыть.

В эту последовательность можно встроить группировку, обеспечивающую создание других вариантов поведения:

О т к р ы т ь ► У с т а н о в и т ь ► П о л о ж и т ь ► [Остаток●Снять●Итог●ОграничитьКредит●Положить]n ►Снять ► Закрыть.

Тестирование разбиений на уровне классов.

Тестирование разбиений уменьшает количество тестовых вариантов, требуемых для проверки классов (тем же способом, что и разбиение по эквивалентности для стандартного ПО). Области ввода и вывода разбивают на категории, а тестовые Варианты разрабатываются для проверки каждой категории.

Три способа разбиения:

1. разбиение на категории по состояниям.
2. разбиение на категории по свойствам.
3. разбиение на категории по функциональности

Разбиение на категории по состояниям.

Основывается на способности операций изменять состояние класса. Обратимся к классу Счет. Операции Снять, Положить изменяют его состояние и образуют первую категорию. Операции Остаток, Итог, ОграничитьКредит не меняют состояние Счета и образуют вторую категорию. Проектируемые тесты отдельно проверяют операции, которые изменяют состояние, а также те операции, которые не изменяют состояние.

Таким образом, для нашего примера:

Тестовый вариант 1:

Открыть ►Установить ►Положить ►Положить ►Снять ►Снять ►Закрыть.

Тестовый вариант 2:

О т к р ы т ь ►У с т а н о в и т ь ►П о л о ж и т ь ►О с т а т о к ►И т о г ►ОграничитьКредит ►Снять ►Закрыть.

Разбиение на категории по свойствам.

Основывается на свойствах, которые используются операциями. В классе Счет для определения разбиений можно использовать свойства остаток и

ограничение кредита. Например, на основе свойства ограничение кредита операции подразделяются на три категории:

1) операции, которые используют ограничение кредита;
2) операции, которые изменяют ограничение кредита;
3) операции, которые не используют и не изменяют ограничение кредита.

Разбиение на категории по функциональности.

Основывается на общности функций, которые выполняют операции. Например, операции в классе Счет могут быть разбиты на категории:

операции инициализации (Открыть, Установить);

вычислительные операции (Положить, Снять);

запросы (Остаток, Итог, ОграничитьКредит);

операции завершения (Закрыть).

28. Тестирование пользовательского интерфейса.

Ошибки GUI

• Неверная реакция на взаимодействие с элементами

• Неправильная навигация между экранными элементами

• Не отображаются нужные элементы

• Проблемы синхронизации между действиями над графическими элементами

Критерии выбора тестов

• Тестирование интерфейса ближе к комплексному тестированию. Тесты на интерфейс не должны повторять модульные тесты на бизнес-логику.

• Интерфейс может (и должен) иметь собственные функциональные требования

Выбор тестов

• Неверная реакция на взаимодействие с элементами (наиболее очевидно)

• Тесты на логику работы

• Неправильная навигация между экранными элементами

• Тесты на навигацию

• Не отображаются нужные элементы • Сравнение изображений

• Проблемы синхронизации между действиями над графическими элементами

• Нагрузочное тестирование интерфейса

Тесты на функциональные требования

• Покрывают основные пользовательские сценарии

• За один проход

• Вперед-назад

• Отмена и начало заново

• Покрывают различные комбинации пользовательских сценариев

• Регрессионные тесты

Неправильная навигация

• Неверно передается фокус

• Неверно блокируются и разблокируются

элементы управления

Проблемы синхронизации

• При совершении действия форма надолго зависает

• Определить критерии для времени отклика • Выполнение функциональных сценариев в

длинном цикле с максимально короткими

задержками между командами

• Как и любой сценарий, связанный с

тестированием многопоточности,

достаточно технически тяжел в реализации • Синхронизация бизнес-логики не входит в

рассмотрение тестов на GUI

29. Тестирование требований.

30. Автоматизированное тестирование программного обеспечения.

Так как сейчас, основной парадигмой разработки программного обеспечения является ООП, то многие программы состоят из классов,

подсистем и систем. На каждом уровне, что логично, используется свой подход к тестированию:

Само собой, эта картинка не раскрывает все виды тестирования, ведь еще есть нагрузочное тестирование, тестирование установки, тестирование на отказ и восстановление, тестирование безопасности и т.д.

Т.к. про любой из видов тестирования можно писать целые книги, то сегодня, предлагаю ограничиться только модульным тестированием (unit testing).

Для начала, давайте посмотрим на определения, что же такое модульные тесты:

Модульное тестирование, или юнит-тестирование (англ. unit testing) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы. (wikipedia.org)

Модульный тест — это автоматизированный фрагмент кода, который вызывает тестируемый метод или класс, а затем проверяет несколько предположений относительно логического поведения метода или класса. (Джефф Левинсон)

Модульный тест — это код, который обеспечивает выполнение части вашего рабочего кода с ожиданием результата. (кто автор не понятно, но взял здесь)

Модульный тест — это метод, который тестирует метод, класс или более крупный компонент вашего приложения изолированно от других частей, внешних систем и ресурсов. (Ларри Брейдер, Алан Кэмерон Уиллс)

Как видите, даже в определениях можно найти такое вот разнообразие. Мне больше всего нравится последние определение. Четко и однозначно

определяющее, что модульные тесты, это изолированные тесты. Когда тестируемая сущность не взаимодействует с привычным окружением, которое тоже может быть причиной появления ошибок.

31. Разработка тестового драйвера для тестирования ПО.

Необходимо разработать программу тестового драйвера для тестирования программных модулей. На вход данная программа получает путь к исходному коду тестируемого модуля. Также драйвер на вход получает файл с описанием тестов, которые нужно выполнить. Описание теста включает в себя название функции, параметры, которые должны быть переданы в функцию и ожидаемый результат. Структура файла с тестами может быть произвольной. Результатом работы тестового драйвера должен быть отчёт о результатах тестирования. Отчёт должен содержать следующую информацию: номер теста, полученный результат, ожидаемый результат, результат прохождения теста (success или faile). Если результат прохождения теста faile, то нужно вывести причину ошибки. Драйвер тестирования должен быть максимально универсальным. Для языков программирования, позволяющих выполнять код в runtime нужно чтобы драйвер работал с разными модулями без перекомпиляции. Если это не возможно, то сделать так, чтобы для тестирования нового программного кода требовалось минимум изменений в коде драйвера тестирования (в идеале в отдельном заголовочном файле изменить путь к файлу и возможно список вызовов методов).