МОДУЛЬ 2 «МЕТОДЫ И ВИДЫ ТЕСТИРОВАНИЯ»

ТИПЫ ПРОЦЕССОВ ТЕСТИРОВАНИЯ

МОДУЛЬНОЕ ТЕСТИРОВАНИЕ. Модульному тестированию подвергаются небольшие модули (процедуры, классы и т.п.). Модульное тестирование обычно выполняется для каждого независимого программного модуля и является наиболее распространенным видом тестирования, особенно для систем малых и средних размеров.

ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ. Проверка корректности всех модулей не гарантирует корректности функционирования системы модулей, поэтому необходимо объединить их в систему и протестировать систему целиком. Интеграционное тестирование, когда система строится поэтапно, группы модулей добавляются постепенно.

СИСТЕМНОЕ ТЕСТИРОВАНИЕ. Полностью реализованный программный продукт подвергается системному тестированию. На данном этапе тестировщика интересует вся программа в целом, как ее видит конечный пользователь. Основой для тестов служат общие требования к программе, включая не только корректность реализации функций, но и производительность, время отклика, устойчивость к сбоям, атакам, ошибкам пользователя и т.д.

НАГРУЗОЧНОЕ ТЕСТИРОВАНИЕ. Нагрузочное тестирование позволяет получать прогнозируемые данные о производительности системы под нагрузкой, а также позволяет команде разработки, принимать более обоснованные решения, направленные на выработку оптимальных архитектурных композиций. Заказчик со своей стороны, получает возможность проводить приёмо-сдаточные испытания в условиях приближенных к реальным условиям.

ФОРМАЛЬНЫЕ ИНСПЕКЦИИ. Формальная инспекция является одним из способов верификации документов и программного кода, создаваемых в процессе разработки программного обеспечения. При формальной инспекции группа специалистов осуществляет проверку соответствия инспектируемых документов исходным документам. Для независимости проверки она осуществляется инспекторами, не участвовавшими в разработке инспектируемого документа.

ИЗДЕРЖКИ ТЕСТИРОВАНИЯ

На практике популярны следующие методы тестирования и отладки, упорядоченные по связанным с их применением затратам.

1. Статические методы тестирования.
2. Модульное тестирование.
3. Интеграционное тестирование.
4. Системное тестирование.
5. Тестирование реального окружения и реального времени.

Зависимость эффективности применения перечисленных методов или их способности к обнаружению соответствующих классов ошибок (С) сопоставлена на рис.8.3 [Котляров 2006] с затратами (В).

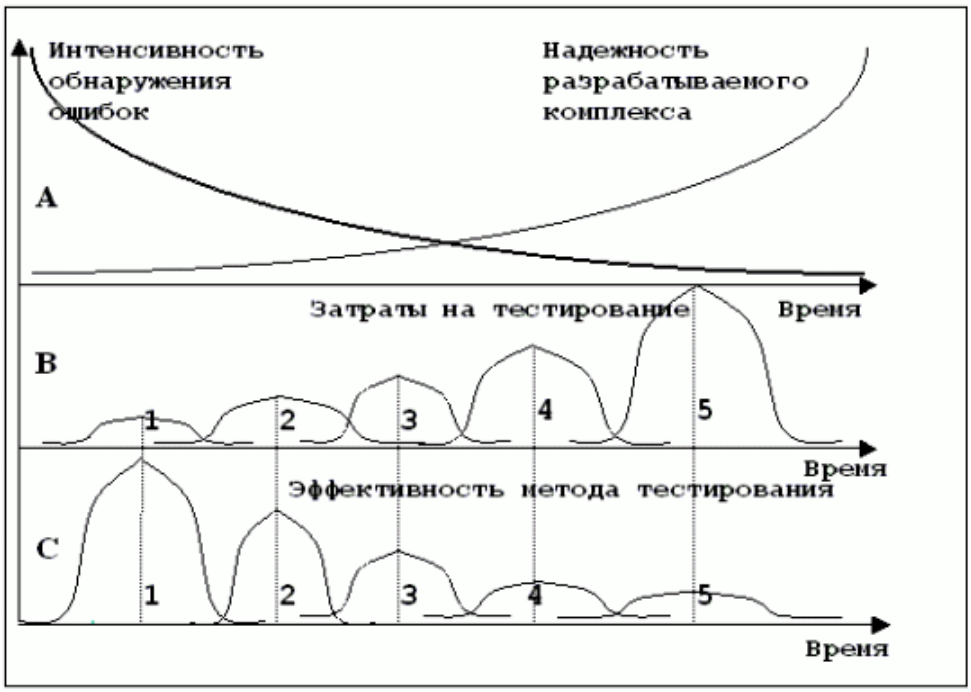

Рис. 8.3. Издержки тестирования.

В таблице 4.3 [Котляров 2006] даны характеристики модульного, интеграционного и системного тестирования и их сравнение между собой.

Таблица 4.3. Характеристики модульного, интеграционного и системного тестирования

	Модульное	**Интеграционное**	**Системное**
Типы дефектов	Локальные дефекты, такие как опечатки в реализации алгоритма, неверные операции, логические и математические выражения, циклы, ошибки в использовании локальных ресурсов, рекурсия и т.п.	Интерфейсные дефекты, такие как неверная трактовка параметров и их формат, неверное использование системных ресурсов и средств коммуникации, и т.п.	Отсутствующая или некорректная функциональность, неудобство использования, непредусмотренные данные и их комбинации, непредусмотренные или не поддержанные сценарии работы, ошибки совместимости, ошибки пользовательской документации, ошибки переносимости продукта на различные платформы, проблемы производительности, инсталляции и т.п.
Необходимость в системе тестирования	Да	Да	Нет (*)
Цена разработки системы тестирования	Низкая	Низкая до умеренной	Умеренная до высокой или неприемлемой
Цена процесса тестирования: разработки, прогона и анализа тестов	Низкая	Низкая	Высокая

(*) прямой необходимости в системе тестирования нет, но цена процесса системного тестирования часто настолько высока, что требует использования систем автоматизации, несмотря на возможно высокую их стоимость.

Поэтому все методы тестирования не только имеют право на существование, но и имеют свою нишу, где хорошо обнаруживают ошибки, тогда как вне ниши их эффективность падает. Поэтому необходимо совмещать различные методы и стратегии отладки и тестирования с целью обеспечения запланированного качества программного продукта при ограниченных затратах, что достижимо при использовании процесса управления качеством программного продукта.

СТАТИЧЕСКОЕ ТЕСТИРОВАНИЕ выявляет формальными методами анализа без выполнения тестируемой программы неверные конструкции или неверные отношения объектов программы (ошибки формального задания) с помощью специальных инструментов контроля кода – CodeChecker.

ДИНАМИЧЕСКОЕ ТЕСТИРОВАНИЕ (собственно тестирование) осуществляет выявление ошибок только на выполняющейся программе с помощью специальных инструментов автоматизации тестирования Testbed Testbench.

МЕТОДЫ ТЕСТИРОВАНИЯ

ЧЕРНЫЙ ЯЩИК

Идея тестирования системы, как черного ящика состоит в том, что все материалы, доступные тестировщику – требования на систему, описывающие ее поведение и сама система, работать с которой он может, только подавая на ее входы некоторые внешние воздействия и наблюдая на выходах некоторый результат. Все внутренние особенности реализации системы скрыты от тестировщика, то есть система и представляет собой “черный ящик”, правильность поведения которого по отношению к требованиям и предстоит проверить.

Основная задача тестировщика для данного метода тестирования состоит в последовательной проверке соответствия поведения системы требованиям. Кроме того, тестировщик должен проверить работу системы в критических ситуациях – что происходит в случае подачи неверных входных значений.

В результате тестирования обычно выявляется два типа проблем системы:

1. Несоответствие поведения системы требованиям.
2. Неадекватное поведение системы в ситуациях, не предусмотренных требованиями.

ПРОБЛЕМЫ ПЕРВОГО ТИПА обычно вызывают изменение программного кода, гораздо реже – изменение требований.

ПРОБЛЕМЫ ВТОРОГО ТИПА однозначно требуют изменения требований ввиду их неполноты – в требованиях явно пропущена ситуация, приводящая к неадекватному поведению системы.

Тестирование черного ящика называют также тестированием по требованиям, т.к. это единственный источник информации для построения тестовых планов.

СТЕКЛЯННЫЙ (БЕЛЫЙ) ЯЩИК

При тестировании системы, как стеклянного ящика, тестировщик имеет доступ не только к требованиям на систему, ее входам и выходам, но и к ее внутренней структуре – видит ее программный код.

Доступность программного кода расширяет возможности тестировщика тем, что он может видеть соответствие требований участкам программного кода и видеть тем самым – на весь ли программный код существуют требования. Программный код, для которого отсутствуют требования, называют кодом, непокрытым требованиями.

ТЕСТИРОВАНИЕ МОДЕЛЕЙ

Тестирование моделей находится несколько в стороне от классических методов верификации программного обеспечения. Прежде всего, это связано с тем, что объект тестирования – не сама система, а ее модель, спроектированная формальными средствами.

Работая с моделью можно создать такие ситуации, которые невозможно создать в тестовой лаборатории для реальной системы. Работая с моделью программного кода системы можно анализировать его свойства и такие параметры системы, как оптимальность алгоритмов или ее устойчивость.

АНАЛИЗ ПРОГРАММНОГО КОДА (ИНСПЕКЦИИ)

Во многих ситуациях тестирование поведения системы в целом невозможно – отдельные участки программного кода могут никогда не выполняться, при этом они будут покрыты требованиями. В этом случае выполняется ручной анализ программного кода на корректность, называемый также просмотрами или инспекциями кода. Если в результате инспекции выявляются проблемные участки, то информация об этом передается разработчикам для исправления наравне с результатами обычных тестов.

МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

Сложная программная система всегда состоит из отдельных частей – модулей, выполняющих ту или иную функцию в составе системы. Чтобы удостовериться в корректной работе системы в целом, необходимо вначале протестировать каждый модуль системы по отдельности. Такое тестирование модулей по отдельности получило называние модульного тестирования. Таким образом, модульное тестирование – это тестирование программы на уровне отдельно взятых модулей, функций или классов.

ЦЕЛЬ МОДУЛЬНОГО ТЕСТИРОВАНИЯ состоит в выявлении локализованных в модуле ошибок, в реализации алгоритмов, то есть проверке соответствия требованиям каждого отдельного модуля системы перед тем, как будет произведена его интеграция в состав системы.

В ходе модульного тестирования решаются четыре основные задачи.

ПЕРВАЯ ЗАДАЧА – разработка тестового окружения и тестовых примеров, а также выполнение тестов, протоколирование результатов выполнения и составление отчетов о проблемах.

ВТОРАЯ ЗАДАЧА – выявление проблемы в дизайне системы и нелогичных или запутанных механизмов работы с модулем.

ТРЕТЬЯ ЗАДАЧА – поддержка процесса изменения системы. Модульные тесты являются мощным инструментом проверки того, что новый вариант программного кода выполняет те же функции, что и старый.

ЧЕТВЕРТАЯ, ЗАДАЧА – создание подробных отчетов о проблемах, которые позволяют локализовать и устранить дефекты в программной системе на ранних стадиях ее разработки или разработки ее новой функциональности.

Модульное тестирование проводится по принципу "белого ящика", то есть основывается на знании внутренней структуры программы, и часто включает те или иные методы анализа покрытия кода.

Модульное тестирование обычно подразумевает создание вокруг каждого модуля определенной среды, включающей заглушки для всех интерфейсов тестируемого модуля. Некоторые из них могут использоваться для подачи входных значений, другие для анализа результатов, присутствие третьих может быть продиктовано требованиями, накладываемыми компилятором и сборщиком.

На уровне модульного тестирования проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов, типа работы с условиями и счетчиками циклов, а также с использованием локальных переменных и ресурсов.

ПОНЯТИЕ МОДУЛЯ И ЕГО ГРАНИЦ

МОДУЛЬ – это компонент минимального размера, который может быть независимо протестирован в ходе верификации программной системы. Обычно за тестируемый модуль принимается либо программный модуль (единица компиляции) в случае, если система разрабатывается на процедурном языке программирования или класс, если система разрабатывается на объектно-ориентированном языке.

В реальности часто возникают проблемы с тем, что считать модулем. Существует несколько подходов к данному вопросу:

- ✓ модуль – это часть программного кода, выполняющая одну функцию с точки зрения функциональных требований;
- ✓ модуль – это программный модуль, т.е. минимальный компилируемый элемент программной системы;
- ✓ модуль – это задача в списке задач проекта (с точки зрения его менеджера);
- ✓ модуль – это участок кода, который может уместиться на одном экране или одном листе бумаги;
- ✓ модуль – это один класс или их множество с единым интерфейсом.

Если система написана на процедурном языке, то процесс тестирования модуля происходит следующим образом. Для каждого модуля разрабатывается тестовый драйвер, вызывающий функции модуля и собирающий результаты их работы и набор заглушек, которые имитируют поведение функций, содержащихся в других модулях, не попадающих под тестирование данного модуля.

ПОДХОДЫ К МОДУЛЬНОМУ ТЕСТИРОВАНИЮ

ПЕРВЫЙ ПОДХОД к модульному тестированию основывается на предположении, что функциональность каждого вновь разработанного модуля должна проверяться в автономном режиме без его интеграции с системой.

ДОСТОИНСТВА – более простая локализация ошибок в модуле, поскольку при автономном тестировании исключается влияние остальных частей системы, которое может вызывать маскировку дефектов (эффект четного числа ошибок).

НЕДОСТАТКИ – повышенная трудоемкость написания драйверов и заглушек, поскольку заглушки должны адекватно моделировать поведение системы в различных ситуациях, а драйвер должен не только создавать тестовое окружение, но и имитировать внутреннее состояние системы, в составе которой должен функционировать модуль.

ВТОРОЙ ПОДХОД к модульному тестированию построен на предположении, что модуль работает в составе системы и если модули интегрировать в систему по одному, то можно протестировать поведение модуля в составе всей системы. Этот подход свойственен большинству современных “облегченных” методологий разработки, в том числе и XP.

ДОСТОИНСТВА – резко сокращаются трудозатраты на разработку заглушек и драйверов, т.к. в роли заглушек выступает уже оттестированная часть системы, а драйвер выполняет только функции передачи и приема данных, не моделируя внутреннее состояние системы.

НЕДОСТАТКИ – возрастает сложность написания тестовых примеров. Для приведения в нужное состояние системы заглушек, как правило, требуется только установить значения тестовых переменных, а для приведения в нужное состояние части реальной системы необходимо выполнить сценарий, приводящий в это состояние. Каждый тестовый пример в этом случае должен содержать такой сценарий. При этом подходе не всегда удается локализовать ошибки, скрытые внутри модуля и которые могут проявиться при интеграции следующих модулей.

ОРГАНИЗАЦИЯ МОДУЛЬНОГО ТЕСТИРОВАНИЯ

Модульное тестирование с точки зрения тестировщика – это комплекс работ по выявлению дефектов в тестируемых модулях. В эти работы включается анализ требований, разработка тестовых требований и тестовых планов, разработка тестового окружения, выполнение тестов, сбор информации об их прохождении.

Согласно стандарту IEEE 1008 процесс модульного тестирования состоит из трех фаз, в состав которых входит 8 видов деятельности (этапов).

1. Фаза планирования тестирования.
   1.1. Этап планирования основных подходов к тестированию, ресурсное планирование и календарное планирование.
   1.2. Этап определения свойств, подлежащих тестирования.
   1.3. Этап уточнения основного плана, сформированного на этапе 1.1.
2. Фаза получения набора тестов.
   2.1. Этап разработки набора тестов.
   2.2. Этап реализации уточненного плана.
3. Фаза измерений тестируемого модуля.
   3.1. Этап выполнения тестовых процедур.
   3.2. Этап определения достаточности тестирования.
   3.3. Этап оценки результатов тестирования и тестируемого модуля.

1.1. Основные задачи, решаемые в ходе этапа планирования, включают в себя:

- ✓ определение общего подхода к тестированию модулей;
- ✓ определение требований к полноте тестирования;
- ✓ определение требований к завершению тестирования;
- ✓ определение требований к ресурсам;
- ✓ определение общего плана-графика работ.

1.2. Основные задачи, решаемые в ходе деятельности по определению свойств системы, подлежащих тестированию, включают в себя:

- ✓ изучение функциональных требований;
- ✓ определение дополнительных требований и связанных процедур;
- ✓ определение состояний тестируемого модуля;
- ✓ определение характеристик входных и выходных данных;
- ✓ выбор элементов, подвергаемых тестированию.

1.3. В завершение фазы планирования производится уточнение основного плана – уточняется общий подход к тестированию, формулируются специальные и дополнительные требования к ресурсам, составляется детальный план-график работ.

2.1. Фаза разработки тестов начинается с разработки набора тестов, который будет использован для тестирования модуля. Основные документы, которые используются на этом этапе: функциональные требования к модулю, архитектура модуля, список элементов, подвергаемых тестированию, план-график работ, определения тестовых примеров от предыдущей версии модуля (если они существовали) и результаты тестирования прошлой версии (если они существовали).

2.2. В ходе этого этапа должны быть решены следующие задачи:

- ✓ разработка архитектуры тестового набора;
- ✓ разработка явных тестовых процедур (тестовых требований);
- ✓ разработка тестовых примеров;
- ✓ разработка тестовых примеров, основанных на архитектуре (в случае необходимости);
- ✓ составление спецификации тестовых примеров.

3.1. На следующем этапе проводится реализация тестов. В ходе этого этапа формируются тестовые наборы данных, которые используются в тестовых примерах, создается тестовое окружение, осуществляется интеграция тестового окружения с тестируемым модулем. В ходе этого этапа решаются две задачи: выполнение тестовых примеров и сбор и анализ результатов тестирования.

Сбору подлежит следующая информация:

- ✓ результат выполнения каждого тестового примера (прошел или не прошел);
- ✓ информация об информационном окружении системы в случае, если тест не прошел;
- ✓ информация о ресурсах, которые потребовались для выполнения тестового примера.

По результатам анализа этой информации составляются запросы на изменение проектной документации, программного кода тестируемого модуля или тестового окружения.

3.2. После прекращения тестирования выполняются работы по оценке проведенного тестирования, в ходе которых:

- ✓ описываются отличия реального процесса тестирования от запланированного;
- ✓ отличия поведения тестируемого модуля от описанного в требованиях (с целью дальнейшей коррекции требований);
- ✓ составляется общий отчет о прохождении тестов, включающий в себя и информацию о покрытии.

3.3. В завершение модульного тестирования необходимо проверить, что все созданные в его ходе артефакты – документы, программный код, файлы отчетов и данных – помещены в базу данных проекта, хранящую все данные, используемые и создаваемые в процессе разработки программной системы.

ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ

ЦЕЛИ И ЗАДАЧИ ИНТЕГРАЦИОННОГО ТЕСТИРОВАНИЯ

После тестирования отдельных модулей следующей задачей является тестирование корректности взаимодействия нескольких модулей, объединенных в единое целое. Такое тестирование называют интеграционным.

ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ – это тестирование части системы, состоящей из двух и более модулей. Интеграционное тестирование называют еще тестированием архитектуры системы.

ЦЕЛЬ ИНТЕГРАЦИОННОГО ТЕСТИРОВАНИЯ – удостовериться в корректности совместной работы компонент системы.

ОСНОВНАЯ ЗАДАЧА ИНТЕГРАЦИОННОГО ТЕСТИРОВАНИЯ - поиск дефектов, связанных с ошибками в реализации и интерпретации интерфейсного взаимодействия между модулями.

Интеграционное тестирование использует модель ”белого ящика” на модульном уровне.

Интеграционное тестирование применяется на этапе сборки модульно оттестированных модулей в единый комплекс.

Интеграционные тесты включают в себя проверки всех возможных видов взаимодействий между программными модулями и элементами, которые определяются в архитектуре системы – таким образом, интеграционные тесты проверяют полноту взаимодействий в тестируемой реализации системы.

ПРИМЕР ИНТЕГРАЦИОННОГО ТЕСТИРОВАНИЯ

Есть два модуля, один из которых накапливает сообщения протокола о принятых файлах, а второй выводит этот протокол на экран. В функциональных требованиях на систему записано, что сообщения должны выводиться в обратном хронологическом порядке. Однако модуль хранения сообщений сохраняет их в прямом порядке, а модуль вывода – использует стек для вывода в обратном порядке. Модульные тесты, затрагивающие каждый модуль по отдельности, не дадут здесь никакого эффекта – вполне реальна обратная ситуация, при которой сообщения хранятся в обратном порядке, а выводятся с использованием очереди. Обнаружить потенциальную проблему можно только, если проверить взаимодействие модулей при помощи интеграционных тестов. Ключевым моментом здесь является то, что в обратном хронологическом порядке сообщения выводит система в целом, т.е. проверив модуль вывода и обнаружив, что он выводит сообщения в прямом порядке, мы не сможем гарантировать того, что мы обнаружили дефект.

СТРУКТУРНАЯ КЛАССИФИКАЦИЯ МЕТОДОВ ИНТЕГРАЦИОННОГО ТЕСТИРОВАНИЯ

В рамках структурной классификации существует несколько методов проведения интеграционного тестирования:

- ✓ Восходящее тестирование.
- ✓ Монолитное тестирование.
- ✓ Нисходящее тестирование.

ВОСХОДЯЩЕЕ ТЕСТИРОВАНИЕ предполагает, что сначала тестируются все программные модули, входящие в состав системы и только затем они объединяются для интеграционного тестирования. При таком подходе значительно упрощается локализация ошибок: если модули протестированы по отдельности, то ошибка при их совместной работе есть проблема их интерфейса.

Восходящий метод имеет существенный недостаток – необходимость в разработке драйвера и заглушек для модульного тестирования перед проведением интеграционного тестирования и необходимость в разработке драйвера и заглушек при интеграционном тестировании части модулей системы (Рис.23) [Синицын 2006].

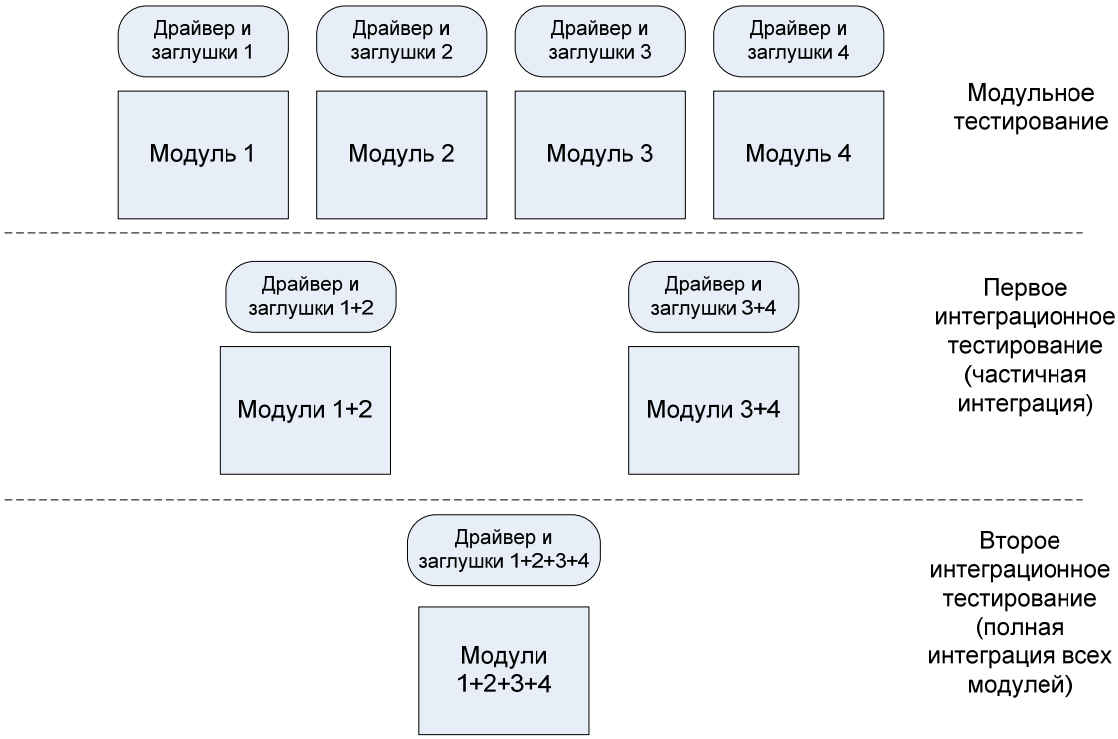

Рис.23 Разработка драйверов и заглушек при восходящем интеграционном тестировании

Отметим, что с одной стороны драйверы и заглушки – мощный инструмент тестирования, а с другой стороны – их разработка требует значительных ресурсов, особенно при изменении состава интегрируемых модулей.

МОНОЛИТНОЕ ТЕСТИРОВАНИЕ предполагает, что отдельные компоненты системы серьезного тестирования не проходили.

Основное преимущество данного метода – отсутствие необходимости в разработке тестового окружения, драйверов и заглушек. После разработки всех модулей выполняется их интеграция, после чего система проверяется вся в целом, как она есть. Основная задача этого тестирования – определить проблемы взаимодействия отдельных модулей системы.

Монолитное тестирование имеет ряд серьезных недостатков:

- ✓ Очень трудно выявить источник ошибки, поэтому в большинстве модулей следует предполагать наличие ошибки.
- ✓ Трудно организовать исправление ошибок. В результате тестирования фиксируется найденная проблема. Однако, тестируемые модули написаны разными людьми, возникает проблема – кто из них является ответственным за поиск устранение дефекта?
- ✓ Процесс тестирования плохо автоматизируется. Преимущество оборачивается недостатком. Каждое внесённое изменение требует повторения всех тестов.

НИСХОДЯЩЕЕ ТЕСТИРОВАНИЕ предполагает, что процесс интеграционного тестирования движется следом за разработкой. Сначала при нисходящем подходе тестируют только самый верхний управляющий уровень системы, без модулей более низкого уровня. Затем постепенно с более высокоуровневыми модулями интегрируются более низкоуровневые модули. В результате отпадает необходимость в драйверах (роль драйвера выполняет более высокоуровневый модуль системы), однако сохраняется нужда в заглушках (Рис.24) [Синицын 2006].

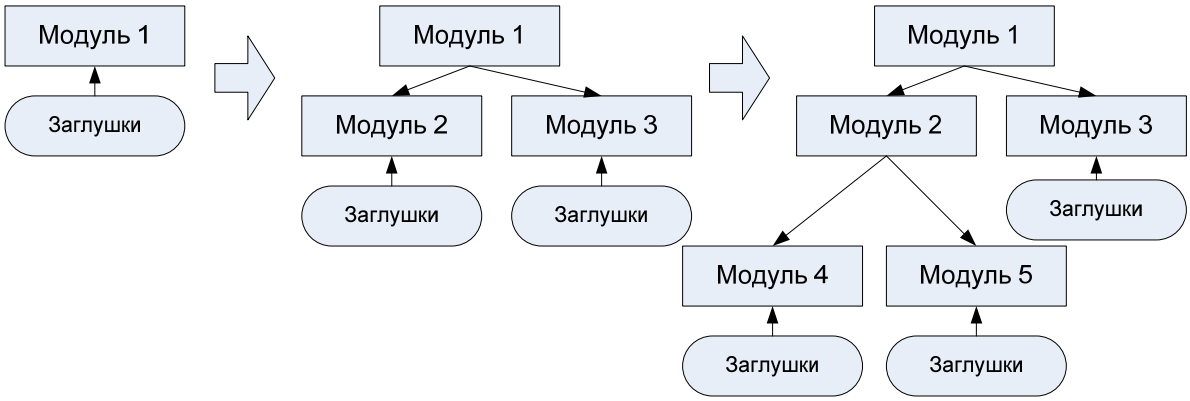

Рис.24 Постепенная интеграция модулей при нисходящем методе тестирования

ВРЕМЕННАЯ КЛАССИФИКАЦИЯ МЕТОДОВ ИНТЕГРАЦИОННОГО ТЕСТИРОВАНИЯ

В рамках временной классификации существует несколько методов проведения интеграционного тестирования:

- ✓ тестирование с поздней интеграцией;
- ✓ тестирование с постоянной интеграцией;
- ✓ тестирование с регулярной или послойной интеграцией.

ТЕСТИРОВАНИЕ С ПОЗДНЕЙ ИНТЕГРАЦИЕЙ – практически полный аналог монолитного тестирования. Интеграционное тестирование при такой схеме откладывается на как можно более поздние сроки проекта. Этот подход оправдывает себя в том случае, если система представляет собой конгломерат слабо связанных между собой модулей, взаимодействующих по какому-либо стандартному интерфейсу, определенному вне проекта.

ТЕСТИРОВАНИЕ С ПОСТОЯННОЙ ИНТЕГРАЦИЕЙ подразумевает, что, как только разрабатывается новый модуль системы, он сразу же интегрируется со всей остальной системой. При этом тесты для этого модуля проверяют как сугубо его внутреннюю функциональность, так и его взаимодействие с остальными модулями системы. Таким образом, этот подход совмещает в себе модульное тестирование и интеграционное. Разработки заглушек при таком подходе не требуется, но может потребоваться разработка драйверов.

ТЕСТИРОВАНИЕ С РЕГУЛЯРНОЙ ИЛИ ПОСЛОЙНОЙ ИНТЕГРАЦИЕЙ – интеграционному тестированию подлежат сильно связанные между собой группы модулей (слои), которые затем также интегрируются между собой. Такой вид интеграционного тестирования называют также иерархическим интеграционным тестированием, поскольку укрупнение интегрированных частей системы, как правило, происходит по иерархическому принципу. Однако, в отличие от нисходящего или восходящего тестирования, направление прохода по иерархии в этом подходе не задано.

В Таблице 6 [Синицын 2006] даны характеристики видов интеграционного тестирования. Время интеграции характеризует момент, когда проводится первое интеграционное тестирование и все последующие. Частота интеграции – насколько часто при разработке выполняется интеграция. Необходимость в драйверах и заглушках определена в последних двух строках таблицы.

Таблица 6. Основные характеристики различных видов интеграционного тестирования

Вид интеграции Свойство	Восходящее	Нисходящее	Монолитное	Поздняя интеграция	Постоянная интеграция	Регулярная интеграция
Время интеграции	поздно (после тестирования модулей)	рано (параллельно с разработкой)	поздно (после разработки всех модулей)	поздно (после разработки всех модулей)	рано (параллельно с разработкой)	рано (параллельно с разработкой)
Частота интеграции	редко	часто	редко	редко	часто	часто
Нужны ли драйверы	да	нет	нет	нет	да	да
Нужны ли заглушки	да	да	нет	нет	нет	да

ПЛАНИРОВАНИЕ ИНТЕГРАЦИОННОГО ТЕСТИРОВАНИЯ

На этапе планирования разрабатывается концепция и стратегия интеграции – документ, который описывает общий подход к определению последовательности, в которой должны интегрироваться модули. Как правило, концепция основывается на одном из видов интеграции, рассмотренных выше (например, на нисходящей), но учитывает особенности конкретной системы (например, вначале должны интегрироваться компоненты работы с базой данных, затем пользовательского интерфейса, затем интерфейсные компоненты и компоненты работы с БД интегрируются вместе).

Составляется интеграционный тест-план, например, кластерного типа, в котором для каждого кластера из интегрированных модулей определяется следующее:

- ✓ кластеры, от которых зависит данный кластер;
- ✓ кластеры, которые должны быть протестированы до тестирования данного кластера;
- ✓ описание функциональности тестируемого кластера;
- ✓ список модулей в кластере;
- ✓ описание тестовых примеров для проверки кластера.

СИСТЕМНОЕ ТЕСТИРОВАНИЕ

ЦЕЛИ И ЗАДАЧИ СИСТЕМНОГО ТЕСТИРОВАНИЯ

После завершения интеграционного тестирования переходят к тестированию системы в целом, как единого объекта тестирования – к системному тестированию. Системное тестирование рассматривает тестируемую систему в целом и оперирует на уровне пользовательских интерфейсов. На этом уровне интересуются поведенческими аспектами системы. Как правило, для системного тестирования используется подход черного ящика, при этом в качестве входных и выходных данных используются реальные данные, с которыми работает система, или данные подобные им.

ОСНОВНАЯ ЗАДАЧА СИСТЕМНОГО ТЕСТИРОВАНИЯ - выявление дефектов, связанных с работой системы в целом, таких как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство в применении и т.д.

СИСТЕМНОЕ ТЕСТИРОВАНИЕ – один из самых сложных видов тестирования. На этом этапе проводится не только функциональное тестирование, но и оценка характеристик качества системы – ее устойчивости,

надежности, безопасности и производительности. На этом этапе выявляются многие проблемы внешних интерфейсов системы, связанные с неверным взаимодействием с другими системами, аппаратным обеспечением, неверным распределением памяти, отсутствием корректного освобождения ресурсов и т.п.

ВИДЫ СИСТЕМНОГО ТЕСТИРОВАНИЯ

Принято выделять следующие виды системного тестирования:

- ✓ Функциональное тестирование;
- ✓ Тестирование производительности;
- ✓ Нагрузочное или стрессовое тестирование;
- ✓ Тестирование конфигурации;
- ✓ Тестирование безопасности;
- ✓ Тестирование надежности и восстановления после сбоев;
- ✓ Тестирование удобства использования.

Исходной информацией для проведения перечисленных видов тестирования являются два класса требований: функциональные и нефункциональные. Функциональные требования явно описывают, что система должна делать и какие выполнять преобразования входных значений в выходные. Нефункциональные требования определяют свойства системы, напрямую не связанные с ее функциональностью. Примером таких свойств может служить время отклика на запрос пользователя, время бесперебойной работы, количество ошибок, которые допускает начинающий пользователь за первую неделю работы и т.п.

ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ – предназначено для доказательства того, что вся система в целом ведет себя в соответствии с ожиданиями пользователя, формализованными в виде системных требований. При тестировании проверяются все функции системы с точки зрения ее пользователей. Система рассматривается как черный ящик, поэтому в данном случае полезно использовать классы эквивалентности. Критерием полноты тестирования является полнота покрытия тестами системных функциональных требований и полнота тестирования классов эквивалентности, а именно:

- ✓ все функциональные требования должны быть протестированы;
- ✓ все классы допустимых входных данных должны корректно обрабатываться системой;
- ✓ все классы недопустимых входных данных должны быть отброшены системой, при этом не должна нарушаться стабильность ее работы;
- ✓ в тестовых примерах должны генерироваться все возможные классы выходных данных системы;
- ✓ во время тестирования система должна побывать во всех своих внутренних состояниях, пройдя при этом по всем возможным переходам между состояниями.

ТЕСТИРОВАНИЕ ПРОИЗВОДИТЕЛЬНОСТИ – направлено на определение того, что система обеспечивает должный уровень производительности при обработке пользовательских запросов. Тестирование производительности выполняется при различных уровнях нагрузки на систему, на различных конфигурациях оборудования. Выделяют три основных фактора, влияющие на производительность системы: количество поддерживаемых системой потоков, количество свободных системных ресурсов, количество свободных аппаратных ресурсов.

Тестирование производительности позволяет выявлять узкие места в системе, которые проявляются в условиях повышенной нагрузки или нехватки системных ресурсов.

Все требования, относящиеся к производительности системы, должны быть четко определены и, обязательно, должны включать в себя числовые оценки параметров производительности.

СТРЕССОВОЕ ТЕСТИРОВАНИЕ. Его основная задача – оценить производительность и устойчивость системы в случае, когда для своей работы она выделяет максимально доступное количество ресурсов, либо когда она работает в условиях их критической нехватки. Основная цель стрессового тестирования – вывести систему из строя, определить те условия, при которых она не сможет далее нормально функционировать.

Стрессовое тестирование очень важно при тестировании web-систем и систем с открытым доступом, уровень нагрузки на которые зачастую очень сложно прогнозировать.

ТЕСТИРОВАНИЕ КОНФИГУРАЦИИ. Программные системы массового назначения предназначены для использования на самом разном оборудовании. Хотя особенности реализации периферийных устройств скрываются драйверами операционных систем, которые имеют унифицированный интерфейс, с точки зрения прикладных систем, проблемы совместимости (как программной, так и аппаратной) все равно существуют.

В ходе тестирования конфигурации проверяется, что система продолжает стабильно работать при горячей замене любого поддерживаемого устройства на аналогичное устройство. Также необходимо проверять, что система корректно обрабатывает проблемы, возникающие в оборудовании, как штатные (например, сигнал конца бумаги в принтере), так и нештатные (сбой по питанию).

ТЕСТИРОВАНИЕ БЕЗОПАСНОСТИ. Если программная система предназначена для хранения или обработки данных, содержимое которых представляет собой тайну определенного рода (личную, коммерческую, государственную и т.п.), то к свойствам системы, обеспечивающим сохранение этой тайны, будут предъявляться повышенные требования. Эти требования должны быть проверены при тестировании безопасности системы. В ходе этого тестирования проверяется, что информация не теряется, не повреждается, ее невозможно подменить, а также к ней невозможно получить несанкционированный доступ, в том числе при помощи использования уязвимости в самой программной системе.

В отечественной практике принято проводить сертификацию программных систем, предназначенных для хранения данных для служебного пользования, секретных, совершенно секретных и совершенно секретных особой важности.

ТЕСТИРОВАНИЕ НАДЕЖНОСТИ И ВОССТАНОВЛЕНИЯ ПОСЛЕ СБОЕВ. Для корректной работы системы в любой ситуации необходимо удостовериться в том, что она восстанавливает свою функциональность и продолжает корректно работать после любой проблемы, прервавшей ее работу. При тестировании восстановления после сбоев имитируются сбои оборудования или окружающего программного обеспечения, либо сбои программной системы, вызванные внешними факторами. При анализе поведения системы в этом случае необходимо обращать внимание на два фактора – минимизацию потерь данных в результате сбоя и минимизацию времени между сбоем и продолжением нормального функционирования системы.

ТЕСТИРОВАНИЕ УДОБСТВА ИСПОЛЬЗОВАНИЯ. Отдельная группа нефункциональных требований – требования к удобству использования пользовательского интерфейса системы.

ЛИТЕРАТУРА

[Котляров 2006] – В.П. Котляров, Т.В. Коликова. Основы тестирования программного обеспечения. Учебное пособие. М.: Интернет-Университет Информационных технологий.

[Синицын 2006] – Синицын С.В., Налютин Н.Ю. Верификация программного обеспечения. Курс лекций. Московский инженерно-физический институт. М. 2006.