МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
Белорусский национальный технический университет

Кафедра «Программное обеспечение информационных систем
и технологий»

Ю. Б. Попова

ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Пособие

*Рекомендовано УМО по образованию
в области информатики и радиоэлектроники
для направления специальности
1-40 05 01-04 «Информационные системы и технологии
(в обработке и представлении информации)»*

Минск
БНТУ
2020

УДК 004.415.53
ББК 32.973-018.2
П58

Рецензенты:
кафедра технологий программирования
Белорусского государственного университета;
канд. техн. наук, зам. директора
ОДО «ВирусБлокАда» *Г. К. Резников*

Попова, Ю. Б.
П58 Тестирование и отладка программного обеспечения: пособие / Ю. Б. Попова. – Минск : БНТУ, 2020. – 66 с.
ISBN 978-985-583-056-7.

В пособии рассмотрены основные понятия из области тестирования программного обеспечения, его классификация, виды, роль и место в жизненном цикле разработки программ. Описаны процессы модульного и интеграционного тестирования, их этапы, возможности автоматизации, особенности работы с инструментами автоматизации. Большое внимание уделено планированию и реализации функционального тестирования, процессу поиска и документирования ошибок. Рассмотрен процесс отладки программного обеспечения, основные этапы, методы и рекомендации по его проведению.

УДК 004.415.53
ББК 32.973-018.2

ISBN 978-985-583-056-7

© Попова Ю. Б., 2020
© Белорусский национальный технический университет, 2020

СОДЕРЖАНИЕ

1. РОЛЬ И МЕСТО ТЕСТИРОВАНИЯ В ЖИЗНЕННОМ ЦИКЛЕ РАЗРАБОТКИ ПРОГРАММ 5

1.1. Определение термина «тестирование программного обеспечения» 5
1.2. Тестирование в жизненном цикле разработки программного обеспечения 6
1.3. Виды тестирования программного обеспечения 8
1.4. Смежные вопросы тестирования 10

2. РАЗРАБОТКА ТРЕБОВАНИЙ К ПРОГРАММНОМУ ПРОДУКТУ 11

2.1. Этапы разработки требований к программному продукту 11
2.2. Категории требований к программному продукту 13

3. МОДУЛЬНОЕ ТЕСТИРОВАНИЕ 14

3.1. Модульное тестирование и его задачи 14
3.2. Обзоры программного кода 16
3.3. Принципы тестирования структуры программных модулей 17
3.4. Способы тестирования взаимодействия модулей 19
3.5. Стратегии выполнения пошагового тестирования 21
3.6. Особенности объектно-ориентированного тестирования 22
3.7. Автоматизация модульного тестирования 25
3.7.1. Семейство xUnit 25
3.7.2. Встроенные инструменты для автоматизации модульного тестирования 28
3.7.3. Использование универсальных инструментов автоматизации тестирования 29

4. ПЛАНИРОВАНИЕ ФУНКЦИОНАЛЬНОГО ТЕСТИРОВАНИЯ 31

4.1. Тестовый план 32
4.2. Разработка тестовых случаев 33
4.3. Эквивалентирование и анализ граничных значений 36

5. РЕАЛИЗАЦИЯ ФУНКЦИОНАЛЬНОГО ТЕСТИРОВАНИЯ 37

5.1. Ошибка, свойства ошибки 37
5.2. Правила составления отчетов об ошибках 39
5.3. Системы документирования и отслеживания ошибок 41
5.4. Жизненный цикл ошибки 42
5.5. Реализация градации тестов 43
    5.5.1. Тест «на дым» и критерии его непрохождения 43
    5.5.2. Позитивный тест 44
    5.5.3. Негативный тест 44
5.6. Особенности тестирования standalone и веб-приложений 47

6. АВТОМАТИЗАЦИЯ ФУНКЦИОНАЛЬНОГО ТЕСТИРОВАНИЯ 48

6.1. Достоинства и недостатки автоматизации функционального тестирования 50
6.2. Требования к автоматизированным тестам 52
6.3. Методы автоматизации функционального тестирования 52
    6.3.1. Метод Play&Record 52
    6.3.2. Метод функциональной декомпозиции 53
    6.3.3. Метод Data-driven 54
    6.3.4. Метод Keyword-driven 54
6.4. Семейство Selenium и его возможности 55
6.5. Проблемы внедрения автоматизации тестирования программного обеспечения 56

7. ОТЛАДКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 58

7.1. Методика отладки 58
7.2. Методы отладки 60
7.3. Психологические аспекты отладки 62
7.4. Средства отладки 63

ЛИТЕРАТУРА 65

1. РОЛЬ И МЕСТО ТЕСТИРОВАНИЯ В ЖИЗНЕННОМ ЦИКЛЕ РАЗРАБОТКИ ПРОГРАММ

1.1. Определение термина «тестирование программного обеспечения»

Тестирование программ зародилось практически одновременно с программированием. Машинное время стоило дорого, поэтому предприятия с повышенными требованиями к надежности программ (например, авиакосмическая промышленность) стали активно разрабатывать методики тестирования.

Долгое время было принято считать, что целью тестирования является доказательство отсутствия ошибок в программе. Однако этот тезис не выдерживает критики, т. к. полный перебор всех возможных вариантов выполнения программы находится за пределами вычислительных возможностей даже для очень небольших программ. Поэтому никакое тестирование не может гарантировать отсутствия ошибок.

Со временем понимание целей тестирования изменилось, и один из основоположников тестирования Гленфорд Майерс предложил следующее определение: «*Тестирование* – это процесс выполнения программ с целью обнаружения ошибок» [1]. Здесь следует отметить, что тестовая деятельность, предусматривающая эксплуатацию программного продукта, называется *динамическим тестированием* (англ., Dynamic Testing). В то же время существует понятие *статического тестирования* (англ., Static Testing), определяющего тестовую деятельность, связанную с анализом программного обеспечения (ПО) и проводимую без его запуска. К статическому тестированию относятся обзоры кода, инспекции, аудит, критический анализ и т. д. Внимательное изучение этих двух подходов к тестированию показывает, что они дополняют друг друга и позволяют находить разные виды ошибок. Поэтому наиболее эффективные процессы разработки ПО используют комбинацию методов тестирования, реализующих эти подходы. В табл. 1.1 приводятся показатели минимальной, средней и максимальной эффективности различных методов тестирования [2].

Из таблицы видно, что тестирование максимально эффективно в тех случаях, когда программа проверяется не только путем запуска,

но и путем чтения, статических проверок и т. п. Поэтому классическое определение Майерса, приведенное выше, оказывается слишком узким и не охватывающим всех аспектов современного тестирования. Поэтому будем пользоваться следующим определением: *Тестирование ПО* (англ., Software Testing) – это процесс анализа или эксплуатации программного обеспечения с целью выявления дефектов. Слово «процесс» используется для уточнения, что тестирование – это плановая и упорядоченная деятельность.

Таблица 1.1

Показатели эффективности различных методов тестирования [2]

Название метода	**Минимальная эффективность**	**Средняя эффективность**	**Максимальная эффективность**
Персональные просмотры проектных документов	15 %	35 %	70 %
Неформальные групповые просмотры	30 %	40 %	60 %
Формальные просмотры проектных документов	35 %	55 %	75 %
Формальные инспекции кода	30 %	60 %	70 %
Моделирование и прототипирование	35 %	65 %	80 %
Проверка за партой	20 %	40 %	60 %
Тестирование модулей	10 %	25 %	50 %
Функциональное тестирование	20 %	35 %	55 %
Комплексное тестирование	25 %	45 %	60 %
Тестирование в реальных условиях	35 %	50 %	65 %
Применение всех перечисленных методов тестирования	**93 %**	**99 %**	**99 %**

1.2. Тестирование в жизненном цикле разработки программного обеспечения

До начала 80-х годов процесс тестирования программного обеспечения был разделен с процессом разработки: вначале программисты реализовывали заданную функциональность, а затем тестиров-

щики приступали к проверке качества созданных программ. Такая модель жизненного цикла разработки ПО называется каскадной (или водопадной) и состоит из 5 основных этапов: разработка требований к программе, проектирование, реализация, тестирование и сопровождение. Однако описанный выше подход создает множество проблем. Например, разработка программ может оказаться достаточно длительной (скажем, несколько месяцев), чем тогда в это время должны заниматься тестировщики? Другая, более серьезная проблема заключается в плохой предсказуемости результатов такого процесса разработки. Ключевым вопросом здесь может быть: какое количество времени потребуется на завершение продукта, в котором существует 500 известных ошибок? На самом деле, предугадать это совершенно невозможно, так как разные ошибки будут требовать разного количества времени на исправление, а исправление известных ошибок будет неизбежно связано с внесением новых. Существует следующая мрачная статистика: даже однострочное изменение в программе с вероятностью 55 % либо не исправляет старую ошибку, либо вносит новую. Если же учитывать изменения любого объема, то в среднем менее 20 % изменений корректны с первого раза.

В связи с этим, в 90-х годах появилась другая методика разработки, которую вслед за компанией Microsoft называют *zero-defect mindset*. Основная идея этой методики заключается в том, что качество программ проверяется не *post factum*, а постоянно в процессе разработки. Например, программист не может перейти к разработке новой функциональности, если существуют известные ошибки высокого приоритета в частях, разработанных им ранее. Так появились шарнирно-каскадная (или V-образная), а также спиралевидная модели разработки ПО [2].

При такой постановке вопроса тестирование становится центральной частью каждого этапа жизненного цикла разработки программ: тестированию подвергаются требования к программному продукту, алгоритмы, исходные коды, модули, программные сборки, функциональность программ и т. д. Практика показывает, что, чем раньше найдена ошибка, тем дешевле ее исправить. Частым примером в литературе является следующий: стоимость незамеченной ошибки в документе требований, которую можно оценить в 2 $, вырастает в 200 $ на этапе сопровождения, поскольку были затрачены силы и время на все предыдущие этапы.

1.3. Виды тестирования программного обеспечения

Долгое время основным способом тестирования программного обеспечения было тестирование *методом «черного ящика»* – программе подавались некоторые данные на вход и проверялись результаты в надежде найти несоответствия. При этом, как именно работает программа, считается несущественным. Этот подход до сих пор является самым распространенным в повседневной практике. Наряду с этим подходом существуют методы тестирования, которые изучают внутреннее устройство программы (исходные тексты, модули и их структуры). Их обобщенно называют *тестированием «белого ящика»*. Также существует тестирование по *методу «серого ящика»*, когда имеется частичный доступ к исходному коду (например, сторонние компоненты или автоматизация функционального тестирования через пользовательский интерфейс).

В настоящее время в литературе приведена обширная классификация видов тестирования ПО [3, 4, 5]. Рассмотрим одну из них по следующим признакам:

1) По знанию внутреннего устройства программы:

– тестирование по методу «черного ящика»;

– тестирование по методу «белого ящика»;

– тестирование по методу «серого ящика».

2) По объекту тестирования:

– тестирование требований к программному продукту;

– тестирование исходного кода (например, обзоры кода);

– модульное тестирование (проверка отдельных программных модулей);

– интеграционное тестирование (проверка взаимодействия программных модулей);

– объектно-ориентированное тестирование (тестирование классов);

– функциональное тестирование (проверка работы заявленной функциональности);

– системное тестирование (проверка работы программы в реальном системном окружении);

– тестирование интерфейса программы;

– тестирование удобства использования;

– локализационное тестирование (проверка работы программы при переходах на другие иностранные языки);

– тестирование производительности;

– тестирование безопасности;

– тестирование на совместимость (или кроссплатформенное и кроссбраузерное тестирование – проверка работы программы на разных операционных системах и в разных браузерах).

3) По субъекту тестирования:

– тестирование, проводимое программистом (например, обзоры кода, модульное тестирование);

– тестирование, проводимое тестировщиком (например, функциональное тестирование, тестирование производительности);

– случайное тестирование (англ., ad hoc testing, проводится не участником проекта без предварительной подготовки с целью найти случайные ошибки в программе);

– приемочные испытания (проводятся, как правило, заказчиком).

4) По позитивности тестов:

– позитивное тестирование;

– негативное тестирование.

5) По степени изолированности компонент:

– модульное тестирование;

– интеграционное тестирование;

– системное тестирование.

6) По степени автоматизации тестирования:

– ручное тестирование (проводится человеком);

– автоматизированное тестирование (полностью проводится компьютером);

– полуавтоматизированное тестирование (проводится и компьютером, и человеком).

7) По степени подготовки к тестированию:

– тестирование по тестовым случаям;

– случайное тестирование.

8) По запуску программы на выполнение:

– статическое тестирование;

– динамическое тестирование;

9) По хронологии тестирования:

– до передачи пользователю (альфа-тестирование, тест приемки, тестирование новых функциональностей, регрессионное тестирование);

– после передачи пользователю (бета-тестирование).

Альфа-тестирование – это тестирование ПО, когда процесс разработки приближается к завершению. В результате проведения такого тестирования в проект могут вноситься незначительные изменения.

Бета-тестирование означает тестирование, при котором разработка и тестирование по существу завершены, и до окончательного выпуска продукта необходимо обнаружить оставшиеся ошибки и проблемы.

1.4. Смежные вопросы тестирования

Об экономической стороне тестирования. Тестирование является затратной деятельностью, отнимающей время и деньги. Поэтому в большинстве случаев разработчики программного обеспечения заранее формулируют какой-либо критерий качества создаваемых программ (определяют так называемую *планку качества*), добиваются выполнения этого критерия и затем прекращают тестирование, выпуская продукт на рынок. Такая концепция получила название *Good Enough Quality* (достаточно хорошее ПО), в противовес более очевидной концепции *Best Possible Quality* (максимально качественное ПО).

К сожалению, принцип *Good Enough Quality* зачастую понимают неправильно, ближе к формулировке *Quality – If Time Permits* (качество, если будет время). Конечно, выпуск плохо протестированного программного продукта из-за недостатка времени – это плохая практика. Опыт показывает, что пользователи склонны со временем забывать даже значительные задержки с выпуском продукта, но плохое качество выпущенного продукта запоминается на всю жизнь. На самом деле, *Good Enough Quality* – это просто поиск разумного компромисса между затратами на тестирование, длительностью разработки продукта и его качеством.

Психологические аспекты тестирования. Тестирование принципиально отличается от программирования по своим психологическим характеристикам. Дело в том, что программирование носит конструктивный характер, а тестирование деструктивно по своей природе. Можно сказать, что программист создает, строит, а тестировщик ищет недостатки в этих строениях. Поэтому программисты так часто не замечают очевидных ошибок в своих программах: к своему творчеству трудно относиться критично.

Все это не значит, что тестирование «хуже», чем программирование, или, что в процессе тестирования нет места творчеству, – просто эти виды деятельности отличаются коренным образом, и для тестирования требуется иной склад характера, чем для программирования. Следовательно, программист не должен тестировать свои собственные программы – для этого необходим другой человек. Более того, программист не должен тестировать даже чужие программы! Дело в том, что программист потратит какое-то время, перенастраиваясь на новую задачу. Даже переключение с одной программистской задачи на другую требует обычно от 10 минут до получаса. Поэтому переключение с одного образа мышления на другой отнимет существенно больше времени.

Таким образом, принято считать, что команда разработчиков не должна совпадать с командой инженеров тестирования, но при этом разработчики и тестировщики должны постоянно взаимодействовать друг с другом для достижения приемлемого качества окончательного продукта.

2. РАЗРАБОТКА ТРЕБОВАНИЙ К ПРОГРАММНОМУ ПРОДУКТУ

Разработка любого программного продукта начинается с определения *требований* (англ., Requirements) к нему. Проводятся встречи, интервью с заказчиком, в результате чего составляется документ, в котором отражены все требования к продукту. Там описываются как функциональные (что должна выполнять программа), так и нефункциональные (например, на каком оборудовании должна работать программа) требования.

2.1. Этапы разработки требований к программному продукту

Процесс разработки требований состоит из следующих этапов [6]:

1) Опросы заказчика – проводятся посредством интервью в виде вопросов и ответов. Используются слайды, макеты, чертежи, аналогичные проекты, обсуждаются пожелания заказчика. В этот момент желательно присутствие тестировщика с целью уточнения способов применения программного продукта, перечня наиболее и наименее

используемой функциональности. Такая информация позволит рационально распределить время, отведенное на тестирование, и уделить повышенное внимание самой важной функциональности.

2) Подготовка документа требований. По мере получения сведений от заказчика, требования фиксируются в виде документа (иногда его также называют документом определения требований), который содержит список всех требований на разговорном естественном языке. В этом документе должны быть отражены следующие свойства требований:

– каждое требование должно иметь уникальный идентификатор;

– требования должны быть представлены с точки зрения пользователей системы и не затрагивать внутренние свойства системы и детализацию программного кода;

– в перечень должны быть включены как функциональные, так и нефункциональные требования;

– документ требований должен храниться в безопасном месте и находиться под управлением конфигурациями.

3) Разработка спецификации требований. Спецификация требований или спецификация к программному продукту (англ., Software Specification) описывает то же самое, что и документ требований, но предназначен для разработчиков, поэтому содержит уточненные данные и нужные технические детали. Например: используемые библиотеки, классы, БД, типы данных, элементы окон, объекты интерфейса.

4) Построение матрицы прослеживаемости требований (англ., Requirements Traceability Matrix). Назначение этой матрицы – поставить в соответствие каждому требованию его реализацию на всех этапах разработки программы, т. е. компоненты проекта, программного кода и тестовые случаи. Такой подход позволяет не потерять ни одно требование ни на каком этапе разработки ПО.

5) Анализ и тестирование требований. После окончания разработки требований приступают к их проверке по следующим критериям:

a) полнота – набор требований считается полным, если все его составные части представлены, и каждая часть выполнена в полном объеме. Требования не должны содержать выражений: и так далее, и тому подобное, и прочее, подлежит утонению, а также не должны ссылаться на несуществующую информацию;

b) однозначность, т. е. требование должно допускать единственное толкование. Должно быть удобочитаемым и понятным;

c) непротиворечивость, т. е. требования не должны противоречить друг другу и существующим стандартам. В случае необходимости можно вводить систему приоритетов;

d) прослеживаемость – каждое требование должно иметь уникальный идентификатор, который позволит проследить его на протяжении всего жизненного цикла разработки программы;

e) осуществимость – каждое требование должно ставить перед системой реально осуществимые задачи как с функциональной точки зрения, так и в смысле затрат времени и средств на разработку;

f) контролепригодность – каждое требование должно быть измеряемым, чтобы тестирование могло выполняться в приемлемых условиях.

2.2. Категории требований к программному продукту

Выделяют следующие категории требований к программному продукту [6]:

1) Функциональные средства. Требования этой категории определяют, какие функции должен выполнять данный программный продукт на системном и пользовательском уровне. Для ясности может быть указано, что не должен выполнять программный продукт.

2) Интерфейсы. Эта категория требований описывает входы, получаемые из внешних систем, и выходы, направляемые во внешние системы, а также указывает, накладываются ли на эти интерфейсы какие-либо ограничения, связанные с форматами данных и носителями информации.

3) Данные. Требования этой категории описывают входные и выходные данные системы, какой при этом используется формат, ограничения, нужно ли сохранять эти данные, какой объем данных поступает в систему, с какой скоростью передачи, с какой точностью должны выполняться вычисления.

4) Производительность. Требования этой категории описывают проблемы масштабирования и синхронизации. Например, количество пользователей, которое одновременно должна обслуживать система, время ожидания ответа на запрос (следует соблюдать осторожность при выражении этих требований в числовых значения, т. к. их необходимо будет проверять).

5) Пользователи и человеческий фактор. В этих требованиях описывается квалификация пользователя этой системы, уровень удобства и простоты использования, например, максимальное количество действий для выполнения некоторой операции в системе.

6) Безопасность. Требования этой категории описывают, как осуществляется доступ к системе, к ее данным, где данные должны дублироваться и как часто.

7) Документация. Требования определяют, должна ли быть документация к системе, в печатном или в интерактивном виде, для кого она предназначена.

8) Устранение неисправностей. Требования этой категории описывают, как система должна реагировать на возникновение неисправностей, надо ли выдавать аварийный сигнал, какое время простоя ожидается.

9) Сопровождение. Здесь описываются требования о том, как производится устранения проблем, и каковы условия поставки новых версий программы.

3. МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

3.1. Модульное тестирование и его задачи

Модульное тестирование (англ., Unit Testing) – это вид тестовой деятельности, при котором проверке подвергаются внутренние рабочие части программы, элементы или модули независимо от способа их вызова. Под модулем принято понимать программу или ограниченную часть кода с одной точкой входа и одной точкой выхода, которая выполняет одну и только одну первичную функцию. Этот тип тестирования, как правило, осуществляется программистом, а не тестировщиком, поскольку для его проведения необходимы доскональные знания структуры и кода программы.

Тестировать свои собственные продукты – это весьма трудное занятие для разработчиков, поскольку они вынуждены изменить свою точку зрения или отношение, перейдя от роли создателя в роль критика. Многие разработчики не любят тщательно тестировать свои программные продукты, так как считают скучным и даже угнетающим наблюдать за тем, как прикладная программа справляется с возложенными на нее задачами. Другие не имеют ничего против подобного подхода и прекрасно выполняют работу по тестированию час-

тей программы. Такой подход, несомненно, приводит к обнаружению ошибок на более ранних этапах разработки программного продукта, а это в свою очередь способствует снижению стоимости ошибки. Даже самый опытный программист допускает ошибки. Некоторые исследования показывают, что плотность распределения дефектов находится в диапазоне от 49,5 до 94,6 на тысячу строк кода. Поэтому каждый разработчик должен самостоятельно тестировать свои «произведения» с максимальной тщательностью, прежде чем передать рабочий продукт независимому испытателю, т. е. тестировщику.

Методы, используемые при модульном тестировании, различаются по следующим признакам [7]:

a) по степени автоматизации – ручные и автоматизированные методы;

b) по форме представления модуля – символьное представление (на языке программирования) или в машинном коде;

c) по компонентам программы, на которые направлено тестирование, – структура программы или преобразование переменных;

d) по запуску программы – статические или динамические методы.

В первую очередь следует тестировать структуру программы, так как операторы анализа условий составляют в среднем 10–15 % от общего числа операторов программы [7]. Искажение логики работы программы приводит к серьезным ошибкам. К тому же данный вид тестирования имеет наилучшие показатели «эффективность/стоимость». Там же предлагается следующая методика тестирования модулей: последовательно проводить различные виды тестирования, начиная с самых простых:

– ручное тестирование (работа за столом);

– символическое тестирование (инспекции, сквозные просмотры);

– тестирование структуры;

– тестирование обработки данных;

– функциональное тестирование (сравнение со спецификацией, взаимодействие с другими модулями).

Из приведенных выше видов тестирования, ручное и символическое относятся к статическому тестированию, которое проводится без запуска программы. Эти два вида, как правило, основаны на обзорах программного кода, только первый проводится автором-разработчиком, а второй – сторонними специалистами (другими

разработчиками, инженерами по качеству или приглашенными инспекторами).

Последние три из перечисленных видов относятся к динамическому тестированию, и проведение каждого из них состоит из следующих этапов:

– планирование тестирования (разработка тестов, формирование контрольных примеров);

– собственно тестирование;

– обработка результатов тестирования.

3.2. Обзоры программного кода

Как уже было сказано, обзоры программного кода являются основой статического тестирования и способны нейтрализовать действие человеческого фактора при выполнении поставленных задач. При модульном тестировании можно выделить следующие положительные моменты обзоров (учитывая, что оно может проводиться как самим разработчиком, так и сторонними специалистами, которых будем называть экспертами) [8, 9]:

1. Позволяют обнаружить посторонние элементы, что нельзя сделать в ходе традиционного тестирования. Обзоры могут следовать по всем выполняемым ветвям разрабатываемого ПО, а с помощью динамического тестирования невозможно проверить все ветви. В результате этого, редко проходимые ветви часто приводят к появлению ошибок.

2. Разные точки зрения, личностные качества и жизненный опыт помогают при обнаружении всех видов проблем, которые незаметны при поверхностном взгляде.

3. Разработчики могут больше внимания уделять творческому аспекту процесса разработки, зная, что эксперты участвуют в проекте и действуют в качестве «подстраховки».

4. Происходит разделение труда, благодаря чему эксперты могут сконцентрироваться на этапе обнаружения проблем.

5. Распространение информации и обучение происходят тогда, когда разработчики встречаются при проведении обзоров кода. Люди быстро воспринимают и распространяют хорошие идеи, которые они замечают в работе других. По словам некоторых экспертов – это самый важный результат выполнения обзоров.

6. Возрастает степень согласованности между членами команды. Происходит установление фактических групповых норм, что облегчает понимание сути программных продуктов и их сопровождение.

7. Менеджеры проекта могут получить представление о действительном состоянии разрабатываемых продуктов.

В конечном счете, в результате обзоров программные продукты улучшаются в силу следующих причин:

– Проблемы обнаруживаются тогда, когда их можно относительно легко исправить без значительных понесенных затрат.

– Локализация проблем происходит практически в месте их возникновения.

– Инспектируются исходные данные какой-либо фазы с целью проверки их соответствия исходным требованиям или критериям выхода.

– Предотвращается возникновение дальнейших проблем с помощью оглашения решений часто происходящих затруднений.

– Распространяются сведения о проекте, что способствует повышению его управляемости.

– Разработчики обучаются тому, каким образом избежать возникновения дефектов при дальнейшей работе.

– Предотвращается возникновение дефектов в текущем продукте, поскольку процесс подготовки материалов для инспекционной проверки способствует уточнению требований и проекта.

– Обучаются новые участники проекта.

– Менеджерам проекта предоставляются надежные опорные точки и предварительные оценки.

– Облегчается поддержка установленного порядка выполнения проекта и обеспечивается объективная, измеримая обратная связь.

3.3. Принципы тестирования структуры программных модулей

Как правило, тестирование структуры программного модуля происходит с помощью методов графического отображения модуля, одним из которых являются ориентированные графы МакКейба [7, 10]. В качестве узлов в графах МакКейба выступают операторы, в качестве ребер – связи между операторами (рис. 3.1).

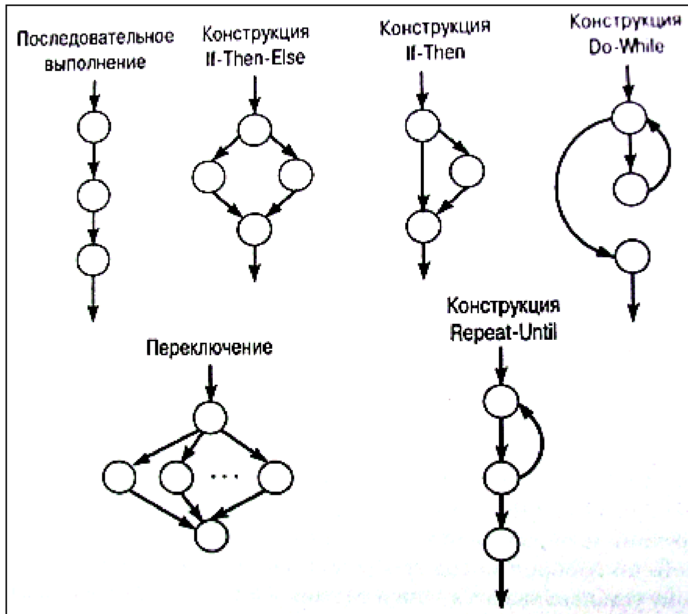

Рис. 3.1. Графические конструкции МакКейба [7]

Тогда для каждого программного модуля может быть построен граф, который графически отобразит все возможные логические проходы по модулю. Проходом (в литературе также встречается термин «маршрут») будем называть путь от первой вершины графа до последней. В основных графических конструкциях МакКейба, приведенных выше, отсутствуют некоторые современные конструкции, например, *try-catch-finally*. Для ее графического отображения можно воспользоваться конструкциями *if-then* или *if-then-else*.

Метрический показатель сложности или цикломатическое число G потокового графа определяется по формуле

$$G = R - V + 2, \qquad (3.1)$$

где R – количество ребер графа;

V – количество вершин графа.

Следует отметить, что данная формула справедлива только для графа, который начинается с одной вершины и заканчивается одной

вершиной. В противном случае цикломатическое число определяют визуально по всем возможным проходам по графу.

Чтобы вычислить этот коэффициент вручную для большого и сложно-структурированного программного модуля, потребуется немало усилий. К счастью, существуют автоматические средства, которые вычисляют метрический показатель сложности МакКейба путем анализа исходного кода.

Метрический показатель сложности не только может пригодиться при установлении проблемных областей, но также может использоваться при создании контрольных примеров. Как только потоковый граф создан, очевидными становятся пути, ведущие через программный модуль, которые предоставляют информацию, необходимую для их выполнения в тесте.

При планировании тестирования структуры программных модулей решаются 2 задачи [7]:

1) формирование критериев выделения маршрутов в программе;
2) выбор стратегии упорядочения выделенных маршрутов.

Критерии выделения маршрутов в программе могут быть следующими:

a) минимальное покрытие графа программы;
b) маршруты, образующиеся при всех возможных комбинациях входящих дуг.

Стратегия упорядочения маршрутов выбирается по следующим параметрам:

– длительности исполнения и числу команд в маршрутах. Такая стратегия выбирается при тестировании программ вычислительного характера;

– количеству условных переходов, определяющих формирование данного маршрута. Такая стратегия выбирается при тестировании логических программ с небольшим объемом вычислений, а также в тех случаях, когда сложно оценить вероятность ветвления и количество исполнений циклов.

3.4. Способы тестирования взаимодействия модулей

После проведения модульного тестирования необходимо проверить совместную работу программных модулей, т. е. провести инте-

грационное тестирование. Выделяют два способа проверки взаимодействия модулей [11]:

1) монолитное тестирование;
2) пошаговое тестирование.

Пусть имеется программа, состоящая из нескольких модулей, представленных на рис. 3.2. Главным модулем программы является модуль *A*, требуемые данные в программу попадают через модуль *J*, а выводятся на экран через модуль *I*.

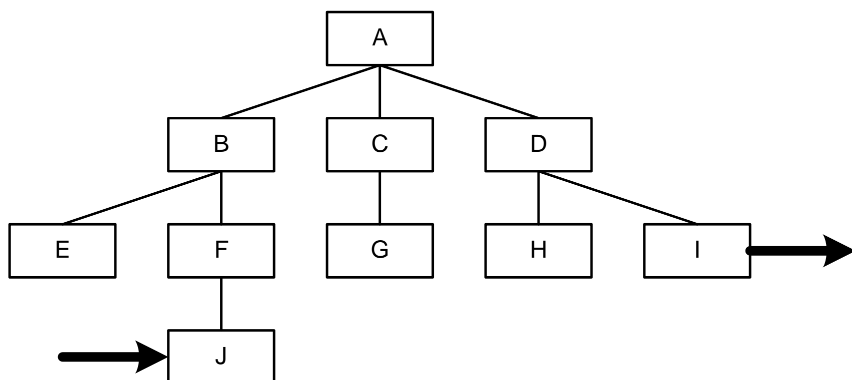

Рис. 3.2. Схема многомодульной программы

Обозначения на рис. 3.2 следующие: прямоугольники – это программные модули, тонкие линии представляют иерархию управления (связи по управлению между модулями), жирные стрелки – ввод и вывод данных в программу.

Монолитное тестирование [11] заключается в том, что каждый модуль тестируется отдельно. Для каждого модуля пишется один модуль-драйвер, который передает тестируемому модулю управление, и один или несколько модулей-заглушек. Например, для модуля *B* нужны 2 заглушки, имитирующие работу модулей *E* и *F*. Когда все модули протестированы, они собираются вместе и тестируется вся программа целиком.

Пошаговое тестирование предполагает, что модули тестируются не изолированно, а подключаются поочередно к набору уже оттестированных модулей.

Можно выделить следующие *недостатки* монолитного тестирования (перед пошаговым) [7, 11]:

1. Требуется много дополнительных действий (написание драйверов и заглушек).
2. Поздно обнаруживаются ошибки в межмодульных взаимодействиях.
3. Следствие из 2 – труднее отлаживать программу.

К *преимуществам* монолитного тестирования можно отнести:

1. Экономию машинного времени (в настоящее время существенной экономии не наблюдается)
2. Возможность параллельной организации работ на начальной фазе тестирования.

3.5. Стратегии выполнения пошагового тестирования

Существует две принципиально различные стратегии выполнения пошагового тестирования [7, 11]:

1) нисходящее тестирование;
2) восходящее тестирование.

Нисходящее тестирование начинается с главного модуля, в нашем случае с модуля *A*. Возникают проблемы: как передать тестовые данные в *A*, ведь ввод и вывод осуществляется в других модулях? Как передать в *A* несколько тестов?

Решение можно представить в следующем виде:

а) написать несколько вариантов заглушек модуля *B* (для каждого теста);

б) написать заглушку *B* так, чтобы она читала тестовые данные из внешнего файла.

В качестве стратегии подключения модулей можно использовать одну из следующих:

– подключаются наиболее важные с точки зрения тестирования модули;

– подключаются модули, осуществляющие операции ввода/вывода (для того, чтобы обеспечивать тестовыми данными «внутренние» модули).

Нисходящее тестирование имеет ряд недостатков: предположим, что модули *I* и *J* уже подключены, и на следующем шаге тестирования заглушка *H* меняется на реальный модуль *H*. Как передать

тестовые данные в *H*? Это нетривиальная задача, потому что между *J* и *H* имеются промежуточные модули, и может оказаться невозможным передать тестовые данные, соответствующие разработанным тестам. К тому же, достаточно сложно интерпретировать результаты тестирования *H*, так как между *H* и *I* также существуют промежуточные модули.

Восходящее тестирование практически полностью противоположно нисходящему тестированию. Начинается с терминальных (не вызывающих другие модули) модулей. А стратегия подключения новых модулей также основывается на степени критичности данного модуля в программе. *Восходящее тестирование* лишено недостатков нисходящего тестирования, однако имеет свой главный недостаток: рабочая программа не существует до тех пор, пока не добавлен последний (в нашем случае *A*) модуль.

Выбор одной из двух представленных стратегий определяется тем, на какие модули (верхнего или нижнего уровня) следует обратить внимание при тестировании в первую очередь.

3.6. Особенности объектно-ориентированного тестирования

С традиционной точки зрения в модульном тестировании наименьшим контролепригодным элементом является элемент или модуль, который можно протестировать методом «белого ящика» [8]. При объектно-ориентированном тестировании этот функциональный элемент не отделяется от своего класса, в котором он инкапсулирован со своими методами. Это означает, что поэлементное тестирование по существу заменяется классовым тестированием. Однако не следует отклоняться слишком далеко от корневых принципов относительно того, что каждый метод класса можно рассматривать как «небольшой элемент», тестирование которого можно выполнять обособленно, применяя для этого методы «белого» и «черного» ящика. Объекты класса необходимо протестировать во всех возможных состояниях. Это означает, что необходимо сымитировать все события, вызывающие изменения состояния объекта.

В традиционном отношении элементы компилируются в подсистемы и подвергаются интеграционным тестам [8]. При объектно-ориентированном подходе не применяется тестирование структур-

ной схемы сверху вниз, поскольку точно не известно, какой класс будет вызван пользователем за предыдущим. Интеграционные тесты заменяются тестированием функциональных возможностей, при котором тестируется набор классов, которые должны реагировать на один входной сигнал или системное событие, или же эксплуатационным тестированием, при котором описывается один способ применения системы, основанный на сценариях случаев эксплуатации.

Тестирование взаимодействия объектов следует по путям сообщения с целью отследить последовательность взаимодействий объектов, которая заканчивается только тогда, когда был вызван последний объект. При этом не посылается сообщение и не вызываются службы любого другого объекта.

Системное, альфа-, бета-тестирование и приемочное испытание пользователем изменяются незначительно, поскольку объектно-ориентированная система проходит эксплуатационные тесты точно так же, как и разработанные традиционным способом системы. Это означает, что процесс V&V по существу не изменяется.

Тестирование классов охватывает виды деятельности, направленные на проверку соответствия реализации этого класса спецификации. Если реализация корректна, то каждый экземпляр этого класса ведет себя подобающим образом.

Тестирование классов в первом приближении аналогично тестированию модулей и может проводиться с помощью обзоров кода и выполнения тестовых случаев.

Прежде чем приступать к тестированию класса, необходимо определить, тестировать его в автономном режиме, как модуль, или каким-то другим способом, как более крупный компонент системы. Для этого необходимо выяснить:

– роль класса в системе, в частности, степень связанного с ним риска;

– сложность класса, измеряемая количеством состояний, операций и связей с другими классами;

– объем трудозатрат, связанных с разработкой тестового драйвера для тестирования этого класса.

Если какой-либо класс должен стать частью некоторой библиотеки классов, целесообразно выполнять всестороннее тестирование классов, даже если затраты на разработку тестового драйвера окажутся высокими.

При тестировании классов можно выделить 5 оцениваемых факторов [8]:

1. Кто выполняет тестирование. Как и при модульном тестировании, тестирование классов выполняет разработчик, поскольку он знаком со всеми подробностями программного кода. Основной недостаток того, что тестовые драйверы и программные коды разрабатываются одним и тем же персоналом, заключается в том, что неправильное понимание спецификации разработчиками распространяется на тестовые наборы и тестовые драйверы. Проблем такого рода можно избежать путем привлечения независимых тестировщиков или других разработчиков для ревизий программных кодов.

2. Что тестировать. Тестировать нужно программный код на точное соответствие его требованиям, т. е. в классе должно быть реализовано все запланированное и ничего лишнего. Если для конкретного класса характерны статические элементы, то их также необходимо тестировать. Такие элементы принадлежат самому классу, но не каждому экземпляру.

3. Когда тестировать. Тестирование класса должно проводиться до того, как возникнет необходимость его использования. Необходимо также проводить регрессионное тестирование класса каждый раз, когда меняется его реализация. Однако до начала тестирования класса его необходимо закодировать и разработать тестовые случаи использования класса. Ранняя разработка тестовых случаев позволяет программисту лучше понять спецификацию класса и построить более успешную реализацию класса, которая пройдет все тестовые случаи. Существует даже практика первоначальной разработки тестовых случаев, а затем программного кода. Такой подход направлен на изначально все предусматривающий программный код и носит название разработка через тестирование (англ., Test-Driven Development, TDD). Главное, чтобы этот подход не привел к проблемам во время интеграции этого класса в более крупную часть системы.

4. Каким образом тестировать. Тестирование классов обычно выполняется путем разработки тестового драйвера. Тестовый драйвер создает экземпляры классов и окружает их соответствующей средой, чтобы стал возможен прогон соответствующего тестового случая. Драйвер посылает одно или большее количество сообщений экземпляру класса (в соответствии с тестовым случаем), затем сверяет результат его работы с ожидаемым и составляет протокол о про-

хождении или не прохождении теста. В обязанности тестового драйвера обычно входит и удаление созданного им экземпляра.

5. В каких объемах тестировать. Адекватность может быть измерена полнотой охвата тестами спецификации и реализации класса, т. е. необходимо тестировать операции и переходы состояний в различных сочетаниях. Поскольку объекты находятся в одном из возможных состояний, эти состояния определяют значимость операций. Поэтому требуется определить, целесообразно ли проводить исчерпывающее тестирование. Если нет, то рекомендуется выполнить наиболее важные тестовые случаи, а менее важные выполнять выборочно.

3.7. Автоматизация модульного тестирования

Автоматизация модульного тестирования направлена за замену ручного запуска модульных тестов некоторым инструментальным средством. Для автоматизации модульного тестирования существует достаточно большое количество инструментов (их принято называть фреймворками), которое можно разделить на 3 группы:

– семейство xUnit;
– встроенные фреймворки в среды разработок;
– универсальные инструменты для разных видов тестирования.

3.7.1. Семейство xUnit

Семейство xUnit – это набор программных средств для разработки модульных тестов, позволяющий реализовать построение тестов, их выполнение и вывод полученных результатов. Семейство xUnit имеет отдельную реализацию для большинства используемых в настоящее время языков программирования, например:

– JUnit – для тестирования Java-кода;
– DUnit – для тестирования программ на Delphi;
– cppUnit – для тестирования c++ программ;
– NUnit – для тестирования .NET приложений;
– PyUnit – для тестирования программ, написанных на Python;
– VBUnit – для программ, написанных на VisualBasic;
– utPLSQL – для языка PLSQL (Oracle).

Также существует инструмент httpUnit, который реализован на языке Java и имеет следующие возможности: эмулирует основное поведение браузера, включая заполнения веб-форм, работу с элементами JavaScript, основную http-аутентификацию, автоматическую страничную переадресацию, поддерживает работу с cookies, позволяет тестовому коду проверить возвращаемую страницу в виде html-текста с контейнерами форм, таблиц и линков. Используя возможности классов JUnit, позволяет легко создавать тестовые сценарии и быстро проводить автоматизированное тестирование функциональности веб-приложения. Инструмент httpUnit позволяет работать напрямую с веб-сервером без участия веб-браузера, заменяя его собой (рис. 3.3–3.4).

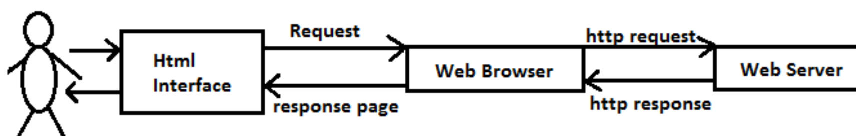

Рис. 3.3. Схема работы веб-приложения

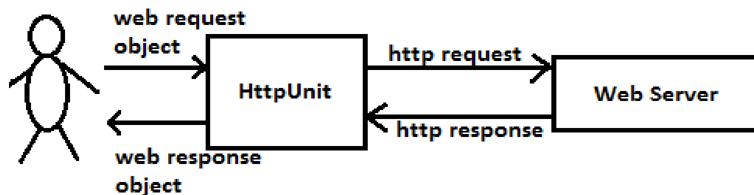

Рис. 3.4. Схема работы инструмента httpUnit

К преимуществам httpUnit можно отнести следующие:

– высокая скорость работы скриптов, а также их работа в фоновом режиме, поскольку отсутствует посредник в виде браузера;

– тестирование «чистого» приложения без учета особенностей браузера;

– бесплатность, поскольку httpUnit является свободно распространяемым инструментом автоматизации тестирования.

Основные классы httpUnit приведены в табл. 3.1.

Таблица 3.1

Основные классы httpUnit

Название класса	Назначение класса
WebConversation	Основной класс HttpUnit. В объектах этого класса хранится информация о сессии браузера, которая возвращается сервером в виде cookies. Для работы с веб-сервером разработчик должен создать запрос и обратиться к WebConversation за ответом. Таким образом, WebConversation имитирует основное взаимодействие пользовательского браузера с веб-приложением, а объект хранит последовательность посылаемых на веб-сервер запросов и возвращаемых в ответ откликов с текстом веб-страниц.
WebRequest	Представляет собой http-запрос, посылаемый веб-серверу приложения. Объект WebRequest хранит всю необходимую информацию, которая должна содержаться в http-запросе, а именно: url, parameters, наборы символов.
WebResponse	Представляет собой http-отклик, возвращаемый веб-сервером приложения. Объект WebResponse хранит информацию о http-отклике, полученном по определенному http-запросу, а именно: тело отклика, наборы символов, исходный код и тип контекста html-страницы. WebResponse предоставляет набор методов для получения отдельных частей исходного кода веб-страницы в разобранном и структурированном виде.
WebForm	Представляет собой html-форму, содержит набор методов для чтения и записи данных из различных полей веб-форм, а также для утверждения формы.
WebTable	Представляет собой html-таблицу, содержит набор методов для чтения и записи данных из различных ячеек веб-таблицы, получения количества строк и столбцов таблицы и т. д.
WebLink	Представляет собой html-ссылку (линк), содержит всю информацию об определенной веб-ссылке: url, parameters и текст ссылки. Объект WebLink позволяет имитировать нажатие на веб-ссылку левой клавишей мыши и переход на новую веб-страницу по необходимому адресу.

Пример:

Поскольку класс WebConversation ответственен за поддержание соединения с сервером, то для его использования необходимо создать запрос, а затем опрашивать объект WebConversation:

WebConversation wc = new WebConversation();

WebRequest req = new GetMethodWebRequest(“URL”);

WEbResponse res = wc.GetResponse(req);

Ответом (response) можно манипулировать как простым текстом, используя, например, *GetTextMethod()* или как объектом DOM, используя *GetDOMMethod()*, или пользуясь другими методами.

3.7.2. Встроенные инструменты для автоматизации модульного тестирования

Самым распространенным встроенным инструментом является MSTest, который интегрирован в MS Visual Studio, и предназначен для тестирования программных модулей, реализованных в этой среде.

Проведем сравнительный анализ инструментов NUnit и MSTest для модульного тестирования. В табл. 3.2 приведена последовательность действий для создания тестов с использованием каждого из рассматриваемых инструментов, а в табл. 3.3 – названия атрибутов для пометок модульных тестов.

Таблица 3.2

Последовательность действий для создания тестов в NUnit и MSTest

NUnit	**MSTest**
Скачать библиотеку NUnit, подключить ее к проекту, создать класс для написания в нем тестов. Создать тестовые методы и пометить их с помощью специальных атрибутов. Для тестируемых методов создать тестовые заглушки.	Использовать версию MS Visual Studio с интегрированным в нее модулем UnitTest. Для создания теста для определенного метода щелкнуть по нему правой кнопкой мыши и выбрать Create Unit Test. После добавления нового тестового проекта MS Visual Studio создаст класс с тестовыми методами для отмеченных тестируемых методов.

Таблица 3.3

Таблица атрибутов NUnit и MSTest

Название	**NUnit**	**MSTests**
Атрибут для тестового класса	TestFixture	TestClass
Атрибут для тестового метода	Test	TestMethod
Атрибут для метода, выполняющегося перед всеми тестами	TestFixtureSetUp	ClassInitialize
Атрибут для метода, выполняющегося после всех тестов	TestFixtureTearDown	ClassCleanUp
Атрибут для метода, выполняющегося перед каждым тестом	TearUp	TestInitialize
Атрибут для метода, выполняющегося после каждого теста	TearDown	TestCleanUp

К недостаткам инструмента NUnit принято относить отсутствие *generic* методов (вместо них используются перегрузки), невозможность тестировать свойства классов, необходимость создания заглушек для тестирования методов с типом доступа *private*. Преимуществом NUnit является его бесплатное распространение.

К недостаткам MSTests относят его высокую стоимость в составе среды разработки MS Visual Studio, а из преимуществ отмечают удобство создания тестов вследствие отсутствия необходимости создавать классы и методы для них вручную.

Анализ двух инструментов показывает, что их возможности практически идентичны, последовательность действий для создания тестов однотипна, поэтому каждый вправе выбирать инструмент по своему усмотрению.

3.7.3. Использование универсальных инструментов автоматизации тестирования

Самым популярным инструментом для проведения различных видов автоматизированного тестирования является TestComplete. Автоматизация модульного тестирования является лишь небольшой частью его возможностей, которую можно реализовать двумя способами:

1) Подключение тестов, ранее написанных в других средах, например, в MSTest либо NUnit. Для запуска таких тестов существует два варианта:

– *вариант 1* – указать в настройках TestComplete расположение этого стороннего теста (Tools→Options→UnitTesting), создать проект для модульного тестирования, указав, что тесты были созданы в другом инструменте. Далее подключаем ранее созданные библиотеки dll, содержащие тесты. При таком варианте инструменту TestComplete отводится роль средства по обработке, сбору и представлению результатов в удобной форме;

– *вариант 2* – в исходном тестовом инструменте подключить библиотеку AutomatedQA.TestComplete.UnitTesting.dll, указать, какие классы будут видимы для TestComplete. Затем производится создание и настройка проекта в TestComplete и далее запуск тестов. На панели справа можно выбрать запуск всех тестов либо только выбранных. При таком варианте TestComplete принимает непосредственное участие в запуске тестов и обработке результатов, причем процесс происходит несколько быстрее.

2) Непосредственное создание модульных тестов в TestComplete. Для этого необходимо реализовать следующие этапы:

a) Создание проекта. В TestComplete в окне Create New Project указывается имя, расположение и скриптовый язык будущего тестового проекта. В открывшемся окне Project Wizard можно указать, какие элементы проекта будут в него включаться (скрипты, список тестируемых приложений и т. д.). После нажатия кнопки Finish, TestComplete создаёт новый проект и отображает его содержимое в панели Project Explorer.

b) Подготовка модульных тестов. TestComplete может видеть внутренние объекты тестируемого приложения и позволяет использовать три варианта написания тестов:

– *вариант 1* – создать метод, который тестирует остальные методы класса. Такой вариант реализуем, поскольку TestComplete может вызывать функции и методы непосредственно из кода приложения и обрабатывать исключения, которые случаются при неправильной работе тестируемого метода. В итоге не придется писать дополнительные тестовые классы;

– *вариант 2* – создать функцию, которая будет тестировать методы класса и генерировать исключение при ошибке. Функция до-

бавляется в тот же класс, где содержатся методы. Преимущества данного метода: количество классов в проекте не изменяется, функция имеет доступ к методам с модификаторами *private* и *protected*;

– *вариант 3* – это создание отдельного класса для тестирования исходного класса. Преимущества: тестируемый и тестирующий классы разделены, что дает возможность легко деактивировать модули, которые содержат тесты.

c) Подготовка тестируемого приложения. Для этого необходимо открыть тестируемый проект, в разделе Bin/Extentions добавить ссылку на AutomatedQA.TestComplete.UnitTesting.dll, установить свойство CopyLocal = True. Затем необходимо вызвать метод AddClasses(), принадлежащий объекту UnitTesting, например, в следующей реализации:

Using TestComplete;
Type[] types = {typeof(MyTestClass)};
UnitTesting.AddClasses(types);

После добавления классов все методы, принадлежащие им, будут видны для TestComplete. Затем необходимо скомпилировать приложение и запустить его.

d) Загрузка тестов. В созданный проект TestComplete необходимо добавить тестируемое приложение (Tested Applications→Add→New Item), которое должно быть запущено. Добавить объект TCUnitTest в раздел UnitTesting. В появившемся окне выбрать процесс, соответствующий тестируемому приложению. Загрузить доступные тесты, выбрав для этого Menu->Mode->Load. Если был выбран автоматический режим, то тесты будут загружены автоматически.

e) Запуск тестов. Для удобства запуска тестов можно добавить несколько строк кода в модуль Script, в результате чего TestComplete будет автоматически запускать тестируемое приложение и прогонять доступные тесты. Результаты каждого прогона модульных тестов сохраняются в UnitTesting логах и могут быть открыты для анализа.

4. ПЛАНИРОВАНИЕ ФУНКЦИОНАЛЬНОГО ТЕСТИРОВАНИЯ

После появления первой рабочей версии требований к программному продукту тестировщики приступают к подготовительным работам для проведения функционального тестирования, т. е. пла-

нируют свою тестовую деятельность. На этом этапе необходимо определиться со следующими моментами:

– какой программный продукт будет тестироваться, каково его назначение;

– как будет использоваться продукт;

– какие части продукта будут тестироваться, а какие нет;

– какие методы и техники тестирования наиболее эффективны для данного проекта;

– какие риски могут возникнуть в процессе тестирования (риск – это возможная ситуация, которая негативно повлияет на результативность процесса либо приведет к увеличению сроков реализации, например, эпидемия гриппа в осенний период или период отпусков летом).

Все указанные выше моменты оформляются в тестовый план и рассылаются ключевым участникам проекта. Параллельно тестировщиками разрабатываются тестовые случаи.

4.1. Тестовый план

Тестовый план – это документ, согласно которому будут проводиться все действия по тестированию. Ответственным за разработку тестового плана, как правило, является руководитель команды тестирования.

Можно привести примерное содержание тестового плана:

1. Объемы тестирования (перечень тестируемых и нетестируемых компонент).
2. Критерии качества (какое количество ошибок может быть отклонено при сдаче продукта).
3. Риски тестирования.
4. Документация тестирования (тестовый план, тестовые случаи, отчеты об ошибках).
5. Стратегия тестирования (это самый большой и самый важный пункт, в котором описываются применяемые виды тестирования и градация тестов).
6. Ресурсы (перечень человеческих ресурсов с разделением прав и обязанностей, а также аппаратные ресурсы: сервера, рабочие станции, инструменты тестирования, описание тестового окружения).

7. График тестирования (основывается на графике выпуска версий).

Составленный тестовый план должен быть проверен и удовлетворять следующим критериям:

– является полным, корректным, недвусмысленным;

– цели каждого вида тестирования определены;

– четко определена стратегия тестирования;

– является реально выполнимым;

– определено тестовое оборудование;

– оговорены условия прекращения тестирования;

– определены ресурсы (человеческие, аппаратные, программные) для тестирования.

4.2. Разработка тестовых случаев

Тестовый сценарий (англ., Test Script) – это алгоритм проверки некоторой функциональности программы в совокупности с ожидаемыми результатами. Примером тестового сценария может быть, например, проверка функциональности добавления некоторой сущности в базу данных. В процессе добавления возможны разные случаи: верно заполнить все поля сущности, выборочно верно заполнить поля, неверно заполнить поля, оставить одно поле пустым, оставить два поля пустыми и т. д. Все отдельные случаи проверок называются *тестовыми случаями* (англ., Test Case). Тестовые случаи организуются в *тестовые наборы* (англ., Test Suite), например, режим работы администратора или незарегистрированного пользователя. Тестовые случаи могут изменяться в течение работы над проектом и полностью формируются по мере понимания задачи, по мере изменения требований, а также по другим причинам. Процесс разработки тестовых случаев состоит из следующих этапов:

1. Ознакомление с проектной документацией: требованиями к программному продукту (либо спецификацией), тестовым планом и непроектной информацией (например, литературой по предметной области для разрабатываемого ПО).

2. Проектирование тестовых случаев. При разработке тестовых случаев актуальным понятием является модульность. При модульном проектировании система разбивается на отдельные компоненты, каждый из которых имеет свое назначение и четко определенные

входы и выходы. В функциональном тестировании этим модульным компонентом является тестовый случай. Это значит, что перед каждым тестовым случаем должна быть поставлена четко сформулированная цель (что именно подвергается тестированию), определена среда тестирования с известными начальными условиями (благодаря чему можно рассчитывать, что при каждом прогоне конкретный тест будет выдавать один и тот же результат). Также один тестовый случай должен проверять только одну функциональность и иметь строго определенный ожидаемый результат (чтобы было понятно успешно или неуспешно прошло испытание).

Формирование тестовых наборов связано с градацией тестов, из которой наиболее распространенной является разбивка на три блока:

– тесты для проверки «на дым» (англ., Smoke Test) – это краткая и быстрая проверка основной функциональности программы с целью принять или отклонить версию программы для дальнейшего тестирования;

– тесты для позитивного тестирования – проверка работы программы в стандартных ситуациях (вводятся верные данные, нажимаются нужные кнопки, имитируется работа «положительного» пользователя);

– тесты для негативного тестирования – проверка работы программы в нестандартных ситуациях (вводятся неверные данные, нажимаются ненужные кнопки, имитируется работа «плохого» пользователя.

3. Написание тестовых случаев. На этом этапе необходимо придерживаться следующих свойств тестовых случаев:

a) тестовый случай должен обладать высокой вероятностью обнаружения дефекта (для этого тестировщик должен стать на позицию конструктивного разрушения);

b) тестовый случай должен быть воспроизводимым;

c) тестовый случай должен обладать четко определенными ожидаемыми результатами и критериями успешного выполнения теста;

d) набор тестовых случаев не должен быть избыточным.

4. Проверка тестовых случаев. По окончании разработки тестовых случаев проводится их проверка, в течение которой обращают внимание на следующие моменты:

– соответствует ли тестовый случай функциональности, заявленной в требованиях;

– покрывают ли тестовые случаи все требования к программному продукту (для этого используется матрица прослеживаемости требований);

– организованы ли тестовые случаи достаточно эффективно, чтобы можно было обходиться минимальными конфигурациями средств тестирования;

– включены ли тесты в систему управления конфигурациями;

– имеются ли среди тестов избыточные, можно ли устранить эту избыточность;

– снабжен ли каждый тестовый случай ожидаемыми результатами;

– достаточно ли подробно разработана методика тестирования, чтобы стала возможной ее автоматизация.

Пример тестового случая добавления нового студента в группу для позитивного теста приведен в табл. 4.1.

Таблица 4.1

Пример тестового случая добавления нового студента

№	№ требования	Название модуля/ экрана	Описание тестового случая	Ожидаемые результаты	Тест пройден? Да/Нет
1	R1001	Режим Администратора, Добавление студента в группу	**Добавление нового студента в группу** 1. Нажать кнопку <Добавить>. 2. Заполнить поля данными: Иванов Иван Иванович, выбрать группу 107222. 3. Нажать <Сохранить>	1. Появляется окно с пустыми полями: Фамилия, Имя, Отчество, а также выпадающий список с существующими номерами групп. 2. Вводимая информация появляется в полях, группа выбрана. 3. Если такого студента в базе данных нет, то он добавляется в список, иначе на экране появляется сообщение «Такой студент уже имеется. Изменить? Да/Нет». Если «Да», то система возвращает курсор в первое поле для изменений. Если «Нет», то система возвращается в первоначальное окно без добавления нового студента.	

4.3. Эквивалентирование и анализ граничных значений

При планировании функционального тестирования неизбежной является задача выбора тестовых данных. Для сокращения переборов область всех входных данных программы можно разбить на конечное число *классов эквивалентности*. Такое разбиение основано на принципе эквивалентности из общей теории систем, который в данном контексте можно выразить следующим образом: если система реагирует некоторым образом на входное значение $x1$, то существует довольно большая вероятность, что система отреагирует аналогичным образом на входное значение $x2$, очень близкое к $x1$. Тогда все множество входных данных системы можно разбить, как минимум, на два класса эквивалентности: класс верных значений (они войдут в позитивный тест) и класс неверных значений (для негативного теста). Тогда граничными значениями будем называть те значения, которые располагаются на границах классов эквивалентности, а эквивалентными – те, которые расположены внутри классов эквивалентности. Например, рассмотрим целочисленное поле для ввода цены товара, которое по требованиям может находиться в диапазоне [0; 1000000] рублей (табл. 4.2).

Таблица 4.2

Пример разбиения на классы эквивалентности

Класс верных значений		Класс неверных значений	
Граничные	Эквивалентные	Граничные	Эквивалентные
0, 1000000	1, 999999 500000	–1, 100000001	–2, 10000000000000, 0, –50000 10.5, абв, 2$

Как видно из примера, в качестве эквивалентных данных принимаются значения, близкие к левой границе, к правой границе и для позитивного теста близко к середине диапазона. Эквивалентные значения из класса неверных могут быть абсолютно разными, но выбранными с целью поломки программы и проверки ее реакции в непредвиденных ситуациях. Независимо от классов эквивалентности рекомендуется проверять 0, –1, +1 и специальные символы.

5. РЕАЛИЗАЦИЯ ФУНКЦИОНАЛЬНОГО ТЕСТИРОВАНИЯ

5.1. Ошибка, свойства ошибки

Функциональное тестирование направлено на поиск ошибок в заявленной функциональности программы. Поэтому ключевым понятием здесь является термин «ошибка». Существует довольно много определений ошибки, приведем некоторые из них:

– ошибка – это расхождение между программой и ее спецификацией. Определение часто критикуют, поскольку пользователь программы может не иметь спецификации, но на экране увидеть, например, Error 404.

– если программа не делает того, чего пользователь от нее вполне обоснованно ожидает, значит налицо программная ошибка (Гленфорд Майерс, [1]);

– не существует ни абсолютного определения ошибок, ни точного критерия наличия их в программе, можно лишь сказать, насколько программа не справляется со своей задачей – это исключительно субъективная характеристика (Борис Бейзер, [12]).

Последние два определения имеют право на существование, однако для краткости будем называть *ошибкой* несоответствие ожидаемых результатов с фактическими полученными.

Выделим следующие свойства ошибок:

1) *Важность*. В этом свойстве могут быть определены следующие уровни:

– критическая важность, когда произошел сбой либо отказ системы, и дальнейшая работа с программой невозможна;

– серьезная важность, когда не работает основная функциональность программы (например, добавление записи в базу данных);

– средняя важность, когда не работает второстепенная функциональность либо имеются недочеты в работе основной функциональности (например, не работает сортировка товара в списке, или не появляется сообщение для подтверждения удаления товара из базы данных);

– низкая важность, когда имеется мелкая ошибка (например, ошибка интерфейса либо орфографическая ошибка).

2) *Воспроизводимость*. Это свойство связано с частотой и условиями воспроизведения, поэтому здесь выделяют два уровня:

– всегда, т. е. ошибка воспроизводится на всех устройствах и не зависит ни от каких условий;

– иногда, т. е. ошибка воспроизводится только при определенных условиях, например, только в определенном браузере.

3) *Приоритет*. Это свойство связано со скоростью исправления ошибки. Можно выделить следующие уровни приоритета:

– очень высокий, когда ошибка должна быть исправлена немедленно;

– высокий, когда ошибка должна быть исправлена как можно скорее;

– средний, когда ошибка может быть исправлена, когда появится свободное время;

– низкий, когда ошибка может быть исправлена в последнюю очередь.

4) *Симптом*. Это свойство также называют категорией ошибки. К одной ошибке иногда может подходить несколько симптомов, поэтому при документировании ошибки достаточно указать один из них, но, желательно, самый точный. Перечень симптомов может отличаться в зависимости от используемой системы управления ошибками (см. п. 5.5), поэтому приведем наиболее часто встречающиеся:

– неожиданное поведение, когда, например, вместо сообщения о сохранении данных, появляется сообщение об их успешном удалении;

– недружественное поведение, когда, как правило, программа не сопровождает свои действия сообщениями и предупреждениями;

– неверное действие, когда программа не выполняет требуемое действие или выполняет неверно, например, не удаляет запись из базы данных;

– отказ системы, когда дальнейшая работа с программой невозможна;

– потеря данных, когда, например, добавленные данные исчезли;

– искажение данных, когда произошла замена данных программы искаженными, или, например, данные выводятся на экран в другой кодировке;

– низкая производительность, когда, например, слишком медленно идут вычисления, или медленно раскрывается таблица;

– локализационная ошибка может быть связана с воспроизведением на разных устройствах, а также, например, в связи со сменой локализации сайта на другой иностранный язык;

– инсталляционная ошибка, которая проявляется во время разных способов инсталляции ПО;

– косметическая ошибка связана с интерфейсом программы;

– ошибка документации связана с недочетами в любой документации, используемой в процессе разработки ПО;

– различия со спецификацией, когда, например, названия элементов интерфейса, их количество, цвет, внешний вид и т. д. не соответствуют заявленным в требованиях или в спецификации к программному продукту;

– отсутствующая функциональность, когда функциональность, заявленная в требованиях, отсутствует в программном продукте (например, отсутствует поле для поиска и, соответственно, вся функциональность поиска);

– запрос на улучшение является симптомом, который может быть выставлен, когда тестировщик предлагает улучшить некоторую функциональность программы либо ее внешний вид. Таким образом, ошибка с этим симптомом может не противоречить требованиям, и, соответственно, может не быть исправлена.

5.2. Правила составления отчетов об ошибках

Процесс поиска ошибок, как правило, состоит в прохождении одного за другим заранее запланированных тестовых случаев, сверки ожидаемых результатов с фактически полученными и вывода, прошел тест или нет. При этом заполняется последняя колонка табл. 4.1. Если тест не пройден, значит найдена ошибка, которая должна быть задокументирована и передана программисту на исправление. Для того чтобы исправление ошибки проходило без особых затруднений, отчет об ошибке должен составляться быстро, но при этом его тон и содержание должны максимально способствовать решению проблемы. Поэтому необходимо проанализировать ошибку, воспроизвести ее несколько раз, выяснить условия воспроизведения, данные, при которых она появляется. И только потом описать ее предельно кратко и четко, поскольку слишком пространное и расплывчатое описание проблемы затрудняет понимание ее сути. Кроме свойств ошибки, описанных выше, необходимо заполнить поля, приведенные в табл. 4.3.

Таблица 4.3

Информация об обнаруженной ошибке

Характеристика	**Описание**	**Обоснование**
Идентификатор ошибки	Уникальный идентификатор, обычно строка из алфавитно-цифровых символов.	Позволяет отслеживать ошибку в процессе изменения ее состояний.
Статус ошибки	Один из допустимых статусов.	Присвоить ошибке статус «Новый» в момент обнаружения.
Кем обнаружен	Имя исполнителя, обнаружившего дефект.	Обеспечивает возможность задавать вопросы относительно дефекта.
Дата обнаружения	День/месяц/год.	Позволяет отслеживать хронологию событий, связанных с ошибкой.
Краткое описание проблемы	Описание на концептуальной форме, как правило, одной строкой.	Описание используется в сообщениях о статусе ошибки.
Описание проблемы	Детальное описание симптомов проблемы и того, как она ограничивает функциональные возможности продукта.	Предназначено для тех, кому необходимо детальное описание проблемы, например, разработчику, который работает над устранением ошибки.
Как воспроизвести проблему	Детально проработанная методика воспроизведения проблемы.	Позволяет разработчикам локализовать проблему.

Ошибки, как правило, сохраняют в автоматизированные системы документирования и отслеживания ошибок (см. п. 5.3). Каждая из таких систем имеет свои особенности при составлении отчетов об ошибках и поля для заполнения, однако можно предложить следующие общие правила:

1) Составляйте отчет об ошибке сразу же после ее обнаружения, иначе про нее можно забыть.

2) Не составляйте отчет на бумаге (поскольку листок может легко потеряться), а заносите в систему документирования и отслеживания ошибок.

3) Составьте полное описание ошибки с указанием операционной системы, браузера, базы данных и/или других подробностей.

4) Опишите подробно шаги для воспроизведения ошибки, чтобы разработчик мог их повторить и увидеть ошибку.

5) Придумайте краткое, но емкое название для ошибки.

6) Не путайте описание ошибки и шаги воспроизведения, а также название ошибки и описание ошибки.

7) Укажите важность ошибки, симптом, частоту появления и приоритет.

8) Пользуйтесь простым и доходчивым языком при составлении отчета об ошибке.

9) Не обвиняйте никого в обнаруженной ошибке, поскольку отчет не должен приводить к конфликту между тестировщиками и программистами, а лишь способствовать разработке качественного программного продукта.

5.3. Системы документирования и отслеживания ошибок

Системы документирования и отслеживания ошибок (англ., Bug Tracking Systems, BTS) в большинстве своем, не ограничиваются работой с ошибками, а представляют собой системы управления проектами. Они позволяют отслеживать процесс тестирования, проверять и составлять отчеты. Общее назначение таких систем можно описать следующим образом:

1) повышать взаимодействие между сотрудниками;

2) ни одна ошибка не должна быть не исправлена, потому что так решил разработчик;

3) как можно меньше ошибок должно остаться из-за проблем взаимодействия сотрудников.

Несмотря на то, что существует довольно большой перечень BTS, все их количество можно разбить на классы:

– свободно распространяемые (например, Bugzilla, Mantis, Redmine, Track);

– платные системы (например, Jira, ClearQuest, TestTrackPro);

– собственная разработка (т. е. компания разработала для себя такую систему).

Следует отметить, что выбор BTS зачастую определяется заказчиком программного обеспечения, молодые и немногочисленные компании останавливаются на бесплатных системах, а крупные компании в состоянии позволить себе собственную разработку по своим требованиям. Тем не менее, можно предложить следующие критерии выбора BTS:

1) доступная система для различных платформ;
2) наличие клиент-серверного приложения;
3) поддержка работы с различными базами данных;
4) стоимость и схема лицензирования;
5) интегрирование в различные среды;
6) гибкость настройки системы;
7) электронная поддержка формирования событий и отчетов.

5.4. Жизненный цикл ошибки

Жизненный цикл ошибки – это путь, который проходит ошибка с момента ее обнаружения до закрытия либо другого терминального статуса. Каждый этап жизни ошибки характеризуется ее статусом. Не все BTS имеют одинаковые названия статусов ошибок и, соответственно, жизненные циклы. Рассмотрим основные из них (рис. 5.1).

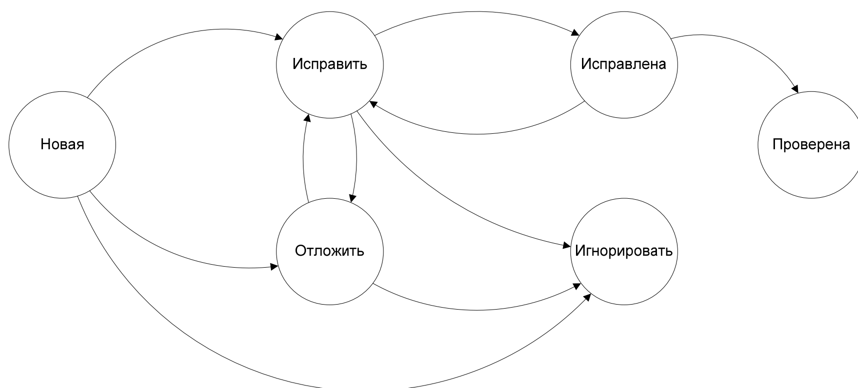

Рис. 5.1. Жизненный цикл ошибки

Жизненный цикл ошибки начинается с ее обнаружения и присвоения статуса *новая* (или найдена, или обнаружена, или submitted). Затем ошибка направляется разработчику для исправления и получает статус *исправить* (или назначить, или assigned). После исправления ошибка получает статус *исправлена* (или fixed). Затем исправленная ошибка проверяется тестировщиком еще раз (как правило в новой версии ПО) и, если она не воспроизводится, то получает статус *проверена* (или закрыта, или verified). На этом жизненный цикл ошибки заканчивается, однако это идеальный случай, который не всегда происходит.

Возможны и другие пути жизненного цикла ошибки:

1. Исправленная ошибка повторяется снова, тогда она отправляется на рассмотрение по новому кругу.
2. Задокументированная ошибка не воспроизводится, тогда ей может быть присвоен статус *игнорировать* (или не воспроизводится, или declined).
3. Возможны также случаи, когда задокументированная ошибка, например, не может быть исправлена в текущей версии, тогда ее откладывают с присвоением статуса *отложена* (или deffered).

Возможны и другие статусы ошибок, определяемые системами документирования ошибок, например, статус *дубликат*, когда в BTS уже задокументирована такая ошибка, или статус *не ошибка*, когда тестировщик ошибочно составил отчет.

5.5. Реализация градации тестов

5.5.1. Тест «на дым» и критерии его непрохождения

Как указывалось выше (п. 4.2), тест на «дым» (в литературе также встречается название приемочный тест) – это быстрая проверка выполнения основной функциональности программы. После приемочного теста делается вывод о пригодности программы для дальнейшего тестирования или нет. Каждая версия разрабатываемого ПО подвергается проверке «на дым», поскольку нет смысла тратить усилия тестировщиков на версию, в которой не работает даже основная функциональность. Критерии непрохождения приемочного теста часто зависят от специфики решаемой задачи и должны быть отражены в тестовом плане, однако можно выделить общие из них:

1) отсутствие каких-либо файлов или компонент, без которых любое тестирование ПО невозможно;

2) сбой программного обеспечения или системы в начале работы, и дальнейшее тестирование невозможно;

3) ошибка в программе, приводящая к сбою в середине работы и делающая дальнейшее тестирование невозможным;

4) достижение определенного процента ошибок, недопустимого для приемочного тестирования, например, 20–25 % тестовых случаев не проходит (как правило, недопустимый процент указывается в тестовом плане).

5.5.2. Позитивный тест

Позитивный тест – это основной вид функционального тестирования, при котором проверяется типичная, логически верная работа программы. Этот вид тестирования проводят в случае пройденного приемочного теста посредством запуска программы и проверки работы каждой функциональности по заранее подготовленным тестовым случаям. Отработанный тестовый случай помечается как «пройденный» или «не пройденный». Для этого в таблице с тестовыми случаями существует отдельная колонка (табл. 4.1). Если тест не пройден, тестировщик составляет отчет об ошибке и сохраняет его в BTS. Если в процессе тестирования обнаруживается ошибка, не предусмотренная тестовыми случаями, то тестировщик также составляет отчет об ошибке и добавляет новый тестовый случай в имеющийся список. Такой подход позволит не потерять возможную ошибку в следующих версиях тестируемой программы.

5.5.3. Негативный тест

Негативный тест проверяет работу программы в непредвиденных и нестандартных ситуациях, например, при некорректно вводимом значении, при незаполненных полях для ввода, при дублировании данных и т. д. Негативный тест часто объединяют с мелкими проверками (например, на разный формат даты или перемещение по элементам интерфейса через табуляцию), тогда он получает название «углубленный тест» (или расширенный, англ., Extended Test). Такой тест проводят на стабильно работающей программе ближе

к окончанию разработки, что связано с экономией времени, поскольку нет смысла проверять нестандартную работу программы, если не работают стандартные ситуации. Негативный тест проводят на основе заранее разработанных тестовых случаев, а также с использованием контрольных перечней (листов проверок, чек-листов, англ., Check List).

Чек-листы – это мини-тестовые случаи, которые создаются для стандартных элементов и полей приложений, применяемых практически в любых проектах, например, тестовых полей, числовых, полей даты, времени и т. д. Поэтому нет смысла для каждого отдельного проекта разрабатывать свои тестовые случаи для таких полей и элементов, а достаточно использовать контрольные перечни, выработанные опытным путем в течение времени каждым тестировщиком, и имеющиеся, как правило, в каждой компании. Рассмотрим некоторые из них.

Проверка одного текстового поля:

– проверить на заполненность (должно быть первоначально пустым или нет);

– проверить на пробелы в начале и в конце текста (если не оговорено в требованиях, то пробелы должны быть опущены);

– проверить на одни пробелы в поле;

– проверить на пробелы в середине текстового поля;

– проверить на специальные символы;

– проверить на символы конкретного языка, если имеется реализация приложения на нескольких иностранных языках;

– проверить на минимальную и максимальную длину поля;

– проверить на работу с кнопками <Shift>, <Insert>, <CapsLock>, а также на выделение текста.

Проверка нескольких текстовых полей:

– проверить по всем пунктам чек-листа для одного текстового поля, а также на сохранение текста, состоящего только из одной строки.

Проверка числовых полей:

– проверить на минимально и максимально допустимое число;

– проверить на отрицательные значения;

– проверить на буквенные символы;

– проверить на 0, 1, –1.

Проверка ListBox:

– проверить на выпадение списка по стрелке и комбинации клавиш <Ctrl> + <A>;

– проверить сортировку в списке (если не оговорено иначе, то должна быть по алфавиту);

– выделенное значение отображается и при сохранении не должно сброситься.

Проверка ComboBox:

– проверить внешний вид бокса;

– проверить на ввод своих значений – они должны отображаться.

Проверка CheckBox:

– проверить работу с помощью мыши и клавиатуры.

Проверка RadioButton:

– проверить на установку позиции по умолчанию;

– проверить выделение поля после его выбора.

Поле валюты:

– проверить на количество знаков для разменной монеты;

– проверить точку или запятую для обозначения целой части.

Поле даты:

– проверить на пустую дату;

– проверить на минимальную и максимальную дату, допустимые для этого поля;

– проверить, чтобы минимальная дата не была больше максимальной (для случая работы с периодом времени);

– проверить на некорректную дату;

– проверить на формат даты (программа должна реагировать на неформатную дату или уметь ее конвертировать). Возможны форматы дат: дд.мм.гггг (Беларусь, Россия, Германия), дд-мм-гггг (Италия, Дания, Нидерланды), дд/мм/гггг (Великобритания, Латинская Америка), мм-дд-гггг (США), мм/дд/гггг (Литва), гггг-мм-дд (Канада, Бельгия, Швеция) и другие.

Кнопки:

– проверить расположение кнопки и ее вид;

– проверить текст на кнопке (нельзя использовать аббревиатуру и сокращения);

– проверить, поместится ли текст на кнопке при переходе на другой иностранный язык;

– проверить на однократное и двойное нажатие на кнопку, а также на табуляцию;

– проверить доступность кнопки согласно требованиям.

Окна:

– проверить на позиционирование окна в зависимости от разрешения экрана;

– проверить цветовую гамму;

– проверить появление скроллинга в необходимых случаях;

– проверить, чтобы все текстовые поля были выравнены, а шрифты были одинаковые.

– проверить, верный ли номер версии стоит в окне About.

5.6. Особенности тестирования standalone и веб-приложений

Все приложения принято делить на две группы: веб-приложения и standalone-приложения. Веб-приложения классифицируются по размеру проекта, по сфере использования и по целевому устройству, а standalone-приложения – по архитектуре, по принципу хранения данных, по рабочей платформе и по функционированию. Общими видами тестирования для всех приложений являются:

1) Функциональное тестирование.

2) Тестирование интерфейса.

3) Тестирование удобства использования.

4) Тестирование производительности (для standalone-приложений учитывают время старта и запуска, время загрузки формы и отчета, для веб-приложений – время отклика страниц).

5) Тестирование безопасности (для standalone-приложений проверяется корректность работы с операционной системой, лицензирование, устойчивость к случайным сбоям и проверка разграничений прав доступа, для веб-приложений – зависимость от браузера, устойчивость к инъекциям, модификации *get* и *post* запросов, изменение информации в URL, прямые запросы в базу, проверка на уязвимость, проверка разграничений прав доступа).

6) Тестирование локализации – это проверка на соответствие локальным стандартам, языковым стандартам и пользовательского интерфейса.

7) Тестирование доступности (включая людей с ограниченными возможностями), реагирование на CAPTCHA, отключение изображений для экономии трафика.

8) Тестирование интеграции с другими приложениями.

Особенными видами тестирования для standalone-приложений являются следующие:

a) кросс-платформенное тестирование. Приложение не должно быть зависимо от операционной системы, если иное не написано в требованиях;

b) тестирование использования ресурсов (утечка памяти);

c) тестирование установки программ и лицензирования. Используются различные способы установки, обращается внимание на типы установок, интерфейс установщика, роли пользователя, язык установки, сообщения в процессе установки, доступность приложения в процессе установки;

d) тестирование механизма обновления;

e) деинсталляция и проверка после этого состояния системы;

f) проверка работы программы на заданной конфигурации и запуск всех поддерживаемых локализаций.

К специальным видам тестирования веб-приложений можно отнести следующие:

a) кроссбраузерное тестирование, т. е. проверка работы приложения в браузерах, предусмотренных в документе требований к программному продукту;

b) тестирование для мобильных устройств. Должны быть специальные версии для этих устройств или проверка на разрешение экрана. Как правило, для такого тестирования используются эмуляторы;

c) поиск поврежденных ссылок и отсутствующих страниц.

6. АВТОМАТИЗАЦИЯ ФУНКЦИОНАЛЬНОГО ТЕСТИРОВАНИЯ

Автоматизация тестирования – это процесс замены ручного тестирования некоторым инструментальным средством. Как правило, этот процесс основывается на ATML-методологии и состоит из следующих этапов:

1) принятие решения об автоматизации тестирования;

2) выбор инструментальных средств тестирования;

3) планирование и проектирование тестирования;
4) выполнение и управление тестированием;
5) оценка результатов тестирования.

Наиболее распространенной формой автоматизации функционального тестирования является тестирование приложений через графический пользовательский интерфейс (англ., Graphic Users Interface, GUI). Популярность такого вида тестирования объясняется двумя факторами: во-первых, приложение тестируется тем же способом, которым его будет использовать человек, во-вторых, можно тестировать приложение, не имея при этом доступа к исходному коду.

На сегодняшний день для автоматизации функционального тестирования через GUI на рынке представлено множество инструментов, среди которых можно выделить SilkTest, Win Runner, Rational Robot, Test Complete, QTP, Watir, Sahi, MS Coded UI и, конечно же, семейство Selenium. Каждый из приведенных инструментов обладает определенными преимуществами и недостатками, а также спецификой применения. Например, TestComplete поддерживает различные виды автоматизации тестирования, включая модульное, функциональное и нагрузочное, однако является очень дорогим. Но для учебных целей существует возможность скачать демонстрационную версию на 30 дней с официального сайта производителя [13]. Семейство Selenium, наоборот, является бесплатным, но позволяет тестировать только веб-приложения. Общим свойством всех инструментов является получение тестового автоматизированного скрипта.

Если рассматривать области применения автоматизации, то наибольший эффект ее применения наблюдается с тестами, которые необходимо проводить большое количество раз, на разных наборах операционных систем и браузеров, при работе с большим количеством данных и для имитации большого количества пользователей, при длинных сценариях, с труднодоступными местами в системе (back-end процессы, работа с лог-файлами, запись в базу данных), при проверке данных, требующих точных математических расчетов. Также выгодно использовать автоматизацию для регрессионного тестирования (набора тестов, которые нужно повторять снова и снова, чтобы быть уверенным, что приложение работает корректно, несмотря на вносимые изменения), для приемочного и углубленного тестов, чтобы проверять основную функциональность работы программы, и объемные негативные проверки. Следует добавить, что

автоматизацию следует применять только на стабильно работающем программном продукте.

6.1. Достоинства и недостатки автоматизации функционального тестирования

Среди достоинств автоматизации функционального тестирования рассмотрим следующие:

1. Автоматизированные тесты могут прогоняться множество раз.
2. Большая скорость выполнения теста по сравнению с ручным.
3. Протекают каждый раз одинаково.
4. Повышается качество регрессионного тестирования.
5. Повышается качество проверки на совместимость.
6. Возможна автоматизация однообразных и однотипных задач.
7. Возможность прогона тестов ночью и в выходные дни.
8. Автоматизированные испытания генерируют отчеты и журналы о выполнении.
9. Уменьшаются временные затраты на формирование статистических данных о проекте.
10. Возможно проведение нагрузочного тестирования, а также тестирование под разными серверами для каждой отдельной конфигурации и на разных иностранных языках.
11. Исключение ошибок, связанных с человеческим фактором.
12. Управление несколькими машинами одновременно (распределенный тест).
13. Перезагрузка машины в другую операционную систему и вход под разными пользователями.
14. Взаимодействие с базами данных напрямую.
15. Поиск поломанных ссылок на веб-страницах.
16. Создание теста на одном браузере, а запуск его на другом.
17. Освобождение человека от рутинной работы с возможностью заняться более интересными и креативными задачами.

Недостатков у автоматизированного тестирования не так уж и много. Обычно принято указывать, что разработка и отладка тестов требует больших затрат времени и средств, а также, что автоматизированные тесты не находят новых ошибок (только те, что тестировщик может сам предусмотреть, ведь, выполняя тест вручную, человек может обратить внимание на некоторые детали и, проведя

несколько дополнительных операций, найти ошибку, а автоматизированный скрипт этого сделать не может). К недостаткам можно еще отнести и стоимость инструмента для автоматизации: в случае, если используется лицензионное ПО, его стоимость может быть достаточно высока. Свободно распространяемые инструменты, как правило, отличаются более скромным набором функциональности и меньшим удобством работы.

Большой список достоинств и мелкий недостатков может произвести впечатление идеального подхода к тестированию через ее автоматизацию. Развеять этот миф поможет перечень необоснованных ожиданий от автоматизации тестирования:

1. Автоматизировать можно все что угодно (ложь, поскольку нельзя автоматизировать тесты, где нельзя обойтись без участия человека, например, тестирование удобства использования).

2. Можно обнаружить бо́льшее количество ошибок (ложь, поскольку автоматизированные тесты работают тоже по тестовым случаям, разработанным тестировщиком).

3. Можно исключить или значительно сократить ручное тестирование (следствие из п. 1).

4. Возможно 100 % покрытие функциональности (следствие из п. 1).

5. Все необходимое тестирование может выполнять одно инструментальное средство (средства автоматизации имеют свою специфику, поэтому одним инструментом все покрыть не получится).

6. Временной график тестирования сократится (по статистике, на разработку одного автоматизированного тестового случая уходит до 15 раз больше времени, чем на прохождение его вручную).

7. Автоматизация недорога (для проведения автоматизации тестирования необходимы квалифицированные специалисты с немалой заработной платой, а также большинство инструментов являются платными).

8. Средства автоматизации просты в использовании (во-первых, чтобы научиться работе с этими инструментами необходимо пройти специальное обучение, а во-вторых, инструменты автоматизации довольно капризны в использовании, и, зачастую, причины не прохождения тестов очень трудно определить).

6.2. Требования к автоматизированным тестам

Для того чтобы начинать разрабатывать автоматизированные тесты, необходимо определиться со следующими вопросами:

– Какую функциональность нужно автоматизировать?

– Какие действия необходимы для проведения теста?

– Где брать тестовые данные?

– Каков ожидаемый результат теста?

– Каковы критерии успешного прохождения теста?

Выделим основные требования к автоматизированным тестам:

1) Тест должен тестировать сам, т. е. не только эмулировать действия пользователя с программой, но и проводить проверки.

2) Тест должен выдавать результат в таком виде, чтобы сразу можно было понять, где ошибка.

3) Один тест должен проверять только одну функциональность либо один случай использования функциональности.

4) Каждый тест должен быть независимым (т. е. выполняться независимо от других тестов).

5) Тест должен запускаться на разных платформах и в разных браузерах.

6) Наличие документированных требований или пояснений к запуску теста.

7) Тест должен одинаково хорошо работать на быстрой машине и медленной.

6.3. Методы автоматизации функционального тестирования

6.3.1. Метод Play&Record

Метод Play&Record (в литературе также встречается название Record&Playback) – это возможность работать в режиме записи действий, совершаемых над приложением, и повторное воспроизведение их автоматически. Записанные действия преобразуются в программный код на языке программирования, который встроен в инструмент автоматизации. Последовательность шагов в этом методе такова:

1. Из инструмента автоматизации происходит захват тестируемого приложения.

2. Включается запись действий.

3. Выполняются требуемые действия над программой.
4. Останавливается запись.
5. Включается проигрывание записанных действий.

Записанные действия отображаются в виде программного кода на встроенных языках программирования, в качестве которых могут быть как собственные языки инструментов, например, 4Test для инструмента SilkTest [14], так и распространенные языки программирования, например, C# или JavaScript в TestComplete. В полученный таким образом скрипт можно вносить изменения, проверки, циклы, формировать логику теста и т. д. К недостаткам метода Play&Record можно отнести следующие:

1) Скрипты, получаемые этим методом, содержат фиксированные значения, которые должны быть изменены каждый раз, когда в приложении что-то меняется.

2) Стоимость, связанная с поддержкой автоматизированных скриптов, полученных по этому методу, астрономическая и неприемлемая, поскольку их приходится часто переписывать.

3) Полученные скрипты не надежны и часто не работают, даже если тестируемое приложение не изменялось. Причинами здесь могут быть различные всплывающие окна, сообщения или другие события, которые происходили либо не происходили в момент записи теста, и, соответственно, которые будет ожидать скрипт во время повторного воспроизведения либо неожиданно произойдут для скрипта.

4) Если тестировщик совершает ошибку, например, во время ввода данных, тест должен быть записан заново.

6.3.2. Метод функциональной декомпозиции

Главный принцип метода заключается в приведении всех тестовых случаев к некоторым функциональным задачам, которыми могут быть:

– навигация (например, доступ к странице заказа из главного меню);
– бизнес-функции (например, оформление заказа);
– проверка данных (например, проверка состояния заказа);
– возврат к навигации.

Таким образом, самый верхний уровень скриптов представляет собой скрипт, содержащий серии вызовов одного или нескольких

скриптов для конкретных тестовых случаев. Скрипты этих тестовых случаев вызывают необходимые скрипты бизнес-логики. Скрипты утилит могут вызываться в любом месте скрипта, где это необходимо.

6.3.3. Метод Data-driven

Метод Data-driven (англ., Data Driven Testing, DDT, тестирование, управляемое данными) является продолжением метода функциональной декомпозиции. Отличие состоит в том, что данные для тестов выносятся за пределы кода скрипта, как правило, в Excel-таблицу. Например, для проверки авторизации может быть создана таблица, которая в каждой строке хранит имя пользователя, пароль и ожидаемый результат. Тогда скрипт обращается к таблице с тестовыми данными и поочередно начинает их перебирать, выполняя нужные действия теста и сравнивая фактические результаты с ожидаемыми. На основании проверок формируется журнал о прохождении тестов.

6.3.4. Метод Keyword-driven

Метод Keyword-driven или тестирование, управляемое ключевыми словами, можно рассматривать как продолжение и модификацию метода Data-driven. Тестовый сценарий здесь представлен в виде электронной таблицы, содержащей в одной из колонок специальные ключевые слова, а в остальных колонках могут быть тестовые данные, ожидаемые результаты и другая необходимая информация. Ключевое слово является, как правило, функцией некоторой библиотеки и реализует предписанную последовательность действий. Управляющий скрипт обращается к таблице тестового сценария, находит ключевое слово в нем, считывает из той же строки тестовые данные и выполняет действие, описанное этим ключевым словом. Полученный результат далее сравнивается с ожидаемым, и генерируется отчет о прохождении теста. Затем скрипт находит следующее ключевое слово, и так до конца документа. Таким образом, реализуется трехслойная модель автоматизации:

– 1-й слой – библиотека ключевых слов;

– 2-й слой – тестовый сценарий в виде электронной таблицы;

– 3-й слой – управляющий скрипт.

К преимуществам такого подхода можно отнести следующие:

1) тестовые сценарии могут быть реализованы в формате электронной таблицы, которая содержит все данные для ввода и проверки, таким образом, сценарий пишется только один раз;

2) тестовые сценарии могут быть написаны в любом формате, который поддерживает конвертацию tab-delimited или comma-delimited;

3) создавать тестовые сценарии может любой тестировщик, даже не владеющий навыками автоматизированного тестирования;

4) если тестовые сценарии уже существуют в каком-либо формате, их не сложно перевести в формат электронных таблиц;

5) скрипты-утилиты, созданные для одного приложения, могут быть использованы с небольшими изменениями для тестирования других программ.

6.4. Семейство Selenium и его возможности

В рамках семейства Selenium имеется несколько программных продуктов, которые могут быть использованы в процессе функционального тестирования. Например, Selenium Server позволяет организовать удаленный запуск браузера, при помощи Selenium Grid можно построить кластер из Selenium-серверов. Также имеется «рекордер» Selenium IDE, который умеет записывать действия пользователя и генерировать код. Однако главным в семействе является Selenium WebDriver, который представляет собой библиотеку, позволяющую разрабатывать программы, управляющие поведением браузера. Для группировки и запуска тестов, а также для генерации отчетов о тестировании при таком подходе используются фреймворки тестирования (п. 3.7). Например, JUnit или TestNG для Java, NUnit или MSTest для .Net, RSpec или Cucumber для Ruby. Разработка тестов ведется, соответственно, в средах Eclipse, Intellij IDEA, MS Visual Studio, RubyMine и так далее.

С целью следования принципам повторного использования тестового кода и функциональной декомпозиции, программный каркас для автоматизации тестирования с использованием Selenium WebDriver разделяется на несколько слоев (рис. 6.1): слой тестов, бизнес-слой, слой для работы с пользовательским интерфейсом и слой вспомогательных библиотек.

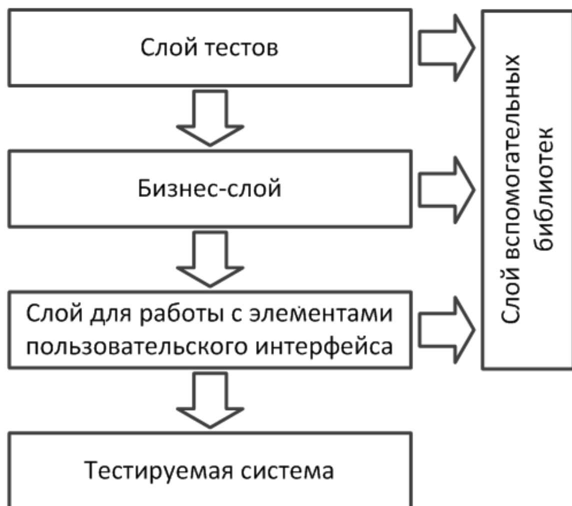

Рис. 6.1. Схема каркаса для автоматизации тестирования
с использованием Selenium WebDriver

Слой тестов содержит описания сценариев в программном коде. Бизнес-слой описывает наиболее часто используемые сложные операции, которые специфичны для тестируемой системы. Слой для работы с элементами пользовательского интерфейса реализует взаимодействие тестирующего кода с тестируемой системой через пользовательский интерфейс. Данный слой в случае веб-приложений можно реализовать на основе программного интерфейса WebDriver.

6.5. Проблемы внедрения автоматизации тестирования программного обеспечения

В ходе внедрения автоматизации тестирования существует риск невозврата вложений в этот процесс. Для оценки возврата инвестиций используется универсальный коэффициент окупаемости инвестиций *ROI* [15]. Этот показатель является одним из основных способов измерения эффективности вложений и рассчитывается по следующей формуле:

$$ROI = прибыль \,/\, затраты. \quad (6.1)$$

Для успешного возврата инвестицией показатель *ROI* должен быть равен либо больше единицы. При внедрении автоматизации необходимо рассчитать, насколько автоматизация эффективней, чем ручное тестирование. Формула с учетом затрат принимает следующий вид [15]:

$$ROI_{автоматизации} = (X - Y) / Y, \tag{6.2}$$

где X – затраты на ручное тестирование;

Y – затраты на автоматизацию тестирования и его поддержку.

В данном случае значение успешности инвестиций начинается от нуля. Поэтому перед внедрением автоматизации можно произвести прогнозирование с оценкой *ROI*, после чего делать вывод о внедрении или не внедрении автоматизации тестирования. Также необходимо учитывать следующие факторы:

– технологии, на которых построена система, а именно, наличие инструментов для их автоматизации;

– потенциальную изменяемость интерфейсов системы, на которых будет основываться автоматизация;

– специфику проверок системы, а именно, их пригодность для автоматизации;

– потенциальный объем автоматизируемых тестов;

– планируемую частоту проведения тестирования;

– необходимость разработки специализированных инструментов для автоматизации;

– рост затрат на поддержку автоматизированных тестов;

– сроки разработки.

При оценке объема автоматизации и планируемого покрытия тестами, можно прибегнуть к разбиению тестов по уровням работы с системой и по видам тестирования. В [16] приводится пример соотношения автоматизированных тестов в различных видах тестирования на проекте по разработке ПО. Данный пример, представленный на рис. 6.2, называется пирамидой автоматизации тестов. Цель следования описанному соотношению – обеспечить максимально возможное покрытие кода при минимальных затратах.

Проблема внедрения автоматизации в короткие сроки решается использованием существующих программных каркасов для автоматизации тестирования, перенесением опыта автоматизации с других

проектов, внедрением процесса разработки на основе тестов или на основе поведения.

Рис. 6.2. Пирамида автоматизации тестов [16]

7. ОТЛАДКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

7.1. Методика отладки

Отладка – это процесс обнаружения причин возникновения ошибок и их последующего исправления. Отладка на некоторых проектах занимает до 50 % времени разработки. Это одна из самых сложных частей программирования. По психологическим причинам отладка – это самая нелюбимая работа программиста. Многие программисты стремятся как можно быстрее написать программу, а затем обнаружить ошибки путем многократного ее выполнения с разнообразными тестовыми данными без их внимательного анализа и тщательного проектирования. Это приводит к значительным затратам времени при установлении причин ошибок и их локализации, заметному снижению надежности программ, что в дальнейшем проявляется на этапе сопровождения программного изделия. Кроме того, известным фактом является внесение новых ошибок при отладке.

В методике отладки принято выделять две части:

– нахождение причины возникновения ошибки (90 % времени отладки, как правило, уходит на эту часть);

– исправление причины ошибки.

Наиболее эффективным считается *обобщенный научный подход* нахождения причин ошибок. Он состоит из следующих шагов:

1) сбор данных через повторяющиеся эксперименты;
2) создание гипотезы, отражающей максимум доступных данных;
3) разработка эксперимента для проверки гипотезы;
4) подтверждение или опровержение гипотезы;
5) итерация предыдущих шагов (при необходимости).

Тогда, с учетом обобщенного научного подхода, методику отладки можно описать в виде последовательности следующих этапов:

1. Стабилизация ошибки (1-й шаг обобщенного научного подхода).
2. Обнаружение точного места ошибки (2–4 шаги обобщенного научного подхода).
3. Исправление ошибки.
4. Проверка исправления.
5. Поиск похожих ошибок.

При стабилизации ошибки рекомендуется свести тест к минимально возможному. Если проблема возникает не стабильно, то ее практически невозможно диагностировать. Однако статистика показывает, что нестабильные ошибки обычно связаны с проблемами инициализации или висячими указателями.

В качестве рекомендаций по обнаружению точного места ошибки можно предложить следующие:

– используйте все доступные данные при выдвижении гипотез;

– воспроизведите ошибку различными способами;

– используйте результаты негативных тестов;

– проведите «мозговой штурм» в поисках новых гипотез;

– сужайте подозрительные фрагменты кода;

– внимательнее отнеситесь к тем методам и функциям, где уже встречались ошибки;

– проверяйте недавние исправления;

– проводите интеграцию постепенно;

– ищите типичные ошибки;

– обсудите проблему с коллегами;

– отдохните от проблемы.

В настоящее время синтаксические ошибки все реже приходится искать вручную, однако, если все же такое произошло, то не доверяйте номеру строки с ошибкой, на которую указывает компилятор, смотрите ±1 строку, также внимательнее отнеситесь к предупреждениям компилятора; используйте метод «разделяй и властвуй»; ищите лишние комментарии или кавычки.

Когда месторасположение ошибки определено, можно переходить к исправлению. Здесь рекомендации могут быть следующими:

– разберитесь в проблеме, прежде чем ее исправлять;

– разберитесь в программе, а не только в проблеме;

– убедитесь в правильности диагноза до исправления;

– не торопитесь при внесении исправлений;

– сохраняйте исходную версию кода;

– исправляйте проблему, а не ее симптом;

– вносите исправления только тогда, когда вы уверены;

– вносите исправления только по одной штуке за раз;

– проверяйте свои исправления;

– ищите похожие ошибки.

7.2. Методы отладки

Большинство ошибок можно обнаружить по косвенным признакам посредством тщательного анализа текстов программ и результатов тестирования без получения дополнительной информации. При этом используют различные методы:

– ручного тестирования;

– индукции;

– дедукции;

– обратного прослеживания.

Метод ручного тестирования (также встречается название метод «грубой силы») – это самый простой и естественный способ, наиболее распространенный и традиционно используемый программистами. Всесторонний анализ за столом исходного кода и алгоритма программы, выходных результатов и сообщений компилятора, вы-

полнение тестируемой программы вручную, используя тестовый набор, при работе с которым была обнаружена ошибка. Для повышения эффективности отладки в текст программы включают операторы отладочного кода, которые регистрируют результаты использования конкретного оператора. Довольно часто проводят распечатку выборочных значений переменных. После завершения отладки отладочные операторы можно оставить в виде комментариев для дальнейшего использования на этапе сопровождения.

Данный метод эффективен, однако трудно применим для больших программ, программ со сложными вычислениями, а также в тех случаях, когда ошибка связана с неверным представлением программиста о выполнении некоторых операций. Данный метод часто используют как составную часть других методов отладки.

Метод индукции основан на тщательном анализе симптомов ошибки, используя стратегию движения от частного к общему. Если компьютер просто «зависает», то фрагмент проявления ошибки вычисляют, исходя из последних полученных результатов и действий пользователя. Полученную таким образом информацию организуют и тщательно изучают, просматривая соответствующий фрагмент программы. В результате этих действий выдвигают гипотезы об ошибках, каждую из которых проверяют. Если гипотеза верна, то детализируют информацию об ошибке, иначе – выдвигают другую гипотезу. Организуя данные об ошибке, целесообразно записать все, что известно о ее проявлениях, причем фиксируют как ситуации, в которых фрагмент с ошибкой выполняется нормально, так и ситуации, в которых ошибка проявляется. Если в результате изучения данных никаких гипотез не появляется, то необходима дополнительная информация об ошибке. Дополнительную информацию можно получить, например, в результате выполнения схожих тестов. В процессе доказательства пытаются выяснить, все ли проявления ошибки объясняет данная гипотеза, если не все, то либо гипотеза не верна, либо ошибок несколько.

Метод дедукции реализует формирование множества причин, которые могли бы вызвать данное проявление ошибки. Затем анализируются причины и исключаются те, которые противоречат имеющимся данным. Если все причины исключены, то следует выполнить дополнительное тестирование исследуемого фрагмента. В противном случае наиболее вероятную гипотезу пытаются доказать.

Если гипотеза объясняет полученные признаки ошибки, то ошибка найдена, иначе – проверяют следующую причину.

Метод обратного прослеживания (или инверсное прослеживание логики программы) наиболее эффективен для небольших программ и направлен на анализ логики выполнения программы в обратном направлении. Отладка начинается с точки программы, где обнаружен неверный ожидаемый результат. На основе полученных в этой точке значений переменных, необходимо определить, исходя из логики программы, какие результаты должны были быть при правильной работе программы. Последовательное продвижение к началу программы позволяет достаточно быстро и точно определить место (и причину возникновения) ошибки, т. е. место между оператором, где результат выполнения программы соответствовал ожидаемому, и оператором, в котором появились расхождения. Процесс продолжают, пока не обнаружат причину ошибки.

7.3. Психологические аспекты отладки

Отладка – это психологически сложный процесс, поскольку приходится искать ошибки в собственном программном коде, приходится думать последовательно и логично, перестраиваться из разработчика в тестировщика.

В процессе отладки выделяют следующие психологические установки:

1) при использовании цикла *while* зачастую понимают, что условие выполняется постоянно, а не в начале или в конце цикла;

2) ошибки в присваиваниях найти намного сложнее, чем ошибки взаимодействия;

3) при создании блоков часто имеют место подразумеваемые скобки, что приводит к неверной работе алгоритма (например, в примере ниже после оператора условия выполнится только одно присвоение):

```
if (x<y) then
        swap :=x;
        x:=y;
        y:=swap;
```

4) при работе с переменными возникает такое понятие, как психологическое расстояние – это легкость различия слов и, соответ-

ственно, переменных людьми. Примеры разного психологического расстояния приведены в табл. 7.1.

Таблица 7.1

Примеры психологического расстояния

1-я переменная	2-я переменная	Психологическое расстояние
STOPT	STOPPT	Почти нуль
SHIFTRF	SHIFTRT	Малозаметное
CLAIMS 1	CLAIMS 2	Небольшое
GCOUNT	CCOUNT	Небольшое
PRODUCT	SUM	Большое

7.4. Средства отладки

Большинство современных сред программирования включают средства отладки, которые обеспечивают максимально эффективную отладку. Они позволяют:

– выполнять программу по шагам, причем как с заходом в подпрограммы, так и выполняя их целиком;

– предусматривать точки останова;

– выполнять программу до оператора, указанного курсором;

– отображать содержимое любых переменных при пошаговом выполнении;

– отслеживать поток сообщений и т. п.

Применять интегрированные средства отладки в рамках среды достаточно просто. Используют разные приемы в зависимости от проявлений ошибки. Если получено сообщение об ошибке, то сначала уточняют, при выполнении какого оператора программы оно получено. Для этого устанавливают точку останова в начало фрагмента, в котором проявляется ошибка, и выполняют операторы в пошаговом режиме до проявления ошибки.

Если получены неверные результаты теста, то локализовать ошибку обычно сложнее. В этом случае сначала определяют фрагмент, при выполнении которого получаются неправильные результаты. Для этого последовательно проверяют интересующие значения в узловых точках. Обнаружив значения, отличающиеся от ожидаемых,

по шагам трассируют соответствующий фрагмент до выявления оператора, выполнение которого дает неверный результат. Для уточнения природы ошибки возможен анализ машинных кодов, флагов и представления программы и значений памяти в шестнадцатеричном виде.

Кроме отладчиков можно выделить следующие средства отладки:

1. Специальные средства расширения языка программирования для контроля типов и диапазонов значений данных, обработки исключительных ситуаций и т.д.
2. Средства для печати значений используемых переменных при аварийном завершении программы.
3. Пакеты программ для прослеживания потоков управления и данных в программе, контроля индексов и регистрации вызовов программ.
4. Генераторы тестовых данных, формирующие требуемые тестовые наборы.
5. Специальные онлайн-отладчики, обеспечивающие автоматизацию рестартов, остановов и прерываний программы, просмотр работы отдельных операторов и т. п.
6. CASE-средства для построения, например, схем потоков данных, модулей данных, схем алгоритмов.
7. Автоматизированные рабочие места программистов, включающие большинство из перечисленных средств.

ЛИТЕРАТУРА

1. Майерс, Г. Искусство тестирования программ / Г. Майерс. – Москва : Финансы и статистика, 1982. – 186 с.
2. Канер, С. Тестирование программного обеспечения / С. Канер. – Киев : ДиаСофт, 2001. – 544 с.
3. International Software Testing Qualifications Board [Электронный ресурс]. – Режим доступа – http://www.istqb.org/downloads/category/34-foundation-level-e-books.html. – Дата доступа: 01.02.2017.
4. Куликов, С. Тестирование программного обеспечения. Базовый курс / С. Куликов [Электронный ресурс]. – Режим доступа: http://svyatoslav.biz/software_testing_book. – Дата доступа: 01.02.2017.
5. Савин, Р. Тестирование Дот Ком / Р. Савин. – Москва : Дело, 2007. – 312 с.
6. Калбертсон, Р. Быстрое тестирование / Р. Калбертсон, К. Браун, Г. Кобб. – Москва : Вильямс, 2002. – 384 с.
7. Липаев, В. В. Тестирование программных средств / В. В. Липаев. – Москва : Финансы и статистика, 1999. – 264 с.
8. Макгрегор, Д. Тестирование объектно-ориентированного программного обеспечения / Д. Макгрегор, Д. Сайкс. – Киев : ДиаСофт, 2002. – 348 с.
9. Кармайкл, Э. Быстрая и качественная разработка программного обеспечения / Э. Кармайкл, Д. Хейнвуд. – Москва : Вильямс, 2003. – 400 с.
10. Watson, A. H. Structured Testing: A testing methodology using the cyclomatic complexity metric / A. H. Watson, T. J. McCabe [Электронный ресурс]. – Режим доступа: www.mccabe.com/pdf/mccabe-nist235r.pdf. – Дата доступа: 01.02.2017.
11. Липаев, В. В. Тестирование компонентов и комплексов программ. Учебник / В. В. Липаев. – М.: Синтег, 2010. – 400 с.
12. Бейзер, Б. Тестирование «черного ящика» / Б. Бейзер. – Киев : ДиаСофт, 2002. – 326 с.
13. Официальный сайт производителя TestComplete [Электронный ресурс]. – Режим доступа: https://smartbear.com/product/testcomplete/overview. – Дата доступа: 01.02.2017.
14. Винниченко, И. В. Автоматизация процессов тестирования / И. В. Винниченко. – СПб. : Питер, 2005. – 203 с.

15. FYI On ROI: A Guide To Calculating Return On Investment By Andrew Beattie // Investopedia [Электронный ресурс]. – Режим доступа: http://www.investopedia.com/articles/basics/10/guide-to-calculating-roi.asp. – Дата доступа: 01.02.2017.

16. Криспин, Л. Гибкое тестирование: практическое руководство для тестировщиков ПО и гибких команд / Л. Криспин, Д. Грегори. – Москва : «Вильямс», 2010. – 464 с.

17. Agans, D. J. Debugging: The 9 indispensable rules for finding even the most elusive software and hardware problems / D. J. Agans. – NY, 2002. – 186 с.

Учебное издание

ПОПОВА Юлия Борисовна

ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Пособие

Редактор *Е. В. Герасименко*
Компьютерная верстка *Н. А. Школьниковой*

Подписано в печать 23.03.2020. Формат 60×84 $^{1}/_{16}$. Бумага офсетная. Ризография.
Усл. печ. л. 3,89. Уч.-изд. л. 3,05. Тираж 500. Заказ 487.

Издатель и полиграфическое исполнение: Белорусский национальный технический университет.
Свидетельство о государственной регистрации издателя, изготовителя, распространителя печатных изданий № 1/173 от 12.02.2014. Пр. Независимости, 65. 220013, г. Минск.