Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»

В. Н. ПЕРОЦКАЯ Д. А. ГРАДУСОВ

ОСНОВЫ ТЕСТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

Владимир 2017

УДК 004.45
ББК 32.972
П27

Рецензенты:
Доктор технических наук, профессор
зав. кафедрой информационных систем и программной инженерии
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых
И. Е. Жигалов

Кандидат физико-математических наук доцент
кафедры гуманитарных и естественно-научных дисциплин
Владимирского филиала Российского университета кооперации
В. В. Красильщиков

Печатается по решению редакционно-издательского совета ВлГУ

Пероцкая, В. Н. Основы тестирования программного
П27 обеспечения : учеб. пособие / В. Н. Пероцкая, Д. А. Градусов ; Владим. гос. ун-т им. А. Г. и Н. Г. Столетовых. – Владимир : Изд-во ВлГУ, 2017. – 100 с.
ISBN 978-5-9984-0777-2

Приведены общие сведения о тестировании программных систем, принципах тестирования, а также примеры ошибок, повлекших за собой серьезные материальные потери и человеческие жертвы. Подробно рассмотрены методики разработки тестовых сценариев. Приведено детальное описание различных примеров применения методик написания тестов. Проанализированы основные аспекты и инструменты автоматизации тестирования, а также основы управления процессом тестирования информационных систем.

Предназначено для студентов 3-го курса, обучающихся по специальности 09.03.03 «Прикладная информатика» дневной и заочной форм обучения.

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС ВО.

Табл. 18. Ил. 21. Библиогр.: 15 назв.

УДК 004.45
ББК 32.972

ISBN 978-5-9984-0777-2

ВВЕДЕНИЕ

Значимость тестирования программных продуктов давно не требует доказательств. Любой пользователь желает работать с системой, которая выполняет свои функции корректно и позволяет человеку достичь ожидаемого результата как можно быстрее и проще.

Очевидно, что программные продукты низкого качества не вызывают доверия у пользователей, поэтому в конечном итоге они исчезнут с рынка. Их место призваны занять более качественные продукты. Тестирование программных продуктов позволяет не только удовлетворить ожидания конечного пользователя, но и избежать значительных убытков для владельца и распространителя, а также создателей программного продукта.

Грамотное тестирование – это залог успешной работы как крупных программных систем, так и небольших коробочных программных продуктов. Важно не просто проверить продукт на соответствие требованиям, а убедиться, что он подходит для конечных пользователей и способен выполнять заявленные функции. Необходимо организовать тестирование как процесс, в который входит не только непосредственное выполнение тестовых сценариев, но и планирование работ, распределение трудовых ресурсов, формирование тестовой модели, контроль за процессом тестирования, сбор метрик, анализ критериев выхода, а также активности по завершении тестирования.

Для успешного построения процесса тестирования используются многочисленные инструменты. Кроме того, наиболее трудоемкие и часто повторяющиеся тестовые сценарии могут быть автоматизированы, что, как правило, позволяет значительно сократить временные затраты на проверку достаточно больших и сложных частей функционала системы.

В учебном пособии подробно рассмотрены принципы и основы тестирования программных систем. Особое внимание уделено различным методикам разработки тестовых сценариев, приведены наглядные примеры их использования. Рассмотрены основные инструменты автоматизации тестирования, принципы их выбора. Кроме того, уделено внимание стандартам тестирования программных продуктов, а также построению процесса тестирования и управления им.

1. ПОНЯТИЕ «ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ»

Программное обеспечение, которое функционирует внутри различных систем, используемых в современном мире, является неотъемлемо важной частью нашей жизни. Мы используем вычислительные машины, станки и оборудование, всевозможные гаджеты и многое другое, что не способно приносить пользу, если не имеет в своей основе некой программной части.

Очень часто потребители и пользователи сталкиваются с тем, что устройство не работает так, как ему следовало бы работать, дает сбои и не приносит ожидаемого результата. В истории программного обеспечения существует масса примеров того, как **некачественное программное обеспечение приводило к огромным финансовым потерям и человеческим жертвам.**

Рассмотрим несколько таких примеров, чтобы наглядно продемонстрировать масштаб проблемы.

Therac-25 представляла собой компьютеризированную машину для радиационной терапии, построенную компанией Atomic Energy of Canada Limited (AECL).

Therac-25 представляла собой усовершенствованную модель Therac-20. Она была способна испускать фотоны или электроны с энергией 25 МэВ с возможностью переключения уровней. Она была меньше, имела больше возможностей и была легче в использовании. Therac-25 была сконструирована так, чтобы компьютерное управление было более полным, чем у предшественников. **Программное обеспечение, разработанное для Therac-25, было способно контролировать состояние оборудования и управлять им. Поэтому было решено удалить аппаратные средства безопасности и полагаться в этом вопросе на программное обеспечение.**

Therac-25 поступила в продажу в конце 1982 года: 11 таких машин были установлены в Северной Америке, 5 – в США и 6 – в Канаде. **Шесть несчастных случаев с большими передозировками произошли между 1985 и 1987 годами.**

В июне 1985 года в г. Кенстоун, округ Мариетта, штат Джорджия женщина в возрасте 61 года была направлена в онкологический

центр для дополнительного лечения аппаратом Therac-25 после хирургического удаления молочной железы [1]. Как считается, пациентка получила одну или две дозы радиации от 15 000 до 20 000 рад (поглощенная доза радиации). Для сравнения: типичная разовая терапевтическая доза радиации составляет до 200 рад. Кенстоунская клиника использовала Therac-25 с 1983 года без происшествий. Техники и фирма AECL не поверили, что эта проблема могла быть вызвана Therac-25. В конечном итоге пациентка потеряла грудь, а также возможность пользоваться руками и плечами из-за радиационного поражения.

Клиника в Хэмилтоне использовала Therac-25 в течение шести месяцев до инцидента с передозировкой. Сорокалетняя женщина поступила в клинику на 24-й сеанс лечения аппаратом Therac-25. Аппарат отключился через пять секунд после того, как оператор запустил его. Операторы были знакомы с частыми неполадками машины. Эти неполадки, вероятно, не имели серьезных последствий для пациента. Поскольку машина показывала, что облучение не было произведено, оператор попытался повторить его. Были произведены пять попыток. После пятой попытки был вызван техник, не обнаруживший проблем в аппарате.

Об инциденте сообщили в AECL, но воспроизвести неполадку и сделать заключение о причинах ошибки в работе Therac-25 не удалось. Однако благодаря этому сообщению фирма AECL обнаружила некоторые слабости конструкции и потенциальные механические проблемы в позиционировании поворотной платформы Therac-25, были сделаны исправления. Пациентка умерла через пять месяцев. Результаты вскрытия показали, что смерть наступила от рака, а не от передозировки радиации. Однако вскрытие также выявило серьезные поражения бедра, вызванные радиационным воздействием. Как было позже определено, пациентка получила дозу порядка 13 000 – 17 000 рад.

За короткую жизнь Therac-25 было обнаружено два программных дефекта:

- логическая ошибка в обновлении параметров, когда оператор менял состояние машины;
- проверка безопасности не срабатывала, когда 8-битный счетчик переполнялся и достигал нуля каждые 256 итераций [1].

Однако с точки зрения безопасности систем самое уязвимое место – это доверие к программному обеспечению. Очевидно, что тот же дефект, который вызвал передозировки при работе Therac-25, присутствовал и у Therac-20. **Та же ошибка при воздействии Therac-20 приводила к отключению машины без передозировки, поскольку в Therac-20 применялись независимые аппаратные устройства безопасности, которые предотвращали ее.**

Ошибки, связанные с зависимостью от скорости, очень трудно обнаружить и воспроизвести. В первом случае две причины точно должны были присутствовать, чтобы активировать ошибку:

1) оператор должен изменить параметры режима и уровня энергии;

2) оператор должен внести изменения в течение восьми секунд.

В случае второго дефекта все зависит от случайности. Ошибка активируется, если клавишу нажимают в тот момент, когда счетчик достигает нуля. Вот почему, несмотря на имеющийся с самого начала функционирования продукта дефект, было отмечено только шесть несчастных случаев. Если бы ошибка была более явной и ее было бы легче запустить, несомненно, она была бы выявлена в AECL в ходе обычных процедур тестирования и проверки качества, и тогда о ней никогда не узнало бы общество. **Если ошибку трудно активировать, ее также будет трудно выявить в ходе обычного процесса тестирования. Трудно найти ошибки, если невозможно их воспроизвести.**

Ракета-носитель Ariane 5 была ответом Европейскому космическому агентству (European Space Agency), пытавшемуся стать лидером в запусках ракет на коммерческом космическом рынке. Стоившая 7 миллиардов долларов и строившаяся в течение 10 лет, Ariane 5 могла вывести на орбиту два трехтонных спутника [1]. Во время первого полета утром 4 июня 1996 года ракета Ariane 5 взорвалась через 40 с после старта. Анализ данных полета быстро показал, что ракета вела себя нормально до того момента, когда она вдруг **отклонилась от курса и самоуничтожилась.** Погода в то утро была приемлемой, так что она не могла оказать влияние. Полетные данные также показали,

что активная система и первичная инерционная система ориентировки (Inertial Reference System), которые влияли на управление соплами твердотопливного ускорителя, более или менее одновременно отказали прямо перед разрушением ракеты.

После инцидента была сформирована комиссия по расследованию. В её распоряжение были предоставлены телеметрические данные ракеты, данные о траектории с радиолокационных станций, данные оптических наблюдений за ракетой и упавшими обломками, а также восстановленная инерционная система ориентировки. Кроме того, комиссия располагала отдельными компонентами ракеты и системами программ, использованных в ней, для тестирований и осмотра. Получив эту информацию, комиссия смогла реконструировать последовательность событий 4 июня 1996 года.

1. Программный модуль, в котором в итоге возникла ошибка, был «унаследован» от ракеты-носителя Ariane 4. Этот модуль производил выравнивание инерционной платформы для того, чтобы оценить точность измерений, проведенных инерционной системой ориентировки. В Ariane 5 после старта данный модуль был не нужен. Однако в Ariane 4 этот модуль работал еще в течение 50 с. Начальная часть траектории полета Ariane 5 существенно отличалась от траектории Ariane 4, и этот программный модуль никогда соответствующим образом не тестировался.

2. Вскоре после старта ошибочный программный модуль попытался посчитать значение, основанное на горизонтальной скорости ракеты. Поскольку для Ariane 5 это значение было существенно больше, чем то, которое ожидалось для Ariane 4, возникла ошибка и на активной, и на запасной инерционной системе ориентировки. **Допустимость такого преобразования не была проверена, поскольку ожидалось, что такого никогда не случится.**

3. Спецификация обработки ошибок в системе указывала, что контекст ошибок должен быть сохранен в постоянной памяти (ПЗУ) до отключения процессора. После ошибки операнда инерционная система ориентировки сохранила контекст ошибки, как было установлено. Эти данные были прочитаны бортовым компьютером. На их ос-

нове компьютер отдал команду соплам твердотопливного ускорителя и главному двигателю. Команда требовала полного отклонения сопел, что вызвало выход ракеты на запредельную траекторию [1].

4. На новой траектории ракета подверглась запредельной аэродинамической нагрузке и начала разрушаться. Стартовые двигатели отделились от ракеты, что запустило ее самоуничтожение.

Несомненно, глупая ошибка, но возникает вопрос: **как эта ошибка миновала стадию тестирования?** Аэрокосмическая индустрия отличается строгими стандартами и скрупулезными процессами и процедурами, направленными на проверку безопасности из-за высокой цены ошибок. Комиссия по расследованию задала тот же вопрос, и **команда обслуживания Ariane 5 представила следующие объяснения:**

- команда Ariane 5 решила не защищать некоторые переменные от возможной ошибки операнда, поскольку они считали, что значения этих переменных либо ограничены физическими факторами, либо имеют существенный запас по максимальной величине;
- команда Ariane 5 решила не включать данные о траектории в функциональные требования для инерционной системы ориентировки. Следовательно, данные о траектории Ariane 5 не использовались при тестировании;
- из-за физических законов трудно осуществить реалистичный полетный тест инерционной системы ориентировки. При функциональном имитационном тестировании полетных программ было решено не включать в тест эту систему главным образом по той причине, что она должна быть проверена при тестировании аппаратного уровня, а также потому, что было бы трудно достигнуть необходимой точности при имитационном тестировании, если бы была использована реальная инерционная система ориентировки [1].

Таким образом, программное обеспечение, содержащее серьезные ошибки и недоработки, которое не прошло тщательную проверку, зачастую представляет опасность, ведет к потере времени и финансовым затратам, является причиной травм или смертей, ставит под удар деловую репутацию компании-разработчика.

Тестирование программного обеспечения представляет собой один из возможных способов оценки его качества. Показателями качества могут служить количество найденных дефектов, их критичность, распределение дефектов по модулям и компонентам тестируемой системы и т. д. **Если найденные в процессе тестирования дефекты исправляются, то качество программного обеспечения повышается.** Однако исправление одного дефекта может порождать новые дефекты, еще более критичные, трудновоспроизводимые. В таком случае динамика изменения показателей качества в общем итоге может дать отрицательную картину.

Согласно стандартному глоссарию терминов, используемых в тестировании программного обеспечения, **тестирование –** это процесс, содержащий в себе все активности жизненного цикла, как динамические, так и статические, касающиеся планирования, подготовки и оценки программного продукта и связанных с этим результатов работ с целью определить, что они соответствуют описанным требованиям, показать, что они подходят для заявленных целей и для определения дефектов [2].

Тестирование включает в себя не только выполнение написанной программы в различных условиях, но и другие активности, которые имеют место **как до, так и после выполнения тестовых прогонов.**

Тестирование включает в себя:

- планирование,
- управление,
- подготовку тестовых данных и выбор условий,
- разработку и выполнение тестовых сценариев,
- проверку результатов,
- оценку критериев выхода,
- создание отчетов о процессе тестирования.

Тестирование преследует несколько целей. Первая и самая очевидная цель данного процесса – это **выявление дефектов.** Дефекты в процессе тестирования могут обнаруживаться не только в программе, но и в документации, которая также подвергается проверке, поэтому другая цель тестирования – **предотвращение дефектов.** Чем

раньше обнаруживается дефект, тем дешевле его исправление и тем меньше последствий он несет как для компании-разработчика, так и для непосредственного пользователя продукта.

Обнаруживая и исправляя дефекты мы тем самым **повышаем уровень качества программного продукта.** Кроме того, информация, содержащаяся в дефектах, является **базой для формирования представления об уровне качества программы**, а следовательно, на основании такой информации **можно принимать управленческие решения.**

Существуют **различные принципы тестирования**, но в качестве основных можно назвать следующие:

1) **тестирование показывает наличие дефектов, но не доказывает, что их нет.** Вероятность наличия дефектов снижается, но отсутствие дефектов не доказывает абсолютную корректность программного обеспечения;

2) **тестирование необходимо начинать как можно раньше в жизненном цикле разработки системы.** Тестирование не может быть бесцельным. У каждой активности необходимо определить четкую цель;

3) **полное или исчерпывающее тестирование невозможно,** так как существует огромное количество комбинаций, предусловий, постусловий и т. д. Для того чтобы наиболее четко определить, как и что необходимо тестировать, применяются анализ рисков, расстановка приоритетов, оценка критичности функционала;

4) **парадокс пестицида** – один из самых известных принципов, который означает, что многократный прогон одних и тех же тестов при соблюдении одинаковых условий неэффективен. Такие тесты перестают находить дефекты, но это не означает, что их нет. Тесты должны наиболее полно охватывать систему, а для этого они должны быть разноплановыми, разносторонними. Написание тестов – творческая задача. Каждый тест может найти большое количество дефектов, если написан грамотно и продуманно;

5) **дефекты скапливаются в небольшом количестве модулей,** на которых в конечном счете необходимо сосредоточить основные усилия по тестированию;

6) **тестирование напрямую зависит от объекта, который подвергается испытаниям.** Прежде чем начать тестирование, нужно изучить систему и понять наиболее критичные и уязвимые места, важность тех или иных функций для пользователей;

7) **заблуждение об отсутствии ошибок в системе.** Данный принцип состоит в том, что, если система не подходит пользователю и изначально была спроектирована неверно, то она не сможет удовлетворить потребности, даже если множество дефектов были обнаружены и устранены [3].

Контрольные вопросы

1. Какие исторические примеры серьезных программных ошибок, которые привели к финансовым потерям и/или человеческим жертвам, вам известны?
2. В каком случае труднее всего найти ошибки?
3. Что может служить показателем качества программного обеспечения?
4. Что может свидетельствовать о повышении качества программного обеспечения?
5. К каким последствиям может привести исправление дефектов?
6. Какие активности включает в себя тестирование?
7. Для чего может использоваться содержащаяся в дефектах информация?
8. Какие цели преследует тестирование?
9. Возможно ли исчерпывающее тестирование? Почему?
10. В чем состоит парадокс пестицида?
11. Как в большинстве случаев распределены дефекты внутри программной системы?
12. Для чего необходим анализ рисков?
13. От чего в большей степени зависит тестирование?
14. Что не может доказать тестирование?
15. В каком случае тестирование оказывается бессильным?

2. ВИДЫ ТЕСТИРОВАНИЯ

Как и любая классификация, классификация видов тестирования подразумевает выделение классификационных признаков. Наиболее общая классификация видов тестирования представлена в табл. 1.

Таблица 1

Виды тестирования

КЛАССИФИКАЦИОННЫЙ ПРИЗНАК	ВИД ТЕСТИРОВАНИЯ
По объекту тестирования	• Функциональное тестирование • нагрузочное тестирование • тестирование производительности • тестирование удобства использования (usability) • тестирование интерфейса пользователя • тестирование безопасности • тестирование совместимости
По знанию системы	• Черный ящик • белый ящик
По степени автоматизированности	• Ручное тестирование • автоматизированное тестирование
По уровню тестирования	• Компонентное тестирование • интеграционное тестирование
По времени проведения	• Альфа-тестирование • бета-тестирование • приемочное тестирование • регрессионное тестирование • повторное тестирование
По признаку позитивности сценариев	• Позитивное тестирование • негативное тестирование
По степени подготовленности к тестированию	• Тестирование по документации • тестирование по тест-кейсам • исследовательское тестирование

Функциональное тестирование (тестирование методом черного ящика, black-box testing). Функции, которые выполняет система, подсистема или компонент, могут быть описаны в таких артефактах процесса разработки, как спецификация требований, сценарии использования системы или функциональная спецификация, либо могут быть недокументированны. Эти функции описывают, «что» данная система делает. Функциональные тесты разрабатываются на основе функций и возможностей системы (описанных в документах или понятных тестировщикам) и их взаимодействия со специфичными системами и могут быть выполнены на всех уровнях тестирования (например, тесты для компонентов могут основываться на спецификациях компонентов).

Один из типов функционального тестирования, **тестирование безопасности,** исследует функции (например, брандмауэр), касающиеся обнаружения угроз, таких как вирусы, поступающих извне. Другой тип функционального тестирования, **тестирование возможности взаимодействия (тестирование совместимости),** оценивает способность программного продукта взаимодействовать с одним или более указанными компонентами или системами.

Нефункциональное тестирование включает (но не ограничивается) нагрузочное тестирование, тестирование производительности, стресс-тестирование, тестирование удобства использования, тестирование восстановления, тестирование надежности и тестирование переносимости. Это тестирование того, «как» система работает. Нефункциональное тестирование может выполняться на всех уровнях тестирования. Термин «нефункциональное тестирование» описывает тесты, необходимые для оценки характеристик систем и программ, которые могут быть количественно измерены (например, время отклика при **тестировании производительности**).

Тестирование удобства использования (usability testing) – согласно ISO 9126, это метод тестирования, направленный на установление степени удобства использования, обучаемости, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий [4].

Структурное тестирование (тестирование методом белого ящика, white-box testing) может выполняться на всех уровнях тестирования. Структурные методы тестирования лучше всего использо-

вать после методов разработки тестов на основе спецификации, чтобы измерить тщательность тестирования, используя измерения покрытия структуры программы.

Тестирование методом белого ящика еще называют «тестированием по маршрутам». Под **маршрутом** понимают последовательности операторов программы, которые выполняются при конкретном варианте исходных данных. Программный код системы в случае структурного тестирования открыт и доступен для анализа и проведения испытаний [5].

Модульное (компонентное тестирование) основано на наличии требований к компонентам системы, дизайне и его деталях, а также на наличии доступа к программному коду. В рамках компонентного тестирования испытываются программы и компоненты, модули баз данных. Дефекты отыскиваются в программных модулях, которые можно подвергать тестированию изолированно, без интеграционных взаимодействий с другими компонентами системы. Часто модульное тестирование заставляет прибегать к использованию заглушек, эмуляторов и других вспомогательных инструментов, которые так или иначе способны заменить полностью или частично некоторые реальные компоненты целостной системы.

Компонентное тестирование наиболее часто производится с доступом к программному коду, поддержкой рабочего окружения и инструментов отладки. Для реальных проектов компонентное тестирование – это уровень тестирования, который более относится к стадии разработки. Разработчик, отлаживая код, находит в нем дефекты, которые исправляются сразу без необходимости занесения их в систему отслеживания дефектов. Важно отметить, что отладка (локализация и исправление дефекта) относится к процессу разработки, а не тестирования. Таким образом, разработчик – непосредственный участник процесса тестирования, и от того, насколько качественно будет проведено модульное тестирование, во многом зависят дальнейшие стадии тестирования системы [3].

Интеграционное тестирование проверяет интерфейсы между компонентами, взаимодействие различных частей системы, таких как операционная система, файловая система, аппаратное обеспечение, а также интерфейсы между системами.

Разработчики рыночного, или коробочного, ПО часто хотят получить отзывы от потенциальных или существующих заказчиков до того, как начнется продажа продукта. **Альфа-тестирование** выполняется организацией, разрабатывающей продукт, но не группой разработчиков. **Бета-тестирование**, или тестирование в условиях эксплуатации, выполняется покупателями или потенциальными заказчиками с использованием их собственного оборудования и ресурсов [3].

Приемочным тестированием системы чаще всего занимаются заказчики или пользователи системы, а также другие заинтересованные лица. Основная цель приемочного тестирования – проверка работоспособности системы, частей системы или отдельных нефункциональных характеристик системы. Поиск дефектов не является главной целью приемочного тестирования [2].

Регрессионное тестирование – это повторное тестирование уже протестированных программ после внесения в них изменений, чтобы обнаружить дефекты, внесенные или пропущенные в результате этих действий. Эти дефекты могут быть как в проверяемом компоненте, так и в связанном или несвязанном с ним. Регрессионное тестирование выполняется, когда в программное обеспечение или его окружение вносятся изменения. *Глубина регрессионного тестирования* оценивается риском пропуска дефектов в программном обеспечении, которое работало ранее [2].

Регрессионное тестирование может выполняться на всех уровнях тестирования; включает функциональное, нефункциональное и структурное тестирование. **Регрессионные наборы тестов запускаются множество раз и меняются медленно, поэтому регрессионное тестирование чаще все автоматизируют.**

После того как дефект обнаружен и исправлен, программу необходимо перепроверить, чтобы убедиться, что исходный дефект успешно устранен. Это называется *подтверждением (верификацией дефекта).* В процессе верификации дефектов тестировщик часто прибегает к повторному прохождению ранее пройденных тестовых сценариев, т. е. проводит *повторное тестирование*.

Системное тестирование сконцентрировано на поведении тестового объекта как целостной системы или продукта. Системное тестирование способно выявить несовместимость с окружением, некорректные комбинации данных, заведомо непредусмотренные сценарии

использования, неудобство для пользователя, отсутствие необходимого пользователям функционала и т. д. Во время системного тестирования тестовое окружение должно быть как можно ближе к предполагаемому эксплуатационному окружению системы для минимизации риска пропуска отказов, связанных с эксплуатационным окружением системы.

Системное тестирование может осуществляться на базе сценариев использования и на базе требований к системе.

Зная, как будет использоваться система, и имея конкретный набор вариантов использования (*use cases*), тестировщик разрабатывает набор тестовых сценариев для прохождения (*test cases*).

Если набор вариантов использования отсутствует, но имеются требования к системе и ее поведению, то на основании этих требований тестировщик также может написать тестовые сценарии, проверяющие, выполняется ли конкретное требование или нет [3].

Зачастую при написании и прохождении тестовых сценариев тестировщик прибегает не только к изучению предоставленных требований, но и *к собственному опыту и здравому смыслу.* Такой подход – одна из главных составляющих процесса тестирования. Данный метод носит название **«тестирование, основанное на опыте».** Но важно помнить, что в этом случае качество продукта будет во многом зависеть от опыта тестировщика.

На основе ранее полученного опыта тестировщик может предположить, как поведет себя система в той или иной ситуации и где стоит ожидать наибольшего количества дефектов. На основании таких знаний разрабатываются тестовые сценарии, способные выявить наиболее ожидаемые дефекты. Такой подход называют **«атакой на недочеты».**

Бывают случаи, когда система слабо документирована или документация к продукту потеряла свою актуальность. В такой ситуации часто прибегают к **исследовательскому тестированию,** которое представляет собой параллельную разработку тестов, их выполнение и изучение системы по разрабатываемым тестовым сценариям и на основе ее использования в процессе тестирования. Исследовательское тестирование возможно в условиях нехватки времени или может применяться для проверки процесса тестирования в целом [2].

После установки система программного обеспечения обычно находится в эксплуатации в течение многих лет. В это время сама система, ее конфигурация или среда исполнения часто изменяются или расширяются. Раннее планирование релизов крайне важно для успешного тестирования в период сопровождения. При этом необходимо отличать запланированные выпуски и срочные исправления. **Тестирование в период сопровождения** выполняется на текущей ОС и может быть вызвано модификацией, переносом или прекращением эксплуатации данной системы.

Контрольные вопросы

1. Назовите основные признаки классификации видов тестирования.
2. В чем разница между функциональным и структурным тестированием?
3. В чем суть компонентного тестирования и каковы его основные особенности?
4. Для чего главным образом необходимо интеграционное тестирование?
5. Какой вид тестирования наиболее часто автоматизируется и почему?
6. В чем разница между альфа- и бета-тестированием?
7. Что призвано оценить тестирование совместимости?
8. Что такое usability testing?
9. В чем состоят особенности тестирования «на опыте»?
10. Что подразумевает под собой «атака на недочёты»?
11. Какие активности включает в себя исследовательское тестирование?
12. При каком условии исследовательское тестирование может быть эффективным?
13. В чем особенности тестирования в период сопровождения?
14. Что такое системное тестирование?
15. В чем разница между системным тестированием на базе сценариев использования и на базе требований?

3. СТАНДАРТЫ, РЕГЛАМЕНТИРУЮЩИЕ ПРОЦЕСС ТЕСТИРОВАНИЯ

Данный раздел посвящен стандартам, которые регламентируют процесс тестирования ПО и связанные с ним процессы. Каждый стандарт представляет собой набор правил и требований, предназначенных для обеспечения правильности действий всех организаций, которые выполняют описанные в стандартах процессы.

В любой модели жизненного цикла ПО (последовательной, итеративно-инкрементной) представлены **характеристики качественного тестирования.** Перечислим основные из них:

- наличие целей тестирования на каждом уровне тестирования;
- любому процессу разработки соответствует свой процесс тестирования;
- возможность рецензирования документации тестировщиками начиная с самых первых версий;
- анализ требований и написание тестов начинается одновременно с деятельностью разработчиков или раньше.

Тестирование и все входящие в него процессы регламентируются многочисленными стандартами. Можно выделить следующие стандарты, относящиеся к тестированию и качеству ПО:

- **IEEE 12207/ISO/IEC 12207-2008 Software Life Cycle Processes** – описывает жизненный цикл программного обеспечения и место различных процессов в нём;
- **ISO/IEC 9126-1:2001 Software Engineering – Software Product Quality** – описывает характеристики качества программных продуктов;
- **IEEE 829-1998 Standard for Software Test Documentation** – описывает виды документов, служащих для подготовки тестов;
- **IEEE 1008-1987 (R1993, R2002) Standard for Software Unit Testing** – описывает организацию модульного тестирования;
- **ISO/IEC 12119:1994 Information Technology. Software Packages – Quality Requirements and Testing** (аналог AS/NZS 4366:1996 и ГОСТ Р-2000, также принят IEEE под номером IEEE 1465-1998) – описывает требования к процедурам тестирования программных систем;
- **ISO/IEC 20000:2005. Процессы, сертификация ISO 20000** – описывает требования к управлению и обслуживанию IT-сервисов.

3.1. IEEE 12207/ISO/IEC 12207-2008 Software Life Cycle Processes

Данный стандарт описывает общую структуру процессов жизненного цикла программных средств. В рамках стандарта определены процессы, активности и задачи, которые актуальны при разработке, тестировании, поставке, приобретении, применении, сопровождении и завершении использования программного продукта.

Под **жизненным циклом программного обеспечения (ЖЦ ПО)** понимается цикл создания и развития ПО, который начинается с момента принятия решения о необходимости создания программного продукта и завершается в момент изъятия программного продукта из эксплуатации [6].

Все процессы жизненного цикла группируются. Группировка процессов представлена в табл. 2.

Таблица 2

Группировка процессов жизненного цикла ПО согласно IEEE 12207

Фактор	Атрибуты
Процессы соглашения	• Поставка; • приобретение
Процессы организационного обеспечения проекта	• Процесс менеджмента модели жизненного цикла; • процесс менеджмента инфраструктуры; • процесс менеджмента портфеля проектов; • процесс менеджмента людских ресурсов; • процесс менеджмента качества
Процессы проекта	• Процессы менеджмента проекта: ✓ процесс планирования проекта; ✓ процесс управления и оценки проекта; • процессы поддержки проекта: ✓ процесс менеджмента решений; ✓ процесс менеджмента рисков; ✓ процесс менеджмента конфигурации; ✓ процесс менеджмента информации; ✓ процесс измерений

Продолжение табл. 2

Фактор	Атрибуты
Технические процессы	• Определение требований правообладателей; • анализ системных требований; • проектирование архитектуры системы; • процесс реализации; • процесс комплексирования системы; • процесс квалификационного тестирования системы; • процесс инсталляции программных средств; • процесс поддержки приемки программных средств; • процесс функционирования программных средств; • процесс сопровождения программных средств; • процесс изъятия из обращения программных средств
Процессы реализации программных средств	• Процесс анализа требований к программным средствам; • процесс проектирования архитектуры программных средств; • процесс детального проектирования программных средств; • процесс конструирования программных средств; • процесс комплексирования программных средств; • процесс квалификационного тестирования программных средств

Окончание табл. 2

Фактор	Атрибуты
Процессы поддержки программных средств	• Процесс менеджмента документации программных средств; • процесс менеджмента конфигурации программных средств; • процесс обеспечения гарантии качества программных средств; • процесс верификации программных средств; • процесс валидации программных средств; • процесс ревизии программных средств; • процесс аудита программных средств; • процесс решения проблем в программных средствах
Процессы повторного применения программных средств	• Процесс проектирования доменов; • процесс менеджмента повторного применения активов; • процесс менеджмента повторного применения программ

3.2. ISO/IEC 9126-1:2001 Software Engineering – Software Product Quality

Стандарт ISO 9126 нацеливает на то, чтобы учитывать три разные точки зрения при рассмотрении качества ПО:

- **точку зрения разработчиков,** которые воспринимают внутреннее качество ПО;
- **точку зрения руководства и аттестации ПО** на соответствие сформулированным к нему требованиям, в ходе которой определяется внешнее качество ПО;
- **точку зрения пользователей,** ощущающих качество ПО при использовании.

Для внешнего и внутреннего качества в рамках ISO 9126 предложена **модель качества, состоящая из 6 факторов и 27 атрибутов**. Данная модель представлена на рис. 1.

Рис. 1. Факторы и атрибуты внешнего и внутреннего качества ПО согласно ISO 9126

Описание факторов и атрибутов, используемых для оценки качества ПО с точки зрения разработчиков и руководства, представлено в табл. 3.

Таблица 3

Факторы и атрибуты внешнего и внутреннего качества ПО согласно ISO 9126

Фактор	Атрибуты
Функциональность (functionality) – способность ПО в определенных условиях решать задачи, нужные пользователям. Определяет, что именно делает ПО	1. *Функциональная пригодность (suitability)* – способность решать нужный набор задач 2. *Точность (accuracy)* – способность выдавать нужные результаты 3. *Способность к взаимодействию, совместимость (interoperability)* – способность взаимодействовать с нужным набором других систем 4. *Соответствие стандартам и правилам (compliance)* – соответствие ПО имеющимся стандартам, нормативным и законодательным актам, другим регулирующим нормам 5. *Защищенность (security)* – способность предотвращать неавторизованный и неразрешенный доступ к данным, коммуникациям и другим элементам ПО

Продолжение табл. 3

Фактор	Атрибуты
Надежность (reliability) – способность ПО поддерживать определенную работоспособность в заданных условиях	1. *Зрелость, завершенность (maturity)* – величина, обратная частоте отказов ПО. Определяется средним временем работы без сбоев и величиной, обратной вероятности возникновения отказа за данный период времени 2. *Устойчивость к отказам (fault tolerance)* – способность поддерживать заданный уровень работоспособности при отказах и нарушениях правил взаимодействия с окружением 3. *Способность к восстановлению (recoverability)* – способность восстанавливать определенный уровень работоспособности и целостность данных после отказа в рамках заданных времени и ресурсов 4. *Соответствие стандартам надежности (reliability compliance)*
Удобство использования (usability), или практичность – способность ПО быть удобным в обучении и использовании, а также привлекательным для пользователей	1. *Понятность (understandability)* – показатель, обратный усилиям, которые затрачиваются пользователями на восприятие основных понятий ПО и осознание способов их использования для решения своих задач 2. *Удобство обучения (learnability)* – показатель, обратный усилиям, затрачиваемым пользователями на обучение работе с ПО 3. *Удобство работы (operability)* – показатель, обратный трудоемкости решения пользователями задач с помощью ПО 4. *Привлекательность (attractiveness)* – способность ПО быть привлекательным для пользователей 5. *Соответствие стандартам удобства использования (usability compliance)*

Продолжение табл. 3

Фактор	Атрибуты
Производительность (efficiency), или эффективность, – способность ПО при заданных условиях обеспечивать необходимую работоспособность по отношению к выделяемым для этого ресурсам	1. *Временная эффективность (time behaviour)* – способность ПО решать определенные задачи за отведенное время 2. *Эффективность использования ресурсов (resource utilisation)* – способность решать нужные задачи с использованием заданных объемов ресурсов определенных видов. Имеются в виду такие ресурсы, как оперативная и долговременная память, сетевые соединения, устройства ввода и вывода и пр. 3. *Соответствие стандартам производительности (efficiency compliance)*
Удобство сопровождения (maintainability) – удобство проведения всех видов деятельности, связанных с сопровождением программ	1. *Анализируемость (analyzability), или удобство проведения анализа* – удобство проведения анализа ошибок, дефектов и недостатков, а также удобство анализа необходимости изменений и их возможных последствий 2. *Удобство внесения изменений (changeability)* – показатель, обратный трудозатратам на выполнение необходимых изменений 3. *Стабильность (stability)* – показатель, обратный риску возникновения неожиданных эффектов при внесении необходимых изменений 4. *Удобство проверки (testability)* – показатель, обратный трудозатратам на проведение тестирования и других видов проверки того, что внесенные изменения привели к нужным результатам *5. Соответствие стандартам удобства сопровождения (maintainability compliance)*
Переносимость (portability) – способность ПО сохранять работоспособность при переносе из одного окружения в другое, включая организационные, аппаратные и программные аспекты окружения	1. *Адаптируемость (adaptability)* – способность ПО приспосабливаться к различным окружениям без проведения для этого действий, помимо заранее предусмотренных 2. *Удобство установки (installability)* – способность ПО быть установленным или развернутым в определенном окружении 3. *Способность к сосуществованию (coexistence)* – способность ПО сосуществовать в общем окружении с другими программами, деля с ними ресурсы

Окончание табл. 3

Фактор	Атрибуты
Переносимость (portability) – способность ПО сохранять работоспособность при переносе из одного окружения в другое, включая организационные, аппаратные и программные аспекты окружения	4. *Удобство замены (replaceability) другого ПО данным* – возможность применения данного ПО вместо других программных систем для решения тех же задач в определенном окружении. 5. *Соответствие стандартам переносимости (portability compliance)* [7]

Для оценки качества ПО при использовании, т. е. с точки зрения целевой аудитории, данный стандарт предлагает систему факторов, представленных в табл. 4.

Таблица 4

Факторы оценки качества ПО с точки зрения пользователей согласно ISO 9126

Фактор	Описание
Эффективность (effectiveness)	Способность решать задачи пользователей с необходимой точностью при использовании в заданном контексте
Продуктивность (productivity)	Способность предоставлять определенные результаты в рамках ожидаемых затрат ресурсов
Безопасность (safety)	Способность обеспечивать необходимо низкий уровень риска нанесения ущерба жизни и здоровью людей, бизнесу, собственности или окружающей среде
Удовлетворение пользователей (satisfaction)	Способность приносить удовлетворение пользователям при использовании в заданном контексте

3.3. IEEE 829-1998 Standard for Software Test Documentation

Данный стандарт описывает требования к вспомогательным артефактам тестирования. Основные артефакты и их описание представлены в табл. 5.

Таблица 5

Основные артефакты тестирования согласно IEEE 829

Артефакт	Описание
Тестовый план (test plan)	Основной документ, связывающий разработку тестов и тестирование с задачами проекта. Определяет необходимые виды тестирования, техники, проверяемую функциональность, критерии оценки полноты тестов и критерии выхода, а также график тестирования
Тестовые сценарии (test case)	Сценарии проведения отдельных тестов. Каждый тестовый сценарий предназначен для проверки определенных свойств некоторых компонентов системы в определенной конфигурации
Описания тестовых процедур (test procedure specifications)	Тестовые процедуры могут быть представлены в виде скриптов или программ, автоматизирующих запуск тестовых сценариев (автоматизированное тестирование), или в виде инструкций для человека, следуя которым можно выполнить те же сценарии (ручное тестирование)
Отчеты о нарушениях (test incident reports, или bug reports)	Описание ошибок, обнаруженных при выполнении тестов, с указанием всех условий, необходимых для их проявления
Итоговый отчет о тестировании (test summary report)	Отчет, аккумулирующий общую информацию по результатам тестов, включающий достигнутое тестовое покрытие и общую оценку качества компонентов тестируемой системы [7]

3.4. IEEE 1008-1987 (R1993, R2002) Standard for Software Unit Testing

Стандарт IEEE 1008 подробно описывает процедуру подготовки модульных тестов, процессы выполнения и оценки результатов. Основные процессы, которые регламентирует стандарт:

- планирование – определение используемых методов тестирования необходимых ресурсов, критериев полноты тестов и критерия выхода;
- определение проверяемых требований и ограничений;
- уточнение, корректировка и детализация планов;
- разработка набора тестов;
- выполнение тестов;
- проверка достижения критерия выхода с последующей оценкой полноты тестирования по специально выбранным критериям;
- оценка затрат ресурсов и качества протестированных модулей [7].

3.5. ISO/IEC 12119:1994 Information Technology. Software Packages – Quality Requirements and Testing

Данный стандарт определяет требования к описанию программного продукта и пользовательской документации. Кроме того, стандарт содержит требования, предъявляемые к программам и данным, а также инструкции по испытаниям программных продуктов на соответствие заданным требованиям.

Описание продукта согласно стандарту должно быть таким, чтобы потенциальные покупатели могли без труда оценить пригодность для них данного продукта ещё до его приобретения. Описание продукта должно включать в себя следующие разделы:

1) общие требования к содержанию;
2) обозначения и указания;
3) функциональные возможности;
4) надёжность;
5) практичность;
6) эффективность;
7) сопровождаемость и мобильность [8].

3.6. ISO/IEC 20 000:2005. Процессы, сертификация ISO 20 000

Данный международный стандарт определяет требования к управлению и обслуживанию IT-сервисов. Цель – обеспечение качества сервиса (системы) на приемлемом уровне. Данный документ можно условно разделить на две основополагающие части:

1) **ISO 20 000-1:2005 «Information technology – Service management. Part 1: Specification»** – это подробное описание требований к системе менеджмента ИТ-сервисов и ответственности за инициирование, выполнение и поддержку в организациях. Процессы в данном разделе объединены в пять групп:

а) **процессы предоставления сервисов** (**Service delivery processes**) – группа процессов управления уровнем сервисов, предоставлением отчётности по предоставлению сервисов, составлением бюджета и статистики затрат;

б) **процессы управления взаимодействием** (**Relationship processes**) – описываются вопросы коммуникации между компанией, заказчиком и различными подрядчиками;

в) **процессы разрешения** (**Resolution processes**) – управление проблемами и инцидентами;

г) **процессы контроля** (**Control processes**) – описываются особенности контроля и управления изменениями в компании;

д) **процессы управления релизами** (**Release processes**) – описываются активности, связанные с внедрением новых и уже имеющихся решений.

2) **ISO 20000-2:2005 «Information technology – Service management. Part 2: Code of Practice»** – содержит практические рекомендации по процессам, требования к которым сформулированы в первой части [9, 10].

Кроме того, данный стандарт предлагает и подробно описывает модель улучшения процессов и управления качеством. Эту модель называют моделью, или **циклом, PDCA**, а также **циклом Деминга** (рис. 2).

Согласно циклу **для обеспечения определенного уровня качества необходимо постоянно повторять следующие процессы:**

- **планирование (PLAN) –** подразумевает ответы на вопросы «что?», «когда?» нужно сделать, а также «кто?» и «с помощью чего?» должен это сделать, т. е. на данном этапе проектируются или корректируются процессы;

• **выполнение (DO)** – включает выполнение запланированных на первом этапе работ;

• **проверка (CHECK)** – собираются метрики, готовится отчётность, согласно которой выявляется, какой результат дало выполнение работ;

• **действие (ACT)** – планы корректируются согласно результатам предыдущего этапа, проводятся требующиеся изменения.

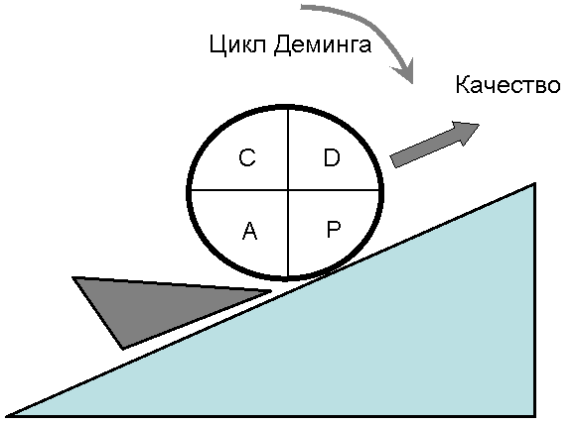

Рис. 2. Цикл Деминга

В стандарте также определяются требования к мерам ответственности руководителей компании, которая занимается разработкой, т. е. предоставляет IT-сервисы, к компетенции персонала и управлению документацией [11].

Контрольные вопросы

1. Какие характеристики качественного тестирования вам известны?
2. Какими стандартами регламентируется процесс тестирования?
3. Что такое жизненный цикл программного обеспечения?
4. Какие группы процессов входят в жизненный цикл программного обеспечения согласно стандарту IEEE 12207?
5. С каких трёх точек зрения согласно ISO 9126 может быть оценено качество программного продукта?
6. Что такое функциональность ПО?
7. Что такое надежность ПО?
8. Что подразумевается под зрелостью ПО?
9. Что такое практичность ПО?
10. Что описывает стандарт IEEE 829?
11. Какие основные процессы составляют Unit-тестирование согласно IEEE 1008?
12. Что описывает стандарт ISO 20 000?
13. Какие два раздела входят в стандарт ISO 20 000 и что включает каждый из них?
14. Какие пять групп процессов описываются в стандарте ISO 20 000?
15. Что такое цикл Деминга и из каких этапов он состоит?

4. МЕТОДИКИ РАЗРАБОТКИ ТЕСТОВ

Данный раздел посвящен способам создания тестовых сценариев. Существует много методик, но здесь подробно рассмотрены эквивалентное разбиение и анализ граничных значений, таблицы альтернатив, диаграммы причинно-следственных связей, диаграммы переходов состояний и таблицы переходов состояний.

4.1. Эквивалентное разбиение

Входные данные для программного обеспечения или системы разбиваются на группы, от которых ожидается сходное поведение, т. е. они должны обрабатываться аналогичным образом. **Эквивалентные области (или классы) могут быть определены как для валидных, так и для невалидных данных, т. е. тех значений, которые должны отвергаться.** Области также могут быть определены для выходных данных, внутренних значений, значений, зависящих от времени (например, до или после некоторого события), и для параметров интерфейса (например, во время интеграционного тестирования). Тесты могут разрабатываться для покрытия всех валидных и всех невалидных классов. **Эквивалентное разбиение применимо на всех уровнях тестирования**, может быть использовано с целью покрытия входных и выходных данных. Оно может применяться при ручном вводе данных, при передаче данных через интерфейсы в систему или при проверке параметров интерфейсов в интеграционном тестировании.

Техника заключается в разбиении всего набора тестов на **классы эквивалентности** с последующим сокращением числа тестов. Целью данной техники является не только сокращение числа тестов, но и сохранение приемлемого тестового покрытия [12].

При использовании этой техники тестировщик должен помнить:

- слишком большое количество эквивалентных классов увеличивает вероятность, что множество тестов будет лишним (избыточным);
- слишком малое число эквивалентных классов увеличивает вероятность, что ошибки продукта будут пропущены.

Примерный алгоритм использования техники следующий:

1. **Определить классы эквивалентности**. Это главный шаг техники. От него во многом зависит эффективность её применения.

2. **Выбрать одного представителя от каждого класса**. На этом шаге из каждого эквивалентного набора тестов выбирают один тест.

3. **Выполнить тесты**. На этом шаге выполняют тесты от каждого класса эквивалентности.

Если есть время, можно протестировать еще несколько представителей от каждого класса. Но нужно иметь в виду, что если правильно выполнен первый шаг, т. е. правильно определены классы эквивалентности, то дополнительные тесты, скорее всего, будут **избыточными** и дадут тот же результат.

На первом шаге, когда определяют классы эквивалентности, могут помочь программисты. Они могут подсказать, как программа ведет себя при тех или иных входных параметрах.

Можно разбивать тесты на классы эквивалентности по разным принципам. От этого эффективность тестирования может выиграть. Например, если тестируют поле ввода, которое принимает максимум пять символов, то можно выбрать разные принципы разбиения на классы эквивалентности:

- по количеству символов;
- по типу символов (цифры, буквы, спецсимволы).

Пример. Рассмотрим функцию подсчета комиссии при отмене бронирования авиабилетов (рис. 3).

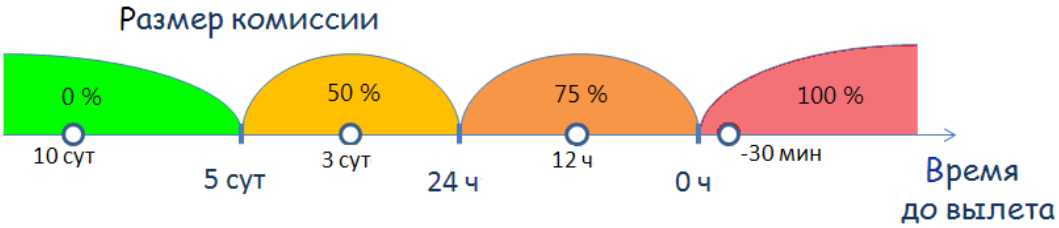

Рис. 3. Эквивалентное разбиение

Предположим, что размер комиссии зависит от времени до вылета, когда совершена отмена:

- за 5 сут до вылета комиссия составляет 0 %;
- меньше 5 сут, но больше 24 ч – 50 %;
- меньше 24 ч, но до вылета – 75 %;
- после вылета – 100 %.

Определим классы эквивалентности (для каждого теста из этих классов мы ожидаем получить одинаковый результат):

- 1-й класс: время до вылета > 5 сут;
- 2-й класс: 24 ч < время до вылета < 5 сут;
- 3-й класс: 0 ч < время до вылета < 24 ч;
- 4-й класс: время до вылета < 0 ч (вылет уже состоялся).

Выберем представителя от каждого класса. Здесь можно выбрать любые значения из класса. Если предположить, что разделение на классы эквивалентности было правильным, то нет разницы, какое значение из диапазона будет выбрано:

- время до вылета = 10 сут (тест из 1-го класса);
- время до вылета = 3 сут (тест из 2-го класса);
- время до вылета = 12 ч (тест из 3-го класса);
- время после вылета = 30 мин (тест из 4-го класса).

Выполним тесты:

- отменим бронь за 10 сут до вылета и проверим, что комиссия составила 0 %;
- отменим бронь за 3 сут до вылета и проверим, что комиссия составила 50 %;
- отменим бронь за 12 ч до вылета и проверим, что комиссия составила 75 %;
- отменим бронь через 30 мин после вылета и проверим, что комиссия составила 100 %.

Таким образом осталось всего четыре теста. А сколько возможных тестов существует? Даже если ввести ограничение, что отмена бронирования может произойти в рамках 10 суток до вылета и 1 суток после вылета, то будет около **950 400 возможных тестов** (если посчитать количество секунд в 11 сутках). Был рассмотрен очень простой пример. **Редко бывает так, что функция зависит только от одной переменной. Обычно классов эквивалентности больше и выделить их сложнее** [12].

Как и любая другая техника, анализ классов эквивалентности имеет достоинства и недостатки. К плюсам можно отнести заметное сокращение времени и улучшение структурированности тестирования, к минусам – то, что при неправильном использовании техники есть риск потерять баги.

4.2. Анализ граничных значений

Поведение на границах эквивалентных областей имеет наибольшие шансы быть некорректным, таким образом, **границы выступают потенциальным источником дефектов.** Минимальные и максимальные значения сегмента являются **граничными значениями.** Граничное значение для валидного сегмента называется **валидным граничным значением,** для невалидного сегмента – **невалидным.** Тесты могут разрабатываться для покрытия как валидных, так и невалидных граничных значений. При разработке тестовых сценариев выбирают тесты для каждого граничного значения. **Анализ граничных значений может применяться на всех уровнях тестирования.** Он относительно легок в применении и эффективен при поиске дефектов. Для выделения границ крайне полезны подробные спецификации. **Данный метод часто рассматривается как дополнение к методу эквивалентного разбиения.** Он может использоваться как для классов эквивалентности данных, вводимых на экране, так и, например, для классов эквивалентности временных диапазонов (например, таймауты или требования по быстродействию транзакций) или для размерности таблиц (например, размер таблицы 256 × 256).

Это техника проверки ошибок на границах классов эквивалентности. Если техника анализа классов эквивалентности ориентирована **на тестовое покрытие,** то эта техника **основана на рисках.** Ее основная идея в том, что **программа может сработать некорректно в области граничных значений.**

Считается, что с граничными значениями связаны серьезные риски. Давно замечено, что при разработке большое число проблем возникает на границах входных переменных. Даже если эквивалентные классы найдены правильно, то граничные значения могут быть ошибочно отнесены к другому классу.

Указанная техника на первый взгляд проста. Но её эффективное применение зависит от способности правильно выделить классы эквивалентности и затем выбрать тесты для проверки границ этих классов. **Цель техники сформулировать несложно: найти ошибки, связанные с граничными значениями** [12].

Примерный алгоритм использования техники анализа граничных значений состоит из четырёх основных шагов:

1. Необходимо **выделить классы эквивалентности**. От правильности разбиения на классы эквивалентности зависит эффективность тестов граничных значений.

2. Далее нужно **определить граничные значения** этих классов.

3. Затем следует **понять, к какому классу будет относиться каждая граница**.

4. Наконец, для каждой границы нужно **провести тесты по проверке значения до границы, на границе и сразу после границы**.

Можно сказать, что количество тестов для проверки граничных значений будет равно количеству границ, умноженному на 3. Причем в литературе по тестированию рекомендуется проверять значения вплотную к границе. Скажем, если есть диапазон целых значений, а граница находится в числе 10, то необходимо проводить тесты с числом 9 (вплотную до границы), 10 (на границе) и 11 (сразу после границы).

Вернемся к **примеру с бронированием** и проведем тестирование граничных значений (рис. 4).

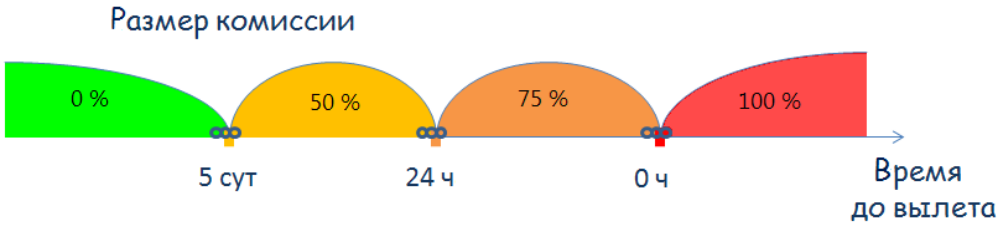

Рис. 4. Анализ граничных значений

Выделим классы эквивалентности:

- время до вылета > 5 сут;
- 24 ч ≤ время до вылета ≤ 5 сут;
- 0 ч < время до вылета < 24 ч;
- время до вылета ≤ 0 ч (вылет уже состоялся).

Определим границы:

- 5 сут;
- 24 ч;
- 0 ч.

Определим, к какому классу относятся границы: на этом шаге имеется спорный момент, на который нужно обратить внимание. При составлении задачи не было продумано, какая логика должна быть на самих границах. Это типичный пример того, как составители требо-

ваний не задумываются о границах. И поэтому программист может трактовать их по-своему:

- 5 сут – ко 2-му классу;
- 24 ч – ко 2-му классу;
- 0 ч – к 4-му классу.

Протестируем значения на границах, до и после них:

- отменим бронь за 5 сут + 1 с до вылета (или просто постараемся выполнить бронь как можно ближе к границе, но слева от нее) и проверим, что комиссия равна 0 %;
- отменим бронь ровно за 5 сут до вылета и проверим, что комиссия равна 50 %;
- отменим бронь за 5 сут – 1 с до вылета и проверим, что комиссия равна 50 %;
- отменим бронь за 24 ч + 1 с до вылета и проверим, что комиссия равна 50 %;
- отменим бронь ровно за 24 ч до вылета и проверим, что комиссия равна 50 %;
- отменим бронь за 24 ч – 1 с до вылета и проверим, что комиссия равна 75 %;
- отменим бронь за 1 с до вылета и проверим, что комиссия равна 75 %;
- отменим бронь ровно во время вылета и проверим, что комиссия равна 100 %;
- отменим бронь спустя 1 с после вылета и проверим, что комиссия равна 100 %.

Таким образом, имеется 9 тестов, по 3 теста на каждую границу. Если суммировать тесты, необходимые для проверки классов эквивалентности и граничных значений, получим: 4 + 9 = 13 тестов.

В некоторых источниках рекомендуется использовать классы эквивалентности и граничные условия вместе по следующим соображениям:

- техника анализа классов эквивалентности говорит о том, что мы должны выбрать минимум по одному значению из каждого класса;
- так как граница обычно относится к какому-то классу, то можно использовать ее как представителя этого класса.

Преимущество техники анализа граничных значений в том, что она **добавляет в технику анализа классов эквивалентности ориентированность на конкретный тип ошибок.** Техника анализа классов эквивалентности просто говорит нам о том, что нужно разбить все тесты на классы и провести тестирование всех классов. А техника граничных значений ориентирована на обнаружение конкретной проблемы – возникновения ошибок на границах классов эквивалентности.

Но, как и для техники анализа классов эквивалентности, эффективность техники анализа граничных значений зависит от правильности ее использования. Необходимо приложить усилия, чтобы правильно определить классы эквивалентности и их границы. Если отнестись к этому поверхностно, то есть риск пропустить ошибки [12].

4.3. Таблицы альтернатив и комбинированные техники

Эквивалентное разбиение и анализ граничных значений являются очень полезными методиками. Особенно они полезны, когда нужно протестировать валидацию полей входных значений пользовательского интерфейса. Однако большая часть тестирования включает тестирование бизнес-логики, которая расположена глубже пользовательского интерфейса.

Одной из мощных методик считается применение таблиц альтернатив. Суть метода состоит в том, что таблицы альтернатив отражают правила, управляющие обработкой транзакционных ситуаций. Благодаря своей простой и лаконичной структуре таблицы альтернатив облегчают проектирование тестов для этих правил – обычно один тест на одно правило.

Под «транзакционными ситуациями» имеют в виду те ситуации, для которых условия (входные данные, предусловия и т. п.), существующие в определенный момент времени для конкретной транзакции, сами по себе достаточны для определения действия, которое должна произвести система.

Для создания тестовых сценариев из таблицы альтернатив проектируются входные тестовые данные, которые удовлетворяют заданным условиям. Выходные тестовые данные соответствуют действиям при заданной комбинации условий. Во время выполнения теста проверяется, соответствуют ли фактические действия ожидаемым.

Создается достаточное количество тестовых сценариев – такое, чтобы каждая комбинация условий была покрыта как минимум одним тестовым сценарием [12].

При использовании таблицы альтернатив критерий покрытия сводится к простому для запоминания правилу – не менее одного теста для каждого столбца в таблице.

Итак, какие же типы дефектов находят с помощью таблиц альтернатив? Их два. Первый, **когда при определенной комбинации условий может произойти неверное событие.** Иными словами, когда существует некое действие, которое система не должна выполнять при такой комбинации условий, но выполняет. Второй тип дефектов – **когда при определенной комбинации условий система может не выполнить корректное действие.** Иными словами, когда существует некое действие, которое система должна выполнить при такой комбинации условий, но не выполняет.

Представим себе приложение для электронной коммерции. На уровне интерфейса пользователя нужно проверить платежную информацию, а именно: тип кредитной карты, номер карты, секретный код карты, месяц и год истечения срока действия карты, а также имя держателя карты. Существует целый набор условий, определяющий процесс обработки:

- Принадлежит ли введенная кредитная карта указанному лицу, и верна ли остальная информация?
- Действует ли карта или истек срок действия?
- Находится ли лицо в пределах лимита по карте или вне его?
- Проходит ли транзакция из обычного места или подозрительного?

Таблица альтернатив (табл. 6) показывает, как эти четыре условия взаимодействуют для того, чтобы определить, какое из трех действий произойдет:

- Нужно ли подтвердить транзакцию?
- Нужно ли связаться с держателем карты (например, чтобы предупредить его о покупке из подозрительного места)?
- Нужно ли связаться с эмитентом (например, чтобы попросить его конфисковать карту с истекшим сроком)?

Как работает таблица. Условия перечислены в **левой верхней** части таблицы, а **действия** – в **левой нижней.** Остальные столбцы

содержат бизнес-правила. Каждое правило утверждает (вкратце): **«В этой конкретной комбинации условий необходимо выполнить конкретную комбинацию действий».**

Таблица 6

Пример таблицы альтернатив (полная)

Условия	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Действитель-ный счет?	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
Активный счет?	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
В пределах лимита?	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Расположение ok?	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Действия																
Подтвердить	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Связаться с держателем	N	Y	Y	Y	N	Y	Y	Y	N	N	N	N	N	N	N	N
Связаться с эмитентом	N	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Количество столбцов, т. е. количество бизнес-правил, равняется 2 в степени числа условий: $2^4 = 16$. Когда условие строго булево (истина или ложь) и мы имеем дело с полной таблицей альтернатив (а не свернутой), то это всегда верно.

Как заполняются условия. Первое условие изменяется медленнее всего. Половина колонок – Истина, половина – Ложь. Второе сверху условие изменяется более быстро, но медленнее, чем все остальные условия. Правило заполнения: четверть – Истина, четверть – Ложь, четверть – Истина, четверть – Ложь. И наконец, четвертое условие – чередование Истина, Ложь, Истина, Ложь и т. д. **Такой шаблон заполнения позволяет гарантировать, что ничего не будет пропущено.**

Получение тестовых сценариев в этом примере простое: каждый столбец соответствует тестовому сценарию. Когда дело дойдет до выполнения тестов, будут созданы условия, которые являются входными данными для тестов. Условия «Истина/Ложь» будут заменены на реальные входные значения для номера кредитной карты, секретного кода, даты истечения срока действия и имени держа-

теля карты во время проектирования тестов или, возможно, даже во время выполнения тестов. Будут проверены действия, которые являются ожидаемыми результатами для тестов.

В некоторых случаях возникает необходимость создать более одного теста для каждого столбца. Эта возможность будет рассмотрена более детально ниже: методики эквивалентного разбиения и анализа граничных значений будут применены для расширения таблицы альтернатив.

Сворачивание таблицы альтернатив. В приведенном случае некоторые тестовые сценарии не имеют особого смысла. Например, как счет может быть нереальным, но при этом активным? Как счет может быть нереальным, но при этом в рамках лимита? Такая ситуация – подсказка, что возможно в таблице альтернатив не нужны все столбцы.

В некоторых случаях можно свернуть таблицу альтернатив, объединив некоторые столбцы, добившись ее лаконичности (иногда ощутимой). В любой ситуации, когда значение одного или нескольких конкретных условий не может повлиять на действия для двух или более комбинаций условий, можно свернуть таблицу альтернатив. Это подразумевает объединение двух или более столбцов, в которых одно или более условие не имеет возможности повлиять на действия. Столбцы, которые можно объединить, обычно, **но не всегда**, находятся рядом. По крайней мере, можно начать с рассмотрения соседних столбцов [12].

Для объединения двух и более столбцов нужно рассмотреть два или более столбца, которые приводят к одинаковому набору действий. **Необходимо помнить, что действия должны быть одинаковыми для всех действий в таблице, а не просто некоторых.** В этих столбцах некоторые условия будут одинаковыми, а другие отличными. Те, которые различаются, очевидно, не влияют на результат. Поэтому можно заменить условия, которые различаются в этих столбцах, прочерком («–»). **Прочерк обычно означает: не важно, не имеет значения или этого не может произойти при заданных условиях.**

Теперь нужно повторить этот процесс для всех следующих столбцов, которые содержат такие же комбинации действий для всех действий в таблице.

Еще один важный пункт: **нужно быть предельно внимательным, когда имеются таблицы, где в каждый момент времени может применяться более чем одно правило. Такие таблицы имеют неразделимые правила.** Они будут рассмотрены ниже.

Табл. 7 отражает ту же самую таблицу альтернатив, что и табл. 6, только свернутую для того, чтобы выявить ненужные столбцы. Наиболее заметно, что столбцы с 9 по 16 в исходной таблице альтернатив были свернуты в один столбец.

Таблица 7

Пример таблицы альтернатив (свернутая)

Условия	**1**	**2**	**3**	**5**	**6**	**7**	**9**
Действительный счет?	Y	Y	Y	Y	Y	Y	N
Активный счет?	Y	Y	Y	N	N	N	–
В пределах лимита?	Y	Y	N	Y	Y	N	–
Расположение ok?	Y	N	–	Y	N	–	–
Действия							
Подтвердить	Y	N	N	N	N	N	N
Связаться с держателем	N	Y	Y	N	Y	Y	N
Связаться с эмитентом	N	N	N	Y	Y	Y	Y

Исходная нумерация столбцов сохранена для простоты сравнения. Нельзя свернуть столбцы 2 и 3, потому что это приведет к прочерку для условий «В пределах лимита» и «Расположение ok». Если проанализировать табл. 6 или табл. 7, то можно заметить, что одно из этих условий должно быть **не** Истина, чтобы связались с держателем карты. Сворачивание правила 4 в правило 3 говорит, что если карта превысила лимит, то с держателем свяжутся независимо от расположения. Такая же логика применятся при сворачивании правила 8 в правило 7.

Формат таблицы при сворачивании не изменяется. Условия перечислены в левой верхней части таблицы, а действия – в левой нижней. Остальные столбцы содержат бизнес-правила. Каждое правило утверждает: «В этой конкретной комбинации условий (отображенных в верхней части правила, некоторые из которых не учитываются), нужно выполнить конкретную комбинацию действий (отображенную в нижней части правила, каждое из которых полностью определено)».

Количество столбцов – не более двух, возведенное в степень числа условий. Это важно, поскольку иначе свертывание нельзя было бы провести. При сворачивании таблицы удобный шаблон заполнения столбца «Истина/Ложь», присутствующий в полной таблице, исчезает. Это еще одна причина быть очень внимательным при сворачивании таблицы, потому что нельзя полагаться на шаблон или математическую формулу для проверки работы.

Неразделимые правила в таблицах альтернатив. Иногда более чем одно правило может быть применено к транзакции. В табл. 8 показан расчет комиссий по кредитной карте. Есть три условия, при этом 1, 2, 3 или все три условия могут выполниться в одном месяце. Как эта ситуация влияет на тестирование? Это немного осложняет тестирование, но можно использовать методический подход и тестирование на основе рисков для того, чтобы избежать значительных затрат.

Таблица 8

Пример неразделимых правил

Условия	**1**	**2**	**3**
Иностранная валюта?	Y	–	–
Перевод средств?	–	Y	–
Просроченная оплата?	–	–	Y
Действия			
Комиссия по иностранной валюте	Y	-	–
Комиссия за оплату	–	Y	–
Комиссия по просрочке	–	–	Y

Сначала нужно проверить таблицу альтернатив как обычную таблицу по каждому правилу и удостовериться в том, что условия, которые не относятся к этому правилу, не тестируются. Это позволяет тестировать правила отдельно – как это приходится делать в ситуациях с разделимыми правилами. Далее нужно оценить тестирование комбинаций правил. Именно «оценить», но не «протестировать все возможные комбинации правил». Это поможет избежать неперебираемого числа комбинаций, что происходит, когда тестировщики начинают тестировать комбинации условий без оценки таких тестов. Теперь в этом случае есть только 8 комбинаций – 3 условия; 2 варианта

значений для каждого условия, 2 в третьей степени равняется 8. Однако если имеется 6 условий по 5 вариантов значений для каждого, то комбинаций будет 15 625.

Определение возможных комбинаций и последующее использование весов риска этих комбинаций – способ избежать неперебираемого числа комбинаций. Нужно постараться получить важные комбинации и не думать об остальных.

Рассмотрим возможность разработки более чем одного тестового сценария для столбца таблицы альтернатив с помощью комбинирования эквивалентного разбиения и методики таблиц альтернатив. Вернемся к табл. 7, а конкретно к столбцу 9. Можно применить эквивалентное разбиение (ЭР) к вопросу «Как много интересующих – с точки зрения тестирования – случаев, когда счет не является действительным?». Как видно из рис. 5, это возможно в следующих потенциально интересных случаях:

- несоответствие номера карты и держателя;
- несоответствие номера карты и даты истечения срока;
- несоответствие номера карты и секретного кода;
- два из перечисленных несоответствия (три варианта);
- все три несоответствия.

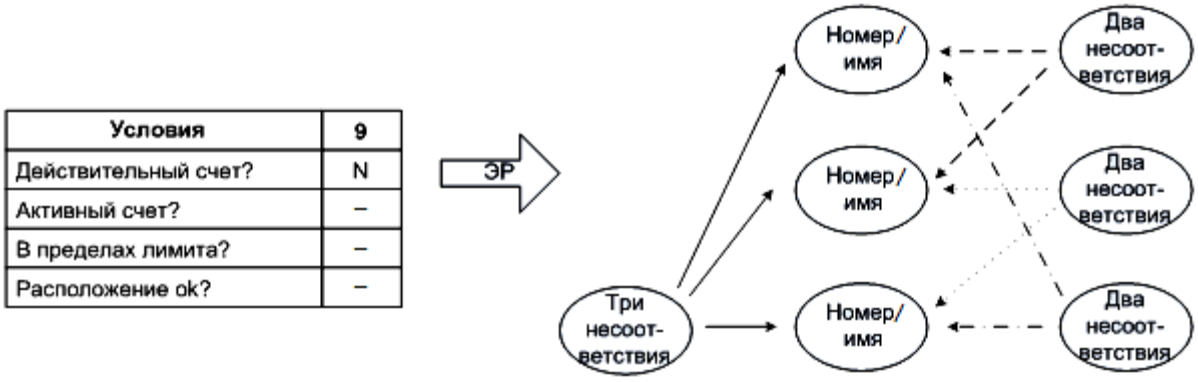

Рис. 5. Эквивалентное разбиение и таблицы альтернатив

Таким образом, для столбца 9 (см. табл. 7) может быть семь тестов.

Можно ли применить анализ граничных значений? Да, можно применить его к таблицам альтернатив, чтобы найти новые и интересные тесты. Например, «Как много интересующих тестовых значений относится к кредитному лимиту?».

Как видно из рис. 6, эквивалентное разбиение (ЭР) и анализ граничных значений (АГЗ) дают шесть случаев:

- счет начинается с нулевого баланса;
- баланс будет находиться в пределах нормы после транзакции;
- баланс будет точно на границе лимита после транзакции;
- баланс будет точно за границей лимита после транзакции;
- баланс был точно на границе лимита до транзакции (что точно приведет к его превышению, если транзакция завершится);
- баланс будет на максимальном значении овердрафта после транзакции (что может быть недопустимо).

Условия	1	2	3	5	6	7
Действительный счет?	Y	Y	Y	Y	Y	Y
Активный счет?	Y	Y	Y	N	N	N
В пределах лимита?	Y	Y	N	Y	Y	N
Расположение ok?	Y	N	-	Y	N	-

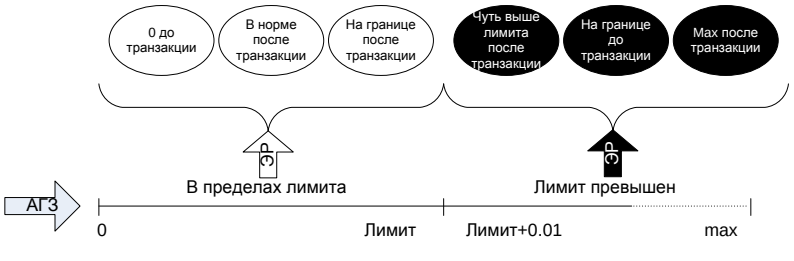

Рис. 6. Анализ граничных значений и таблицы альтернатив

Комбинируя указанные методики ЭР и АГЗ с таблицей альтернатив, видим, что тестов на проверку условия «Сверх лимита» будет больше, чем столбцов – на один. Иными словами, будет четыре теста на проверку «В пределах лимита» и три теста на проверку «Сверх лимита». Это верно до тех пор, пока не потребуется убедиться, что каждый класс эквивалентности «В пределах лимита» представлен в подтвержденной транзакции. В этом случае столбец 1 будет представлен не одним тестом, а тремя [12].

4.4. Диаграммы причинно-следственных связей

При сворачивании таблицы альтернатив диаграмма причинно-следственных связей может помочь удостовериться в том, что случайно не свернуты столбцы, которые не должны были сворачиваться.

Процесс создания диаграммы причинно-следственных связей из таблицы альтернатив или наоборот достаточно прост. **Для создания диаграммы из таблицы сначала нужно выписать все условия слева. Затем справа перечислить все действия. Теперь для каждо-**

го действия нужно определить, какие комбинации условий приводят к действию. Одно или несколько условий связываются с действием, используя булевы операторы, которые показаны на рис. 7. Затем необходимо повторить этот процесс для всех действий в таблице альтернатив.

Важно всегда помнить, что любая таблица альтернатив может быть трансформирована в диаграмму причинно-следственных связей и наоборот. Поэтому, что использовать для проектирования тестов, – решать тест-аналитику.

При использовании диаграмм причинно-следственных связей необходимо построить так называемую «таблицу истинности», содержащую все возможные комбинации и гарантирующую как минимум один тест на каждый столбец таблицы истинности. **Это и является минимальным критерием покрытия.**

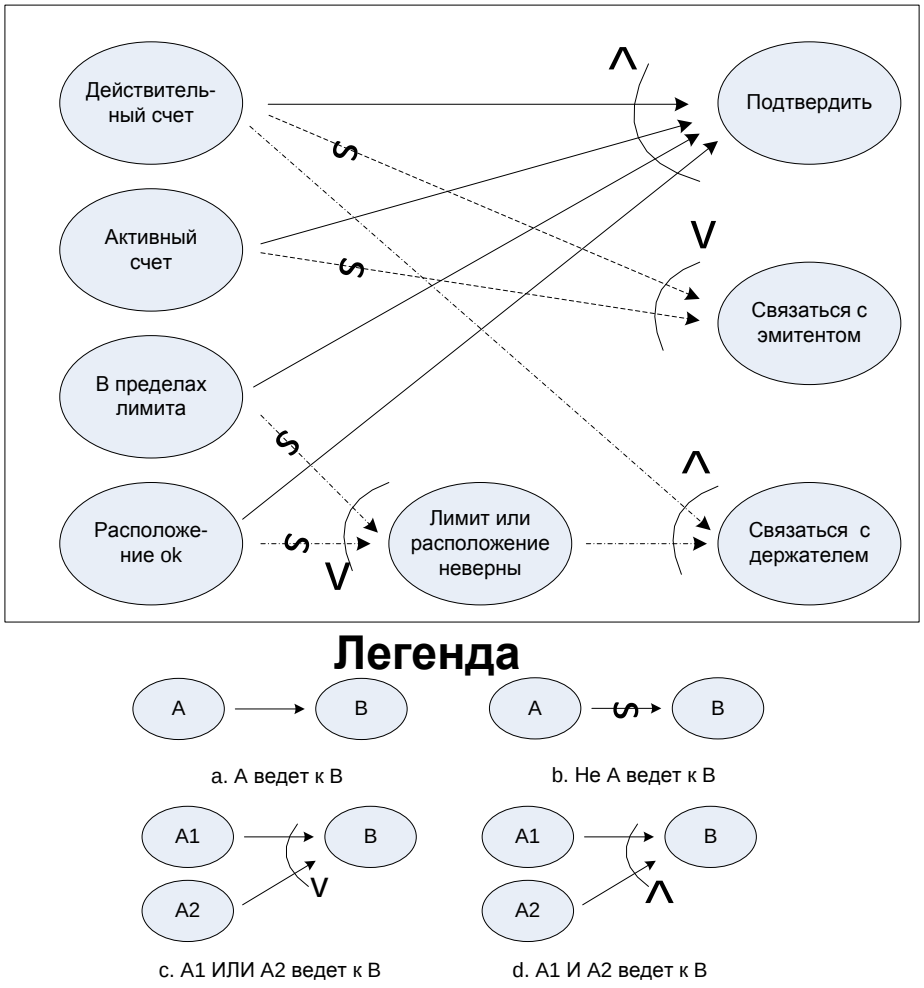

Рис. 7. Пример диаграммы причинно-следственных связей

Если нужно построить таблицу альтернатив по диаграмме причинно-следственных связей, сначала нужно перечислить все условия в левой верхней части «чистой» таблицы. Затем перечислить все действия в левой нижней части таблицы, под условиями. Следуя приведенному выше шаблону, необходимо сгенерировать все возможные комбинации условий. Далее согласно диаграмме причинно-следственных связей нужно определить действия, которые должны быть выполнены и не выполнены для каждой комбинации условий. Когда все ячейки действий полностью заполнены, можно свернуть таблицу, если требуется.

На рис. 7 показана диаграмма причинно-следственных связей, соответствующая приведенной таблице альтернатив (см. табл. 6 – 7). Может возникнуть вопрос: «Какой именно, полной или свернутой?». Ответ – обеим. Полная и свернутая таблицы логически эквивалентны, если только при сворачивании не было допущено ошибки.

На рис. 7 показана легенда, указывающая, как читать операции:

a. Простая причинно-следственная связь: если A – истина, то произойдет B, или, другими словами, A ведет к B.

b. Отрицание: если A – не истина, то произойдет B, или, другими словами, не A ведет к B.

c. Операция ИЛИ: если A1 или A2 – истина, то произойдет B, или, другими словами, A1 или A2 ведет к B.

d. Операция И: если A1 и A2 одновременно истина, то произойдет B, или, другими словами, A1 и A2 ведут к B.

Рассмотрим связи между условиями и действиями. **Сплошные линии с оператором И** показывают, что **все** четыре условия должны быть удовлетворены, чтобы транзакция была подтверждена. **Пунктирные линии с отрицанием и оператором ИЛИ** показывают, что если счет недействительный или счет не активный, нужно связаться с эмитентом.

Штрихпунктирные линии имеют более сложную интерпретацию. Для начала скомбинированы условия «Внутри лимита» и «Расположение ok» с помощью операторов отрицания и ИЛИ для того, чтобы создать промежуточное условие «Лимит или расположение неверны». Затем это условие комбинируется с условием «Действительный счет», чтобы сказать, что если лимит превышен или имеет место ситуация с подозрительным расположением, но счет действительный, то нужно связаться с держателем карты [12].

4.5. Тестирование на основе состояний и диаграммы переходов состояний

Рассмотрим тестирование на основе состояний. Такая методика идеально подходит в ситуациях, когда имеются последовательности происходящих событий и условий, которые соответствуют этим событиям, а соответствующая обработка конкретной ситуации зависит от событий или условий, которые произошли раньше. В некоторых случаях последовательность событий потенциально может быть бесконечной, что, естественно, выходит за рамки возможностей тестирования. Необходима такая методика разработки тестов, которая позволяла бы работать с произвольно длинными последовательностями событий.

В основе такой методики лежит **диаграмма или таблица переходов состояний. Диаграмма или таблица связывает начальные состояния, события и условия с конечными состояниями и действиями.** Диаграмма состояний по существу является графом специального вида, который представляет некоторый автомат. Иными словами, существует некий статус-кво, и система находится в текущем состоянии. Затем происходит некоторое событие, которое система должна обработать. На обработку этого события может влиять одно или более условий. Комбинация событий и условий вызывает срабатывание перехода: или из текущего состояния в новое, или из текущего снова в текущее состояние. В случае перехода система предпринимает одно или более действий.

Имея такую модель, можно генерировать тесты, которые проходят по состояниям и переходам. Входные данные вызывают события и создают условия, в то время как ожидаемыми результатами теста являются новые состояния и действия, предпринимаемые системой.

В тестировании на основании состояний применяются различные критерии покрытия. Слабейший критерий требует, чтобы тесты прошли каждое состояние и проверили каждый переход. Такой критерий может применяться к диаграммам переходов состояний. Более жесткий критерий покрытия – когда каждая строка в таблице переходов состояний покрыта как минимум одним тестом [12].

Еще один потенциально более жесткий критерий покрытия требует, чтобы каждая последовательность переходов длины N и меньше была покрыта как минимум одним тестом. N может быть равно 1, 2, 3, 4 и больше. Это называется **«Покрытием переходов Чау» («Chow's switch coverage»)**, по имени профессора Чау, разработавшего методику, или «Покрытием N-1 переходов» – по достигнутому уровню покрытия. Если покрыть все переходы единичной длины, тогда «Покрытие N-1 переходов» означает «Покрытие 0 переходов». Следует заметить, что это тоже нижний уровень покрытия, который рассматривался выше. Если покрыть все переходы длины 1 и 2, тогда «Покрытие N-1 переходов» означает «Покрытие 1-го перехода». Это, естественно, более высокий уровень покрытия по сравнению с нижним [2].

Но «Покрытие 1-го перехода» необязательно выше уровнем, чем покрытие каждой строки, потому что таблица переходов ведет к тестированию комбинаций событий/условий, чего не происходит в диаграмме переходов состояний. Так называемые «переходы» в «Покрытии N-1 переходов» получаются из диаграммы переходов, а не таблицы.

Какова же гипотеза ошибки при тестировании на основе состояний? Нужно искать ситуации, в которых происходят неверные действия или переходы в неверные состояния в ответ на конкретное событие при заданном наборе условий, основанных на истории комбинаций событий/условий до текущего момента.

Согласно глоссарию ISTQB**, тестирование переходов состояний (state transition testing) –** это разработка тестов методом черного ящика, в котором сценарии тестирования строятся на основе выполнения корректных и некорректных переходов состояний [2].

На рис. 8 изображена диаграмма переходов состояний для выбора и покупки товаров из приложения электронной коммерции. Она показывает взаимодействие системы с клиентом **с точки зрения клиента.** Отметим ключевые элементы диаграммы переходов состояний в целом и конкретные особенности указанной диаграммы.

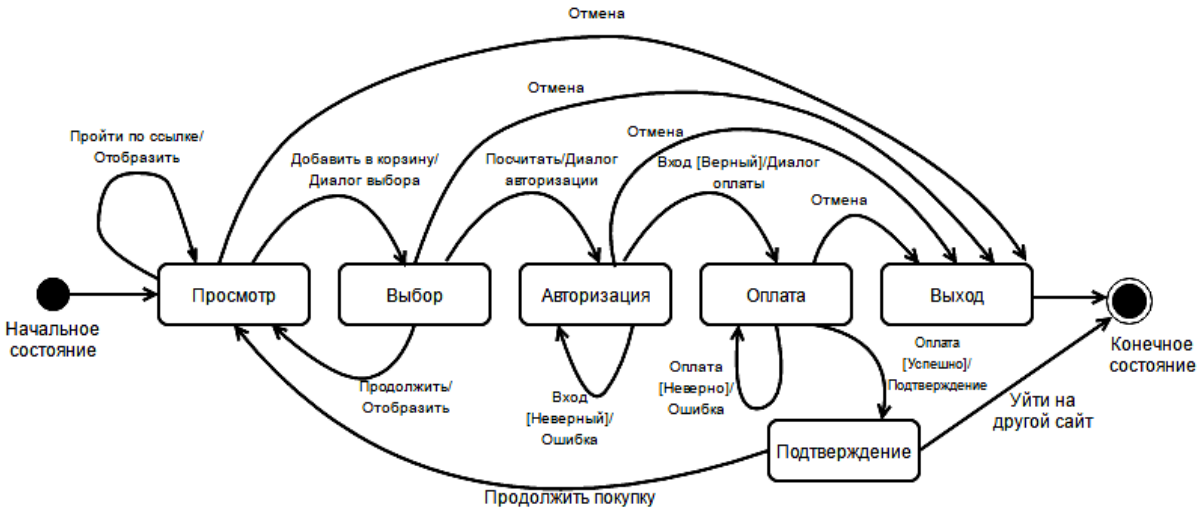

Рис. 8. Пример диаграммы переходов состояний

Слева на рис. 8 изображен небольшой **элемент в виде кружка и стрелки – «Начальное состояние».** Такая нотация показывает, что с точки зрения клиента транзакция начинается с просмотра веб-сайта. Можно переходить по ссылкам и просматривать каталог товаров, оставаясь в состоянии просмотра. **Стрелка в виде петли** над состоянием просмотра отражает как раз такой процесс. **Узлы в виде прямоугольников со скругленными углами** представляют состояния. **Стрелки отражают переходы.**

В состояние «Выбор» можно войти путем добавления товара в корзину. **«Добавить в корзину» – событие.** Затем система отображает «Диалог выбора», в котором спрашивает пользователя о количестве товаров, которое он хочет приобрести, а также о любой другой информации, которая необходима для добавления товара в корзину. Как только это сделано, пользователь может сообщить системе, что он хочет продолжить покупку. В этом случае система снова отображает домашнюю страницу, и **пользователь возвращается в состояние просмотра. С точки зрения нотации следует заметить, что действия, выполняемые системой, записаны после символа наклонной черты на стрелке перехода.**

Существует альтернатива: покупатель может выбрать оформление покупки. В этом случае он **входит в состояние авторизации.** Он вводит регистрационную информацию. **К этой информации применимо условие: верна ли регистрационная информация.** Если не

верна, то система отображает **ошибку**, и пользователь остается в состоянии авторизации. Если верна, то система отображает **первую страницу в диалоге оплаты. Заметим, что «Верный» и «Неверный», отображенные в квадратных скобках, с точки зрения нотации обозначают условия.**

В состоянии оплаты система будет отображать страницы, а покупатель вводить данные об оплате. Корректность вводимой информации будет определять возможность завершения и подтверждения транзакции системой. Как только транзакция подтверждена, пользователь может продолжить покупку либо пойти на другой сайт. **Также заметим, что пользователь всегда может отменить транзакцию и уйти на другой сайт.**

Часто при разборе тестирования на основе состояний возникает вопрос: «Как различить состояние, событие и действие?» Основные отличия следующие:

- в **состоянии** объект находится до тех пор, пока что-то не произойдет – что-то внешнее по отношению к самому объекту, обычно – вызывающее переход. В состоянии объект может существовать бесконечно долго;
- **событие** происходит либо сразу, либо в ограниченный, конечный период времени. Это что-то, что наступает, происходит вне, – и вызывает переход;
- **действие** – это ответ системы во время перехода. Действие, как и событие, либо мгновенно, либо требует ограниченного, конечного времени.

Известно, что иногда одну ситуацию можно отразить поразному, особенно когда одно состояние или действие может быть разбито на последовательность элементарных состояний, событий и действий. Чуть позже рассмотрим такой случай на примере разбиения состояния оплаты на **подсостояния.**

И наконец, заметим, что **диаграмма нарисована с точки зрения клиента.** Если изобразить ее с точки зрения системы, то она выглядела бы по-другому. **Поддержание постоянной и единой точки зрения критично для изображения таких диаграмм, иначе будут происходить абсурдные вещи.**

Тестирование на основе состояний использует формальную модель, поэтому имеется формальная процедура для получения тестов из модели. Приведенный ниже список показывает **процедуру, которую можно использовать для получения тестов, позволяющих достигнуть покрытия состояний/переходов (т. е. «покрытие 0 переходов»).**

1. Введем правило для того чтобы понять, где тестовая процедура или шаг должны начинаться и где заканчиваться. В качестве примера утвердим, что тестовая процедура должна начинаться в начальном состоянии и может закончиться только в конечном состоянии. Причина использования слов «может» или «должен» для определения окончания состоит в том, что в ситуациях, где начальное и конечное состояния – одно и то же, может потребоваться разрешить последовательности состояний и переходов, которые проходят через начальное состояние более чем один раз.

2. Начиная с разрешенного состояния начала теста, определим последовательность комбинаций событий и условий, которые приводят к разрешенному состоянию окончания теста. Для каждого перехода, который происходит, зафиксируем ожидаемое действие, которое должна произвести система. **Это ожидаемый результат.**

3. При прохождении каждого состояния и каждого перехода помечаем его как покрытый. Простейший способ для этого – распечатать диаграмму переходов состояний и, используя карандаш или маркер, помечать каждый узел и стрелку по мере покрытия.

4. Повторять шаги 2 и 3 до тех пор, пока не пройдены все состояния и все переходы. Иными словами, пока каждое состояние и переход не будут помечены карандашом или маркером.

Такая процедура позволяет генерировать логические тестовые сценарии. Для создания именованных тестовых сценариев (тестовых сценариев низкого уровня) необходимо сгенерировать фактические входные и выходные данные. Здесь рассматривается создание логических тестовых сценариев для иллюстрации методики, но нужно помнить, что в определенной точке перед выполнением тестов должны быть созданы именованные тестовые сценарии.

Рассмотрим этот процесс на примере проанализированного ранее приложения электронной коммерции. На рис. 9 пунктирными линиями показаны покрытые состояния и переходы.

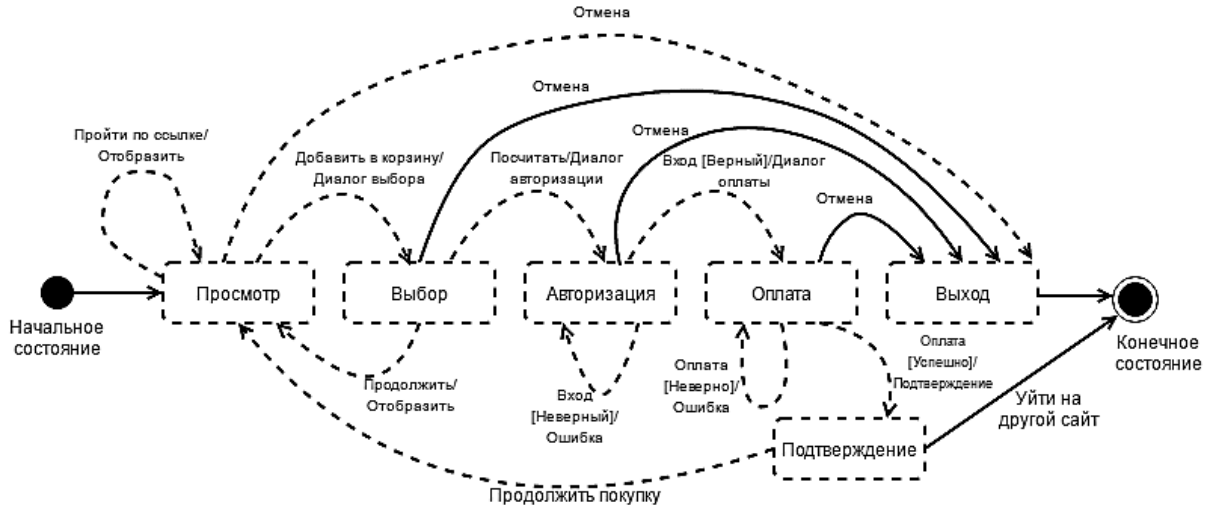

Рис. 9. Проверка покрытия 1

На рис. 9 нужно отметить две вещи: 1) **правило, которое гласит, что тест должен начинаться в начальном состоянии и заканчиваться в конечном;** 2) **какие шаги включены в тест:** («Просмотр», «Пройти по ссылке», «Отобразить», «Добавить в корзину», «Диалог выбора», «Продолжить», «Отобразить», «Добавить в корзину», «Диалог выбора», «Посчитать», «Диалог авторизации», «Вход [Неверный]», «Ошибка», «Вход [Верный]», «Диалог оплаты», «Оплата [Неверно]», «Ошибка», «Оплата [Успешно]», «Подтверждение», «Продолжить покупку», «Отобразить», «Отмена», «Выход»).

Теперь проверим полноту покрытия, которое отслеживается на диаграмме переходов состояний. Как видно, покрыты все состояния и большинство переходов, но не все. Необходимо создать еще несколько тестов.

На рис. 10 изображено покрытие, которое было достигнуто путем дополнительных тестовых процедур, которые покрывают диаграмму переходов состояний:

1) («Просмотр», «Пройти по ссылке», «Отобразить», «Добавить в корзину», «Диалог выбора», «Продолжить», «Отобразить», «Добавить в корзину», «Диалог выбора», «Посчитать», «Диалог авторизации», «Вход [Неверный]», «Ошибка», «Вход [Верный]», «Диалог оплаты», «Оплата [Неверно]», «Ошибка», «Оплата [Успешно]», «Подтверждение», «Продолжить покупку», «Отобразить», «Отмена», «Выход»);

2) («Просмотр», «Добавить в корзину», «Диалог выбора», «Отмена», «Выход»);

3) («Просмотр», «Добавить в корзину», «Диалог выбора», «Посчитать», «Диалог авторизации», «Отмена», «Выход»);

4) («Просмотр», «Добавить в корзину», «Диалог выбора», «Посчитать», «Диалог авторизации», «Вход [Верный]», «Диалог оплаты», «Отмена», «Выход»);

5) («Просмотр», «Добавить в корзину», «Диалог выбора», «Продолжить», «Отобразить», «Добавить в корзину», «Диалог выбора», «Посчитать», «Диалог авторизации», «Вход [Верный]», «Диалог оплаты», «Оплата [Успешно]», «Подтверждение», «Уйти на другой сайт»).

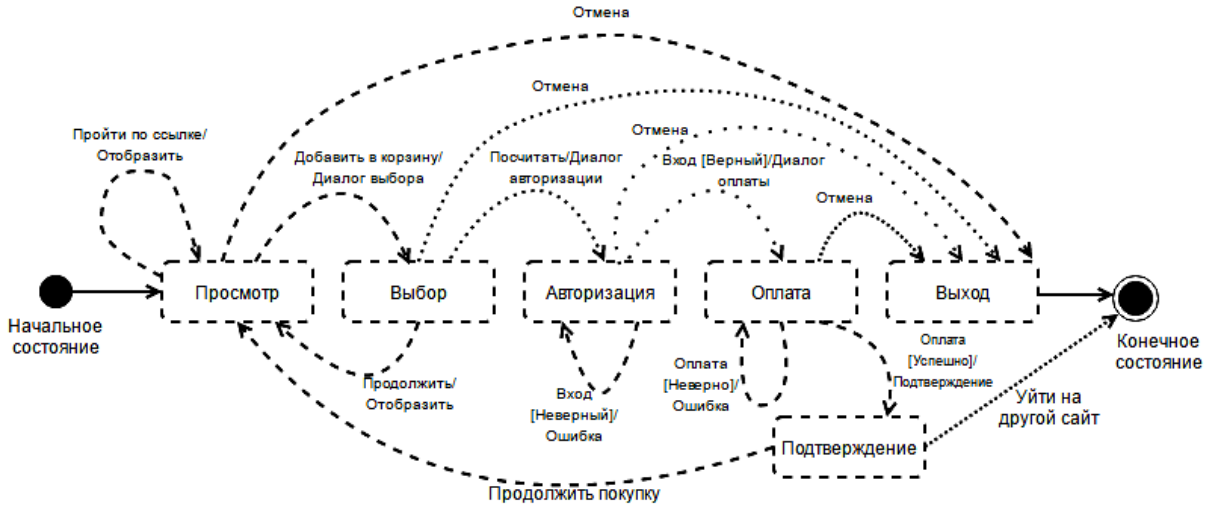

Рис. 10. Завершающая проверка покрытия

Рис. 10 отражает проверку покрытия состояний и переходов. Процесс нельзя считать завершенным до тех пор, пока каждое состояние и каждый переход не помечены, как сделано на этом рисунке.

Суперсостояния и подсостояния. В некоторых случаях полезно преобразовать отдельное состояние в суперсостояние, состоящее из двух или более подсостояний. На рис. 11 отражено, что состояние оплаты из примера расширено тремя подсостояниями.

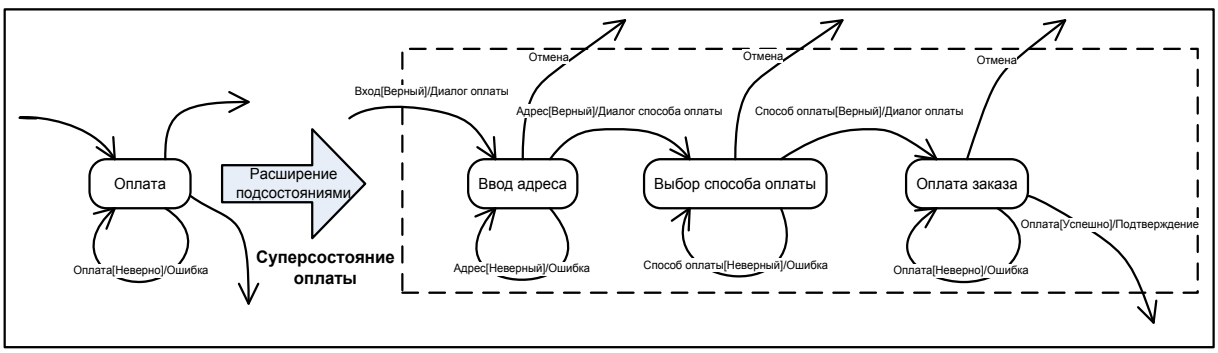

Рис. 11. Суперсостояния и подсостояния

Правило базового покрытия здесь простое: покрыть все переходы в суперсостояние, все переходы из суперсостояния, все подсостояния и все переходы внутри суперсостояния.

Заметим, что для нашего примера это увеличит число тестов, поскольку теперь имеется три перехода «Отмена» из состояния «Оплата» в состояние «Выход» вместо одного. Это также увеличит количество элементов в тестах, т. е. станет больше событий и действий, за счет того, что нужно убедиться, что протестированы как минимум три различных типа сущностей неверной оплаты [12].

4.6. Таблицы переходов состояний

Таблицы переходов состояний полезны и бизнес-аналитикам, и проектировщикам систем, так как позволяют учитывать комбинации состояний, событий и условий, которые можно пропустить.

Для построения таблицы переходов состояний сначала нужно перечислить все состояния из диаграммы переходов состояний (рис. 12) из примера, который рассматривался в предыдущем разделе.

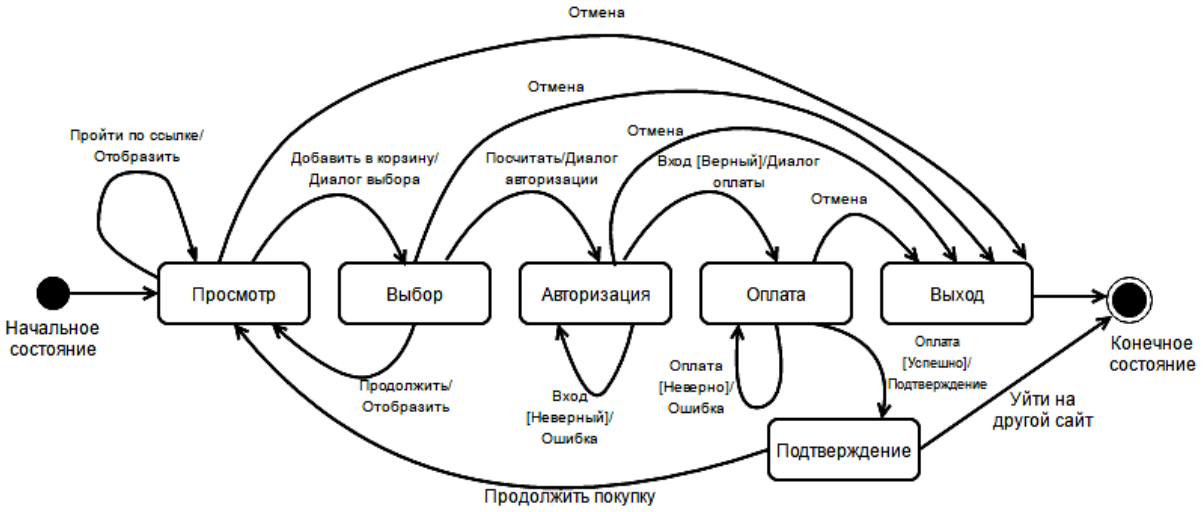

Рис. 12. Диаграмма переходов состояний для приложения электронной коммерции

Далее необходимо перечислить все комбинации событий/условий из диаграммы переходов состояний. Затем создается таблица, в которой каждая строка соответствует комбинации состояния и каждого события/условия. Каждая строка имеет четыре поля:

- текущее состояние;
- событие/условие;
- действие;
- новое состояние.

Для строк, где диаграмма переходов состояний определяет действие и новое состояние для заданных комбинаций текущего состояния и события/условия, можно заполнить два поля (действие и новое состояние) из диаграммы переходов состояний. **Однако для остальных строк в таблице они не определены.**

Теперь можно обращаться к бизнес-аналитикам, проектировщикам системы или другим участникам, чтобы уточнить, что же должно произойти в каждой из этих ситуаций. Скорее всего вам скажут, что такого никогда не произойдет. Но тест-аналитики знают, что это значит, а потому их основная задача – показать, как это может случиться.

На рис. 13 изображена часть таблицы, которую можно создать для примера приложения электронной коммерции, который рассматривался в предыдущем разделе.

Состояния		События/Условия		Текущее состояние	Событие/Условие	Действие	Новое состояние
				Просмотр	Пройти по ссылке	Отобразить	Просмотр
		Пройти по ссылке		Просмотр	Добавить в корзину	Диалог выбора	Выбор
		Добавить в корзину		**Просмотр**	**Продолжить**	**Не определено**	**Не определено**
		Продолжить		**Просмотр**	**Выписать**	**Не определено**	**Не определено**
Просмотр		Выписать		**Просмотр**	**Вход [неверный]**	**Не определено**	**Не определено**
Выбор		Вход [неверный]		**Просмотр**	**Вход [верный]**	**Не определено**	**Не определено**
Авторизация		Вход [верный]		**Просмотр**	**Оплата [неверно]**	**Не определено**	**Не определено**
Оплата	×	Оплата [неверно]	=	**Просмотр**	**Оплата [успешно]**	**Не определено**	**Не определено**
Подтверждение		Оплата [успешно]		**Просмотр**	**Отмена**	**Нет действия**	**Выйти**
Выйти		Отмена		**Просмотр**	**Продолжить покупку**	**Не определено**	**Не определено**
		Продолжить покупку		**Просмотр**	**Уйти на другой сайт**	**Не определено**	**Не определено**
		Уйти на другой сайт		**Выбор**	**Пройти по ссылке**	**Не определено**	**Не определено**
				{53 строки, сгенерированных по шаблону, не показаны}			
				Выйти	**Уйти на другой сайт**	**Не определено**	**Не определено**

Рис. 13. Пример таблицы переходов состояний

Здесь имеются **6 состояний:**

- просмотр;
- выбор;
- авторизация;
- оплата;
- подтверждение;
- выход.

Также имеются **11 условий/событий:**

- пройти по ссылке;
- добавить в корзину;
- продолжить;
- выписать;
- вход [Неверный];
- вход [Верный];
- оплата [Успешно];
- оплата [Неверно];
- отмена;
- продолжить покупку;
- уйти на другой сайт.

Указанное выше количество условий и состояний означает, что полная таблица переходов состояний будет иметь 66 строк – по одной строке для каждой пары «состояние – событие/условие» [2].

Для получения набора тестов, который покрывает таблицу переходов состояний, можно следовать следующей процедуре. Следует заметить, что мы строим уже существующий набор тестов, созданный на основе диаграммы переходов состояний, для достижения покрытия состояний/переходов или «покрытия 0 переходов».

1. Начнем с набора тестов (включая правила начального и конечного состояния), полученных из диаграммы переходов состояний, дающей покрытие состояний/переходов.

2. Построим таблицу переходов состояний и убедимся, что тесты покрывают все «определенные» строки (определены действие и новое состояние). Если они не покрывают, то либо неверно сгенерирован существующий набор тестов, либо неверно построена таблица переходов состояний, либо неверна диаграмма переходов состояний. Нельзя продолжать до тех пор, пока проблема не найдена и не устра-

нена, включая создание таблицы переходов состояний или набора тестов заново, если потребуется.

3. Выберем тесты, покрывающие состояние, для которого в таблице существует одна или более «неопределенных» строк. Изменим тесты и попытаемся создать комбинацию событий/условий для «неопределенной» строки. Заметим, что действие в этом случае не определено.

4. По мере изменения тестов помечаем строки как покрытые. Проще всего сделать это, взяв распечатанную версию таблицы и, используя карандаш или маркер, помечать каждую строку по мере покрытия.

5. Повторяем шаги 3 и 4, пока все строки не будут покрыты.

Такая процедура создает **логические тестовые сценарии.**

Для наглядности рассмотрим **пример.**

Существующий тест: («Просмотр», «Добавить в корзину», «Диалог выбора», «Выписать», «Диалог авторизации», «Вход [Верный]», «Диалог оплаты», «Отмена», «Выход»).

Измененные тесты:

- («Просмотр», *попытка:* «Продолжить», *неопределенное действие*, «Добавить в корзину», «Диалог выбора», «Выписать», «Диалог авторизации», «Вход [Верный]», «Диалог оплаты», «Отмена», «Выход»);
- («Просмотр», *попытка:* «Выписать», *неопределенное действие*, «Добавить в корзину», «Диалог выбора», «Выписать», «Диалог авторизации», «Вход [Верный]», «Диалог оплаты», «Отмена», «Выход»);
- можно создать еще шесть измененных тестов, но приводить их здесь не будем.

Не пытайтесь покрыть «неопределенные» комбинации событий/условий более чем для одного состояния в любом тесте, **поскольку вы наверняка не знаете, останется ли система пригодной для тестирования после этого!**

В лучшем случае «неопределенная» комбинация событий/условий останется проигнорированной или будет получен отказ с осмысленным сообщением об ошибке, а после этого процесс продолжится в нормальном режиме.

Это пример получения тестов на основании таблицы, построенной для примера с приложением электронной коммерции. Как можно заметить, в самом начале был выбран тест из набора полученных ранее тестов: («Просмотр», «Добавить в корзину», «Диалог выбора», «Выписать», «Диалог авторизации», «Вход [Верный]», «Диалог оплаты», «Отмена», «Выход»).

Затем мы начали создавать измененные тесты для покрытия только «неопределенных» событий/условий в состоянии «Просмотр». Кроме того, в этом примере не показано еще шесть измененных тестов. Как видно, создание таких тестов – механический процесс. **Если вы аккуратны при отслеживании покрытых строк, например с использованием маркера, то почти невозможно пропустить тест.**

Также можно заметить, что **в каждый тест включена только одна «неопределенная» комбинация событий/условий.** Почему? **Это разновидность правила эквивалентного разбиения, согласно которому нельзя создавать «некорректные» тестовые сценарии, которые включают более одного «некорректного» тестового данного.** В нашем случае каждая строка представляет собой «некорректные» тестовые данные. Если попытаться покрыть две строки одним тестом, то мы не можем быть уверены, что система останется пригодной для тестирования после первого «некорректного» тестового данного.

Мы пометили действие как «неопределенное». Какое поведение системы в этих условиях можно считать идеальным? **Лучше, если «неопределенные» события/условия будут проигнорированы или – еще лучше – отклонены с осмысленным сообщением об ошибке.** После этого система должна функционировать нормально. При отсутствии какого-либо решения бизнес-аналитика, проектировщиков системы или другого влиятельного специалиста, а также спецификации требований мы вправе принять позицию, когда любой иной исход считается дефектом, включая непонятные сообщения об ошибках наподобие «То, что только что произошло, – не могло произойти» [12].

Покрытие переходов. Рассмотрим концепцию покрытия переходов на примере все того же приложения электронной коммерции. На рис. 14 изображена последовательность переходов, которая получена с помощью покрытия переходов. Диаграмма схожа с рис. 15 с

той лишь разницей, что на ней имена состояния заменены на буквы, а имена переходов – на цифры. **Теперь пара «состояние – переход» может быть определена буквой и цифрой.**

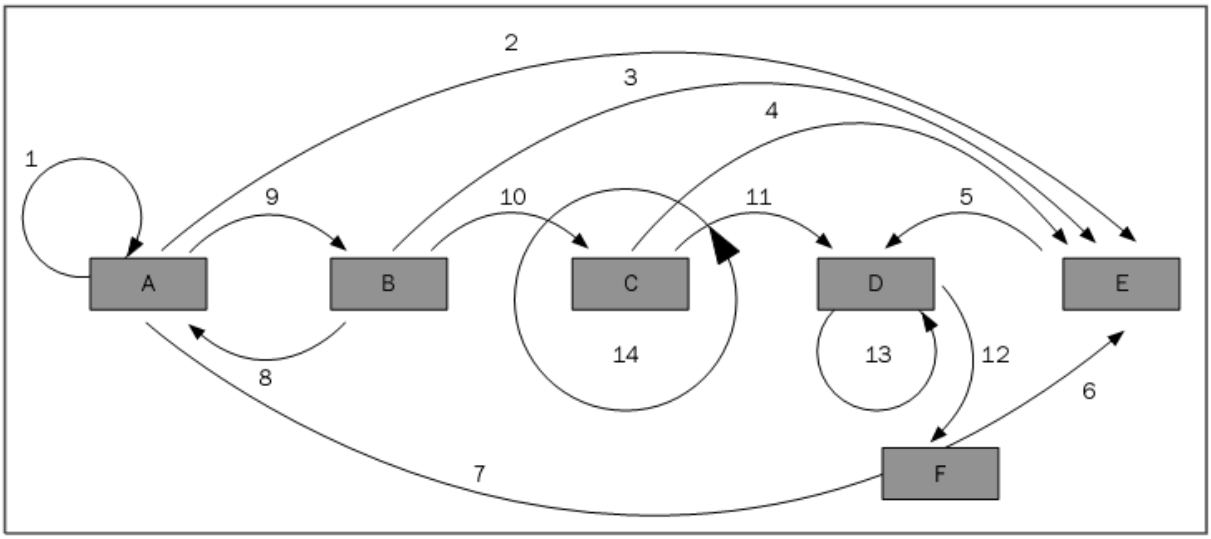

Рис. 14. Диаграмма состояний и переходов для приложения электронной коммерции с заменами имён состояний на буквы, а имён переходов – на цифры

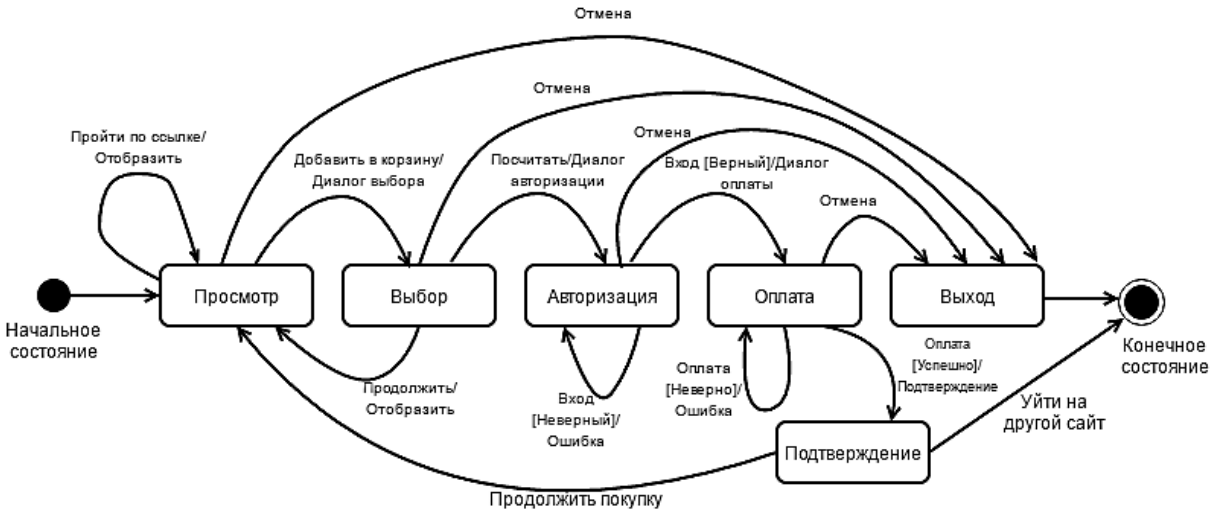

Рис. 15. Диаграмма состояний и переходов для приложения электронной коммерции

Не будем перечислять все комбинации в таблице, представленной на рис. 16, поскольку исходя из диаграммы понятно, какое состояние следует после определенного перехода. Существует только одна стрелка с указанным номером, которая ведет из состояния с указанной буквой, и эта стрелка ведет точно в одно состояние [12].

0-switch			1-switch								
A1	A2	A9	A1A1	A1A2	A1A9				A9B10	A9B8	A9B3
B10	B8	B3	B10C14	B10C11	B10C4	B8A1	B8A2	B8A9			
C14	C11	C4	C14C14	C14C11	C14C4	C11D13	C11D12	C11D5			
D13	D12	D5	D13D13	D13D12	D13D5	D12F6	D12F7				
F6	F7					F7A1	F7A2	F7A9			

Рис. 16. Таблица покрытия переходов

Контрольные вопросы

1. Что такое эквивалентное разбиение?
2. Для каких данных могут быть определены классы эквивалентности?
3. В чем суть методики анализа граничных значений?
4. Как связан анализ граничных значений с эквивалентным разбиением?
5. Как построить таблицу альтернатив?
6. Что такое неразделимые правила в таблицах альтернатив?
7. Как производится сворачивание таблицы альтернатив?
8. Эквивалентны ли полная и свернутая таблицы альтернатив?
9. Можно ли комбинировать техники проектирования тестов? Приведите примеры.
10. В чем разница между состоянием, событием и действием?
11. Что такое тестирование переходов состояний?
12. Как построить таблицу переходов состояний?
13. Что необходимо предпринять тест-аналитику, если поведение системы не описано в требованиях?
14. Можно ли создавать тесты, в которых имеется более одного некорректного тестового данного? Почему?
15. Что такое суперсостояние?
16. Что такое подсостояние?
17. Что такое Chow’s switch coverage?
18. Можно ли преобразовать диаграмму причинно-следственных связей в таблицу альтернатив и наоборот?
19. Нарушается ли общее правило заполнения таблицы альтернатив при её сворачивании?
20. Для чего сворачивают таблицу альтернатив?

5. АВТОМАТИЗАЦИЯ

В данном разделе рассмотрены основные понятия и особенности автоматизированного тестирования. Как стало ясно из предыдущих разделов, количество тестовых сценариев, как правило, велико. Кроме того, прогон одних и тех же тестов может осуществляться многократно, например, при регрессионном тестировании. Для того чтобы ускорить и упростить работу специалиста по тестированию, в таких случаях зачастую прибегают к автоматизации.

Автоматизированное тестирование программного обеспечения (**Software Automation Testing**) – это процесс верификации программного обеспечения, при котором основные функции и шаги теста, такие как запуск, инициализация, выполнение, анализ и выдача результата, выполняются автоматически при помощи инструментов для автоматизированного тестирования.

Во многих случаях специалист по функциональному тестированию, проверяющий программное обеспечение вручную, имеет и квалификацию специалиста по автоматизации тестирования. Иногда эти роли могут брать на себя два разных специалиста.

Специалист по автоматизированному тестированию программного обеспечения (**Software Automation Tester**) – это технический специалист (тестировщик или разработчик программного обеспечения), обеспечивающий создание, отладку и поддержку работоспособного состояния тест-скриптов, тестовых наборов и инструментов для автоматизированного тестирования. Для выполнения задач автоматизации специалист по автоматизированному тестированию использует специальное программное обеспечение, посредством которого осуществляется создание, отладка, выполнение и анализ результатов прогона тест-скриптов.

Набор инструкций для автоматической проверки определенной части программного обеспечения называется тест-скриптом (**Test Script),** или просто **автотестом**. Несколько автотестов образуют тестовый набор (**Test Suite**), который представляет собой комбинацию тест-скриптов для проверки определенной части программного обеспечения, объединенной общей функциональностью или целями, преследуемыми запуском данного набора.

Автотесты могут быть просто написаны, но еще не сгруппированы в наборы для запуска. Порядок запуска определяется в ходе планирования тестирования на основе оценки рисков и приоритизации. **Тесты для запуска (Test Run)** – это комбинация автотестов или тестовых наборов для последующего совместного запуска (последовательного или параллельного, в зависимости от преследуемых целей и возможностей инструмента для автоматизированного тестирования).

5.1. Цели, преимущества и недостатки автоматизации

Целесообразность автоматизации тестирования определяется ответом на вопросы: «Какие преимущества могут быть получены от автоматизации?» и «Больше ли в данном конкретном случае преимуществ, чем недостатков?». Если в разрабатываемом продукте отсутствуют такие модули, для которых автоматизация тестирования дает больше преимуществ, чем недостатков, либо недостатки для конкретного случая неприемлемы, то автоматизацию можно считать нецелесообразной.

Основными целями автоматизации процесса тестирования являются:

- сокращение сроков тестирования;
- высвобождение людских ресурсов;
- упрощение и ускорение процесса регрессионного тестирования;
- увеличение покрытия системы тестами и др.

При принятии решения стоит помнить, что альтернативой автоматизации является ручное тестирование, у которого также есть свои недостатки.

Рассмотрим подробнее преимущества и недостатки автоматизированного тестирования.

Преимущества автоматизации тестирования:

1) **повторяемость** – все написанные тесты всегда будут выполняться однообразно, т. е. исключен «человеческий фактор». Тестировщик не пропустит тест по неосторожности и ничего не напутает в результатах;

2) **выполнение без вмешательства** – во время выполнения тестов инженер-тестировщик может заниматься другими полезными

делами, или тесты могут выполняться в нерабочее время (этот метод предпочтительнее, так как нагрузка на локальные сети ночью снижена);

3) **скорость выполнения** – автоматизированному скрипту не нужно сверяться с инструкциями и документациями, что сильно экономит время выполнения;

4) **меньшие затраты на поддержку** – когда автоматические скрипты уже написаны, на их поддержку и анализ результатов требуется, как правило, меньшее время, чем на проведение того же объема тестирования вручную.

Недостатки автоматизации тестирования:

1) **большие затраты на разработку** – разработка автоматизированных тестов – это сложный процесс, так как фактически идет разработка приложения, которое тестирует другое приложение. В сложных автоматизированных тестах также есть фреймворки, утилиты, библиотеки и прочее. Естественно, все это нужно тестировать и отлаживать, а это требует времени и людских ресурсов;

2) **повторяемость** – все написанные тесты всегда будут выполняться однообразно. Это одновременно является и недостатком, так как тестировщик, выполняя тест вручную, может обратить внимание на некоторые детали и, проведя несколько дополнительных операций, найти дефект. Скрипт этого сделать не может;

3) **затраты на поддержку** – несмотря на то, что в случае автоматизированных тестов они меньше, чем затраты на ручное тестирование того же функционала, – они все же есть. Чем чаще изменяется приложение, тем они выше. Большой проблемой, как правило, становится редизайн пользовательского интерфейса приложения, который влечёт за собой практически полное переписывание автоматизированных тестов;

4) **пропуск мелких ошибок** – автоматический скрипт может пропускать мелкие ошибки, на проверку которых он не запрограммирован. Это могут быть неточности в позиционировании окон, ошибки в надписях, которые не проверяются, ошибки контролов и форм, с которыми не осуществляется взаимодействие во время выполнения скрипта. При ручном тестировании вероятность пропуска таких дефектов меньше;

5) **стоимость инструмента для автоматизации** – в случае, если используется лицензированное ПО, его стоимость может быть достаточно высока. Свободно распространяемые инструменты, как правило, имеют более скромный функционал и отличаются меньшим удобством работы. Кроме того, отдельную проблему представляет выбор инструмента автоматизации, от которого зависит успех дальнейшей работы [13].

Исходя из описанных преимуществ и недостатков автоматизированного тестирования, можно сделать вывод о том, что автоматизацию необходимо использовать совместно с ручным тестированием. В противном случае уровень качества программного продукта может быть снижен за счет наличия дефектов, пропущенных тестовыми скриптами. Также нельзя не отметить, что автоматизация, дополняющая ручное тестирование, способна повысить уровень качества программного продукта за счет обнаружения дефектов в труднодоступных для ручного тестирования областях.

5.2. Области автоматизации

Области автоматизации могут быть различными. Как правило, под автоматизацию попадают логгирование файлов, процессы, происходящие без участия пользователя, запись и редактирование данных в базе данных (БД), т. е. те операции, выполнение которых вручную либо затруднено, либо невозможно, либо не воссоздает реальных ситуаций использования системы.

Примеры:

- отдельные тестовые сценарии, проверяющие базовые операции создания/чтения/изменения/удаления сущностей (т. е. CRUD операции – Create / Read / Update / Delete). Например, создание, просмотр, изменение и удаление данных о какой-либо сущности системы;
- проверки работы с файлами, проверки интерфейсов и других областей, которые затруднительно тестировать вручную. Например, система создает xml-файл для отправки во внешнюю систему. Необходимо проверить корректность структуры и содержимого созданного файла.

Помимо **труднодоступных мест системы,** автоматизация зачастую затрагивает **часто используемую функциональность, риски от наличия ошибок в которой высоки.** Автоматизированная проверка критических мест в системе может гарантировать быстрое нахождение ошибок и их быстрое исправление.

Самое большое количество времени уходит на проверку всевозможных **валидационных ошибок и правил заполнения полей.** Здесь достаточно вспомнить количество комбинаций, возникающее при попытке перебрать различные наборы данных и граничные значения наборов. Все становится ещё сложнее, если поля формы зависят друг от друга. Данные операции легко автоматизируются и могут затем быть проверены без участия тестировщика.

Автоматизировать стараются и **длинные типовые сценарии,** которые выполняет система с пользовательским участием или без него, а также взаимодействие системы с внешними по отношению к ней системами.

Пример: пользователь заходит в интернет-магазин, просматривает товары, кладет выбранное в корзину, оформляет заказ, выходит из интернет-магазина. Это так называемый **end-to-end сценарий,** который проверяет совокупность действий. Такие сценарии возвращают систему в состояние, практически равное тому, с которого началось прохождение теста. Поэтому такие сценарии почти не оказывают воздействия на следующие тесты в наборе.

Помимо всего вышеперечисленного, автоматизация важна там, где тестировщик может что-то пропустить или сделать мелкую ошибку, ведущую к абсолютной некорректности выполняемой им проверки. Например, **сложные математические расчеты.** Один раз написанный и отлаженный скрипт может гарантировать более достоверную проверку.

Автоматизация затрагивает **правильность поиска информации,** а также может облегчать **создание тестовых данных в системе.** Например, генерацию тестовых пользователей системы, заполнение базы данных определёнными наборами значений и т. д. [13].

5.3. Уровни автоматизации

Если подойти к вопросу условно, то тестируемая система (приложение) может быть разбита на три уровня (рис. 17).

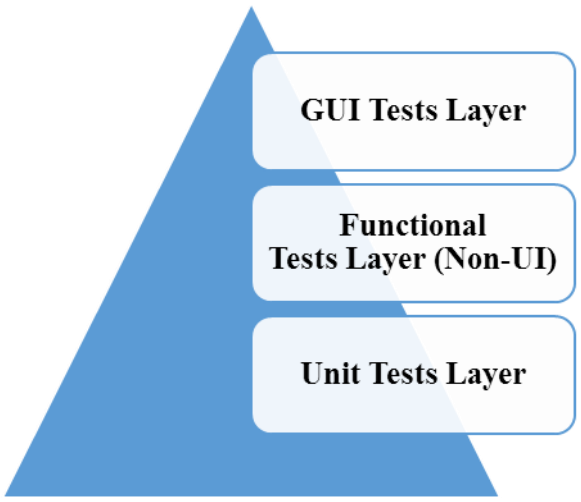

Рис. 17. Три уровня автоматизации

Рассмотрим каждый уровень более детально.

Уровень модульного тестирования (Unit Tests layer). Под автоматизированными тестами на этом уровне понимаются модульные или компонентные тесты, которые чаще всего пишут разработчики программного продукта. Такие тесты позволяют начать тестирование на ранней стадии разработки проекта. Более того, разработчики используют их для проверки исправлений дефектов. Количество тестов на уровне модульного тестирования увеличивается за счет разработки новых модулей и тестов для них [13].

Уровень функционального тестирования (Functional Tests Layer (non-UI)). Не всю бизнес-логику приложения можно протестировать через слой GUI. Иногда бизнес-логика скрыта от пользователя, и это является особенностью реализации. Но тестирование бизнес-логики все равно необходимо. В таком случае для команды тестирования реализуется доступ напрямую к функциональному слою, что обеспечивает возможность тестировать бизнес-логику приложения, минуя пользовательский интерфейс.

Уровень тестирования через пользовательский интерфейс (GUI Tests Layer). На данном уровне есть возможность тестировать не только интерфейс пользователя, но также и функциональность, выполняя операции, которые вызывают бизнес-логику приложения. Такие тесты, как правило, более эффективны, чем тестирование

функционального слоя напрямую, так как функциональность тестируется посредством эмуляции действий конечного пользователя через графический интерфейс.

Идеальной является ситуация, когда автоматизация тестирования выполнена на всех трех указанных выше уровнях [13].

5.4. Выбор инструмента для автоматизированного тестирования

Выбор инструмента зависит от объекта тестирования и требований к тестовым сценариям, так как инструменты тестирования не могут поддерживать абсолютно все технологии, используемые при разработке приложений. Зачастую выбор инструмента – это многочисленные пробы и ошибки. Оптимальность инструмента автоматизации определяется следующими ключевыми моментами:

1) **время, требуемое на поддержку скриптов.** Если небольшой скрипт, который осуществляет нажатие на определенную кнопку, придется полностью переписать, в случае простого изменения целевой кнопки, то инструмент, вероятнее всего, не будет удобен. Необходимо подбирать такой инструмент, который позволяет выносить переменные в начало скриптов и при необходимости просто заменять их значения. Еще более удобным вариантом является возможность описать наборы кнопок, меню и прочие подобные элементы как отдельные классы;

2) **способность распознавать элементы управления в разрабатываемом приложении.** Если большинство элементов не распознается с помощью выбранного инструмента, то необходимо искать специальные модули или плагины, которые обеспечат такую возможность. Если не удается отыскать что-либо подходящее, то придется отказаться от выбранного инструмента, так как он не обеспечит удобство написания автоматизированных тестов. Здесь хорошо работает правило: чем больше элементов управления способен распознать выбранный инструмент, тем меньше времени потребуется на создание и поддержание работоспособности автотестов;

3) **удобство написания новых тестов.** Очень важно знать, имеется ли у инструмента возможность поддержки объектно ориен-

тированного программирования (ООП), насколько удобно писать код, а также насколько он читаем. Важную роль играет удобство среды разработки и порог вхождения (насколько сложно новому специалисту по автоматизации освоить инструмент, если он никогда ранее с ним не работал).

Критериев выбора инструмента автоматизации существует гораздо больше. Здесь приведены лишь самые основные. При выборе следует также учитывать особенности разрабатываемой системы и распределение ролей в проектной команде.

Зачастую выбирают несколько инструментов для тестирования функций приложения. Существует большое количество всевозможных инструментов, поддерживающих различные языки и технологии разработки и позволяющих наиболее удобно проверять те или иные аспекты работы разрабатываемой системы.

Рассмотрим различные **инструменты для автоматизированного функционального тестирования** (табл. 9).

Таблица 9

Инструменты автоматизированного тестирования

Производитель	Инструмент
Hewlett-Packard (Mercury Interactive)	QuickTest Professional, WinRunner
IBM Rational	Rational Robot, Rational Functional Tester
Borland (Segue)	SilkTest
AutomatedQA Corp	TestComplete
Microsoft	Microsoft VS 2005
SeleniumHQ	Selenium

Наиболее часто в современной автоматизации тестирования используют **Java-библиотеки для автоматизации тестирования (java-based test tools and libraries).** Существует большое количество таких библиотек. В табл. 10 описаны наиболее известные из них.

Таблица 10

Java-based test tools and libraries

Инструмент	Описание
Selenium	Selenium представляет собой набор различных программных инструментов, каждый из которых использует свой подход к поддержке автоматизации тестирования web-приложений на многих платформах
Watij	Watij (произносится как «wattage») предназначается для тестирования web-приложений на Java. Это чистый Java API, созданный, чтобы обеспечить автоматизацию тестирования web-приложений с помощью реального браузера
HtmlUnit	HtmlUnit – это «браузер» для программ на Java. Он моделирует HTML-документы и предоставляет API, что позволяет ссылаться на страницы, заполнять формы, кликать на ссылки и т. д. Имеется достаточно хорошая поддержка JavaScript, которая постоянно улучшается. Также он способен работать даже с достаточно сложными AJAX-библиотеками, эмулировать Firefox или Internet Explorer в зависимости от используемой конфигурации. Данный инструмент обычно используется для тестирования или извлечения информации с web-сайтов. HtmlUnit не является инструментом unit-тестирования. Он лишь позволяет имитировать браузер и предназначен для использования с другими тестовыми фреймворками, например такими, как JUnit или TestNG
HttpUnit	Написан на Java и эмулирует определенные особенности поведения браузера, включая работу с формами, JavaScript, простую http-аутентификацию, файлы cookies и автоматический редирект на страницы. Позволяет рассматривать полученные страницы как текст, XML DOM, контейнеры форм, ссылок и таблиц. При использовании совместно с Junit довольно просто написать тесты, которые очень быстро проверят работоспособность web-сайта

Окончание табл. 10

Инструмент	Описание
Jameleon	Jameleon – это автоматизированная среда тестирования, которая может быть использована как техническими, так и нетехническими специалистами. Одна из основных особенностей – создание группы ключевых слов и тегов, которые представляют различные экраны приложения. Вся логика, которую необходимо автоматизировать для каждого конкретного экрана, может быть определена на Java и сопоставлена с этими ключевыми словами. Ключевые слова затем могут быть организованы вместе с разными наборами данных, чтобы сформировать тестовые скрипты без необходимости глубоко понимать, как работает приложение. Тест-скрипты затем используются для автоматизации тестирования и генерации тестовой документации
JUnit	JUnit – это простая основа для написания повторяемых тестов. Это пример архитектуры xUnit для модульного тестирования
Abbot	Abbot – это простой фреймворк для модульного и функционального тестирования графических пользовательских интерфейсов на Java. Он облегчает генерацию действий пользователя и исследование состояний компонентов. Имеется поддержка записи и воспроизведения для любого Java-приложения
Marathon	Marathon позволяет захватывать взаимодействия пользователя с приложениями, а также добавлять проверки, что выполняется корректное действие. Сгенерированный скрипт может быть подвержен рефакторингу (изменение с целью оптимизации) [13]

Кроме представленных Java-фреймворков, существует огромное количество фреймворков и инструментов, ориентированных на другие языки программирования, такие как: php, ruby, javascript, C#, javascript, python, perl и др.

5.5. Архитектура тестов

Структура тестов (как автоматизированных, так и выполняемых вручную) в общем случае идентична. Это необходимо для удобства совместного использования автоматизированных и ручных тестов. **Любой тест включает в себя три основных блока:**

- preconditions;
- steps;
- post-conditions.

Схематически данная структура изображена на рис. 18.

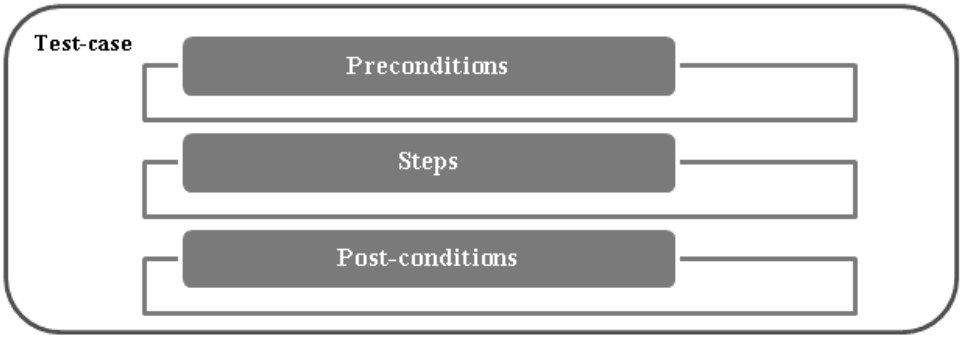

Рис. 18. Общая структура теста

В блоке **«Preconditions»** описываются начальные условия, которые должны быть выполнены, чтобы можно было пошагово проходить тест-кейс. Для ручного теста это может быть, например, авторизация в приложении и открытие определенной формы, на которой далее по шагам будут производиться какие-либо действия, проверяющие конкретную функциональность. Кроме того, в качестве предусловия может выступать описание пользователя (тестового клиента) системы, для которого должны быть выполнены шаги. Например, определенные формы приложения доступны только юридическому лицу. Тогда в предусловии к тесту может быть сказано следующее: «Юридическое лицо авторизовано в системе». Для автотеста предусловия в целом такие же, за исключением того, что тест-скрипт при выполнении самостоятельно подготавливает систему к дальнейшему выполнению шагов теста и инициализирует тестовые данные. Если упустить предусловия, то тест может быть выполнен некорректно, или тестировщик, выполняя тест, упустит важные условия и дефекты останутся не обнаруженными.

Блок **«Steps»** предполагает описание шагов, по которым будет непосредственно проводиться тест. Каждый шаг ручного теста должен содержать лишь одно действие и корректное описание ожидаемого для данного действия результата. Для автоматизированного скрипта действует то же правило. Несколько действий в рамках одного шага могут привести систему в непригодное для выполнения следующего шага состояние, а следовательно, к некорректности выполнения теста и пропуску дефектов. Данный блок также предусматривает сохранение результатов прохождения теста. Это может быть **провал (если хотя бы на одном шаге фактическое поведение системы отличалось от описанного ожидаемого результата) или успешное прохождение.** Кроме того, выполнение теста может быть **блокировано** имеющимся в системе дефектом. Четко фиксируются как сами результаты, так и шаги, на которых тест был провален. Рекомендуется, чтобы все шаги теста относились к единственной проверке. Большое количество разнотипных проверок в рамках одного теста является некорректным и также может привести к пропуску имеющихся в системе дефектов. **Кроме того, нельзя совмещать негативные и позитивные проверки в рамках одного теста** [13].

Блок **«Post-Conditions»** включает в себя удаление ненужных тестовых данных, подготовку системы в состояние, пригодное для выполнения следующего теста, а также корректное завершение работы системы. Этот блок очень важен для автотеста. В ручном тесте данный блок может просто подразумеваться, но не описываться.

Помимо трех самых важных разделов тест может содержать в себе и краткое описание осуществляемых проверок. Данный блок называется **«Summary».** Здесь указываются основные требования и цели выполнения проверок.

Для автотестов создается также **библиотека по обработке исключений и ошибок,** например:

- PreConditionException;
- TestCaseException;
- PostConditionException.

Очень важно поддерживать как ручные, так и автоматизированные тесты в актуальном состоянии [13].

Пример того, как выглядят шаги теста, представлен в табл. 11.

Таблица 11

Пример теста

Summary: проверка выбора телефонного номера из адресной книги мобильного устройства		
Preconditions: пользователь авторизован в системе, открыта форма перевода по номеру мобильного телефона		
Номер шага	**Действия**	**Ожидаемый результат**
1	Нажать на кнопку выбора номера телефона из адресной книги	Открыта адресная книга устройства
2	Выбрать номер телефона нажатием на него	Адресная книга закрыта. Выбранный номер подставлен в поле «Номер телефона» на форме перевода по номеру мобильного телефона

Контрольные вопросы

1. Что такое автоматизированное тестирование и чем оно отличается от ручного?
2. Что означает понятие Software Automation Tester?
3. Что такое тест-скрипт?
4. Для чего тест-скрипты объединяют в наборы?
5. Как определить целесообразность автоматизации тестирования?
6. Почему повторяемость относится как к преимуществам, так и к недостаткам автоматизированного тестирования?
7. Какие цели автоматизации тестирования вы можете назвать?
8. Что лучше всего автоматизировать?
9. Приведите пример end-to-end сценария.
10. Применимы ли уже известные вам методики проектирования тестов к автоматизированным тестам?

11. Почему регрессионное тестирование наиболее часто автоматизируют?

12. Какова архитектура теста?

13. Почему архитектура ручного и автоматизированного теста в целом идентична?

14. Перечислите уровни автоматизации и охарактеризуйте каждый из них.

15. Какие из инструментов, используемых для автоматизации тестирования, вам известны?

16. Что необходимо учитывать при выборе инструмента автоматизации тестирования?

17. Важны ли особенности разрабатываемой системы при выборе инструмента автоматизации тестирования?

18. Можно ли обойтись только автоматизированным тестированием?

19. Приведите пример предусловия к тесту.

20. Приведите пример шагов теста.

21. Почему важно привести систему к пригодному для тестирования состоянию как до, так и после выполнения теста?

22. Какую дополнительную библиотеку обычно создают для автотестов?

23. Можно ли совмещать негативные и позитивные проверки в рамках одного теста?

24. Почему один тест должен подразумевать одну проверку?

25. Почему один шаг теста не должен содержать несколько действий?

6. УПРАВЛЕНИЕ ТЕСТИРОВАНИЕМ

Данный раздел посвящен основам управления тестированием и его наиболее важным аспектам. Здесь будут рассмотрены основные инструменты, помогающие поддерживать процесс тестирования и эффективно им управлять.

6.1. Лица, заинтересованные в тестировании

В проектной команде можно выделить несколько групп лиц, которые прямо или косвенно заинтересованы в тестировании программного продукта.

К таким группам относятся:

- лица, задействованные в тестировании прямо или косвенно;
- лица, использующие результаты тестирования;
- лица, заинтересованные в качестве конечного продукта.

В зависимости от специфики проекта в каждую из перечисленных групп могут входить различные люди. Например, **тестировщики и тест-менеджеры**, которые непосредственно тестируют и управляют процессом тестирования программного продукта соответственно. **Архитекторы и разработчики**, которые получают результаты тестирования и на их основании принимают всевозможные решения. **Менеджеры проекта, бизнес-аналитики, службы поддержки,** которые на основании результатов тестирования продукта также принимают решения, дополняют или корректируют требования, ставят новые задачи, управляют инцидентами (проблемами, возникающими при использовании системы конечными пользователями). Кроме того, заинтересованными в тестировании могут быть и **государственные структуры**, если они так или иначе регламентируют работу программного продукта. Нельзя не отметить, что **конечные пользователи системы** также являются заинтересованными в тестировании лицами, так как они желают использовать качественный программный продукт.

Определение заинтересованных лиц – самый первый шаг при планировании тестирования. Необходимо не просто выявить группы заинтересованных лиц, **но и определить их ожидания и цели относительно тестирования.** Зачастую такая информация систематизируется в виде **списков или реестров заинтересованных лиц.** Это позволяет координировать коммуникации с каждой из групп заинтересованных лиц.

Для того чтобы сделать процесс тестирования открытым для всех заинтересованных лиц, необходимо разработать метрики, кото-

рые покрывают цели, преследуемые всеми заинтересованными лицами. Реестры заинтересованных лиц позволяют понять, кого нужно держать в курсе о результатах процесса тестирования, с кем нужно сотрудничать, чьи интересы необходимо учитывать при планировании тестирования, чьи цели наиболее приоритетны, кого необходимо информировать о возникающих проблемах и т. д.

Рассмотрим две основные позиции в тестировании:

- руководитель тестирования (тест-менеджер);
- тестировщик.

Роль руководителя тестирования может исполнять менеджер проекта, менеджер разработки, менеджер по качеству или управляющий группой тестирования.

В задачи тест-менеджера, как правило, входит:

- составление и анализ стратегии тестирования для конкретного проекта;
- координирование планов и стратегий тестирования с менеджерами проектов и другими людьми;
- планирование тестов, выбор методов тестирования, оценка времени, трудозатрат, стоимости тестирования и наличия ресурсов, определение уровней и циклов тестирования, планирование управления инцидентами;
- подготовка, создание и исполнение тестов, отслеживание и контроль результатов тестирования, проверка критерия выхода;
- принятие решения об автоматизации тестирования;
- выбор средств поддержки тестирования и организация обучения тестировщиков;
- формирование отчетов по тестированию на основе данных, полученных при проведении тестирования.

Задачи тестировщика:

- проверка и дополнение планов тестирования;
- анализ, рецензирование и оценка тестируемости пользовательских требований, спецификаций и моделей;
- подготовка тестовых данных и настройка тестового окружения;

- разработка тестов, их выполнение, систематизация результатов, документирование результатов и оформление дефектов;
- автоматизация тестов;
- актуализация тестов по мере необходимости [14].

6.2. Планирование тестирования

В общем случае планирование тестирования включает следующие основные действия:

- создание тест-плана;
- продумывание стратегии тестирования;
- оценка трудозатрат и рисков;
- прогнозирование сроков и составление графика проведения тестирования;
- определение используемых инструментов.

Рассмотрим каждую из активностей более подробно.

Тест-план – это основной документ в тестировании, относящийся к проектной документации и описывающий весь объём работ по тестированию. Тест-план создается с целью согласования активностей процесса тестирования со всеми заинтересованными лицами, приоритизации задач по тестированию, учета требуемых ресурсов, возможных рисков и своевременного планирования трудовых и материальных затрат на тестирование.

Тест-план может иметь различную форму представления. Он может быть оформлен в виде документа или схемы. **Любой тест-план должен содержать следующую информацию:**

- объект тестирования и его особенности;
- функции и компоненты тестируемой системы;
- стратегия тестирования (виды тестирования, применяемые к тестируемому объекту);
- тестовые среды;
- активности по тестированию, их порядок и длительность;
- риски и способы их уменьшения.

На рис. 19 представлен один из типовых схематичных шаблонов тест-плана.

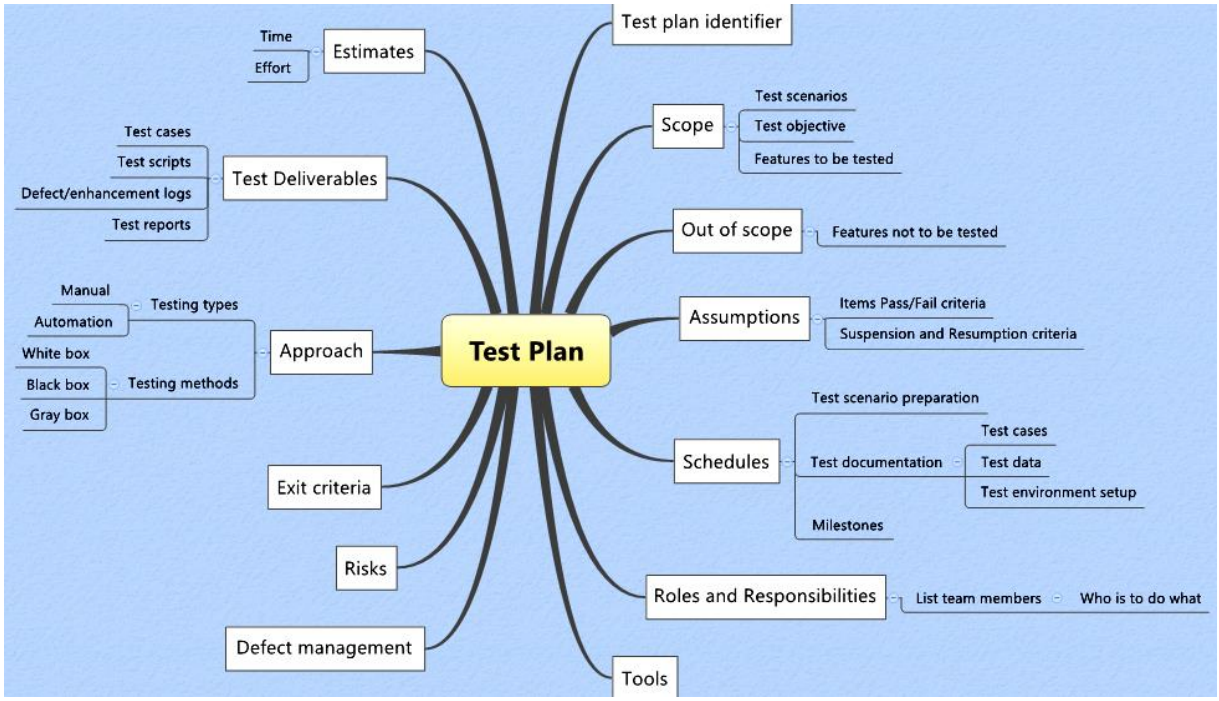

Рис. 19. Шаблон тест-плана

При планировании тестирования важно определять критерий входа и критерий выхода. **Критерий входа** показывает, когда нужно начинать тестирование.

Обычно критерии входа покрывают:

- готовность и доступность тестового окружения;
- готовность средства тестирования в окружении;
- доступность тестируемого кода;
- доступность тестовых данных.

Критерий выхода определяет, когда нужно прекращать тестирование, например по окончании уровня тестирования или когда набор тестов достиг определенной цели.

Обычно критерии выхода покрывают:

- тщательность оценки, например покрытие кода, функциональности или рисков;
- оценку плотности дефектов или измерение надежности;
- стоимость;
- остаточные риски, такие как неисправленные дефекты или недостаток тестового покрытия какой-либо области;
- план, основанный на времени выхода ПО на рынок.

Стратегия тестирования может быть частью общего тест-плана или отдельным документом. Стратегия дополняет тест-план и содержит информацию об инструментах и техниках тестирования, типах тестов для каждого компонента системы и его функций, настройках конфигурации. Стратегия тестирования, как и тест-план, может быть представлена в виде словесного описания или в более наглядных формах (таблицы, схемы).

Неотъемлемой частью планирования тестирования является стратегия управления рисками. Информация о возможных рисках используется при планировании и расстановке приоритетов тестирования. Необходимо определить наиболее уязвимые области системы и сконцентрировать на них наибольшие усилия по тестированию.

Для определения рисков, связанных с системой (риски продукта), при планировании тестирования необходимо ответить на ряд вопросов:

- Что является наиболее важным функционалом системы? Следует хорошо понимать предметную область и важность той или иной функциональности для конечного пользователя.
- Если имеется проблема в какой-либо части функционала, то сильно ли она повлияет на потенциальных пользователей, будет ли заметна заказчику и повлечет ли материальные потери? Важно также понимать уровень материальных потерь.
- Как часто используется тот или иной модуль или функция системы и кем? Для ответа на данный вопрос требуется знание целевой аудитории, для которой создается система [14].

Существует большое количество рисков, которые непосредственно **не связаны с создаваемым программным продуктом,** но могут оказать значительное влияние на его качество (так называемые **проектные риски**). В качестве примера можно привести такие факторы, как нехватка персонала, низкая квалификация сотрудников, нескоординированные коммуникации с заказчиком, недостаток тестовых инструментов, тестовых устройств, слабое понимание специалистами предметной области, нескоординированное руководство, ненадлежащее планирование и т. д.

Помимо оценки рисков, требуется оценить трудозатраты в тестировании. Зачастую выделяют **два подхода к оценке трудозатрат:**

- **основанный на метриках:** оценка трудозатрат основана на метриках предыдущих или сходных проектов или на типичных значениях;
- **основанный на экспертной оценке:** оценка задач производится владельцем этих задач или экспертом, например тестировщиком, который будет выполнять задачу и уже имеет опыт работы с подобными задачами [15].

Как только оценка трудозатрат выполнена, определяют ресурсы и составляют график тестирования. На этом этапе фиксируются сроки тестирования, которые при успешном раскладе не должны быть превышены. Работы по тестированию могут зависеть от многих факторов, включая:

- **характеристики продукта:** качество спецификаций или другой информации, используемых моделей тестирования (т. е. основы тестирования), размер продукта, сложность предметной области, требования к надежности и безопасности и требования к документации;
- **характеристики процесса разработки:** стабильность организации, используемые средства, процессы тестирования, квалификация вовлеченных людей и временные ограничения;
- **результат тестирования:** количество дефектов и объем работы, которую необходимо переделать.

В рамках планирования тестирования и проекта в целом также выбирают используемые инструменты, о которых речь будет идти далее.

6.3. Мониторинг и контроль

Мониторинг процесса тестирования – это постоянное наблюдение и отслеживание процесса, его активностей и результатов в определенных временных рамках. Мониторинг необходим для получения результата и обзора процесса тестирования. Отслеживание может быть проведено вручную или автоматически при помощи специальных инструментов. На основе полученной информации могут быть измерены критерии выхода, например покрытие, плотность дефектов и т. д. [2].

Для оценки результатов мониторинга используют всевозможные **метрики прогресса тестирования** по сравнению с запланированным графиком и выделенным бюджетом.

В качестве **примеров тестовых метрик можно привести:**

- процент работ по подготовке тестовых сред;
- процентное соотношение запланированных и подготовленных тестовых сценариев;
- количество выполненных и невыполненных тестовых сценариев;
- процент пройденных успешно и проваленных тестовых сценариев;
- процент заблокированных тестовых сценариев;
- процент неактуальных тестовых сценариев;
- количество найденных и исправленных дефектов;
- плотность дефектов;
- количество дефектов по критичности;
- количество переоткрытых в ходе повторного тестирования дефектов;
- интенсивность отказов;
- тестовое покрытие требований, рисков или кода;
- стоимость тестирования и др.

В рамках мониторинга тестирования наиболее часто используют так называемую **матрицу покрытия (трассировки)**. Это специальный документ, который позволяет более структурированно отобразить требования к продукту, отследить степень покрытия требований тестами и дает возможность поддержать тесты в более актуальном состоянии, если в требования вносятся изменения.

Матрицу трассировки составляют в несколько этапов:

1) составление нумерованного списка требований к приложению;
2) составление нумерованного списка тестов;
3) непосредственное составление матрицы трассировки [14].

Рассмотрим процесс составления матрицы трассировки на примере нескольких требований к выбору номера мобильного телефона из телефонной книги при выполнении перевода по номеру мобильного телефона в мобильном банке.

Составим нумерованный список требований (табл. 12).

Таблица 12

Пример нумерованного списка требований к функционалу системы

Номер требования	Требование
Требование 1	При нажатии на иконку в виде телефонной книги на форме перевода по номеру мобильного телефона необходимо открывать телефонную книгу на мобильном устройстве
Требование 2	При нажатии на номер телефона в телефонной книге телефонная книга должна закрываться. Должна отображаться форма перевода по номеру мобильного телефона
Требование 3	При выборе номера нажатием на него в телефонной книге он должен подставляться в поле «Номер телефона» на форме перевода по номеру мобильного телефона
Требование 4	При подстановке номера телефона в поле «Номер телефона» номер должен форматироваться по правилу: удаляются все лишние символы и в поле подставляются только 10 последних цифр выбранного номера

Далее составим нумерованный список тестов, которые написаны для покрытия перечисленных требований (табл. 13).

Таблица 13

Пример нумерованного списка тестов

Номер теста	Проверка
Тест 1	Открытие адресной книги с формы перевода по номеру мобильного
Тест 2	Выбор номера телефона из адресной книги
Тест 3	Подстановка номера телефона в поле «Номер телефона»
Тест 4	Подстановка номера, состоящего более чем из 10 цифр
Тест 5	Подстановка некорректно сохраненного номера, содержащего символы, отличные от цифр
Тест 6	Подстановка номера, содержащего менее 10 цифр

Наконец составим матрицу трассировки на основании табл. 12, 13. Результат представлен в табл. 14.

Таблица 14

Пример матрицы трассировки

Номер требования/теста	Тест 1	Тест 2	Тест 3	Тест 4	Тест 5	Тест 6	Количество тестов, покрывающих требование
Требование 1	X						1
Требование 2		X	X	X	X	X	5
Требование 3			X	X	X	X	4
Требование 4				X	X	X	3

Мониторинг тестирования дает информацию для контроля процесса тестирования в рамках проекта. **Контроль тестирования** в отличие от мониторинга описывает любые направляющие или корректирующие действия, принятые как результат по полученной и собранной информации и значениям метрик. Контроль в общем случае затрагивает любые действия по тестированию и, кроме того, может воздействовать на любые задачи жизненного цикла программного обеспечения.

В рамках контроля тестирования могут приниматься решения на основании данных мониторинга и значений метрик. Далее на базе принятых решений могут расставляться дополнительные приоритеты или корректироваться ранее поставленные. Кроме того, дополнительно оцениваются риски продукта и внешние, не связанные с продуктом, риски. Может быть изменено расписание тестирования и откорректирован тест-план.

В ходе тестирования и по его завершении обязательным является составление **отчётности по тестированию**. Формы отчетности могут быть самыми разными и, как правило, зависят от потребностей и специфики проекта. Отчеты могут предоставить заинтересованным лицам следующие сведения:

- сроки достижения критериев выхода;
- результаты анализа метрик тестирования для принятия дальнейших решений.

На основании таких результатов оцениваются оставшиеся дефекты, принимается решение о продолжении тестирования или его завершении, а также оцениваются возможные риски и составляется общая картина качества тестируемого ПО.

Завершение процесса тестирования также включает в себя несколько видов активности.

Во-первых, необходимо проверить завершение выполнения всех запланированных тестов. Далее нужно убедиться, что все известные дефекты исправлены, а неисправленные отложены на будущий релиз. Важно, что при откладывании дефекта на будущий релиз необходимо четко понимать степень критичности дефекта и оценивать его влияние на потенциальных пользователей системы. Не исправленный вовремя критический дефект может привести к материальным потерям пользователей, а значит, и к потерям компании-разработчика.

Во-вторых, все артефакты тестирования (тесты, не исправленные по разным причинам дефекты, тестовые наборы автоматического и ручного тестирования и т. д.) **должны быть задокументированы и переданы заинтересованным лицам.**

В-третьих, все полученные знания и опыт, а также ошибки, проблемы и способы их разрешения должны быть учтены, систематизированы, **зафиксированы в виде полезных статей/документов. Это позволит избежать трудностей в дальнейшей работе** [14].

6.4. Управление тестами

В процессе тестирования количество тестовых сценариев постоянно растет, формируются различные тестовые наборы, тесты дописываются, актуализируются, многократно выполняются. Поэтому возникает потребность в инструменте, который позволил бы грамотно и быстро управлять всеми имеющимися в проекте тестами. В настоящее время рынок программного обеспечения предлагает несколько подобных инструментов, суть работы которых в целом сходна. В табл. 15 представлены названия и краткие описания функциональности инструментов управления тестами. Каждый из инструментов имеет свои преимущества и недостатки. Выбор инструмента управления тестами, как и инструмента автоматизации, зависит от специфики и потребностей самого проекта.

Таблица 15

Инструменты управления тестами

Инструмент	Описание
Sitechco	Веб-версия бесплатна и доступна на русском и английском языках. Система позволяет создавать, редактировать и хранить чек-листы (листы проверок, в которых, в отличие от тестов, отсутствует подробное описание шагов и предусловий). Результаты прохождения проверок могут документироваться, а также есть возможность формировать отчеты **Подробнее об инструменте:** http://sitechco.ru/
TestLink	Инструмент с открытым исходным кодом, позволяющий создавать и поддерживать несколько проектов. Внутри каждого проекта могут создаваться тесты, тестовые наборы и тест-планы. Результаты выполнения тестов можно сохранять, благодаря чему легко формировать различную отчетность, которая может быть разослана заинтересованным лицам. Каждый пользователь в системе играет свою роль, что дает возможность разделять функции обычных тестировщиков и тест-менеджеров. Интеграция с системами управления дефектами **Подробнее об инструменте:** http://testlink.org/
TestRail	Данный инструмент является платным. Бесплатен лишь доступ к системе в течение 30 дней. По аналогии с остальными инструментами здесь есть возможность создавать и импортировать тесты, группировать их в тестовые наборы, создавать тест-планы, строить отчёты **Подробнее об инструменте:** http://www.gurock.com/testrail/
HP Quality Center	Платный инструмент для управления процессом контроля качества на различных этапах разработки ПО. Продукт включает в себя несколько модулей: Requirements management, Management, Test Plan, Test Lab, Defects management. Лицензия может быть приобретена как для отдельных модулей, так и для всей системы в целом в зависимости от потребностей проекта **Подробнее об инструменте:** http://www8.hp.com/ru/ru/software-solutions/quality-center-quality-management/
Rational Quality Manager	Платный инструмент, который подходит как для управления тестированием и тестами, так и для управления проектом в целом. Данный инструмент также хорошо подходит для управления требованиями. Кроме того, есть возможность связывать между собой различные браузеры, базы данных, операционные системы, т. е. создавать различные тестовые конфигурации. Имеется бесплатная пробная версия на 90 дней [14] **Подробнее об инструменте:** http://www-03.ibm.com/software/products/ru/ratiqualmana

6.5. Управление дефектами

Дефект (баг или bug, от англ. «жук») – ошибка в программе или системе, которая приводит к неожиданному и некорректному поведению и, как следствие, к некорректному ожидаемому результату.

Помимо необходимости управлять тестами в рамках процесса тестирования также существует необходимость управления дефектами. Для того чтобы понять, с чего начинается дефект, рассмотрим его **жизненный цикл** (рис. 20).

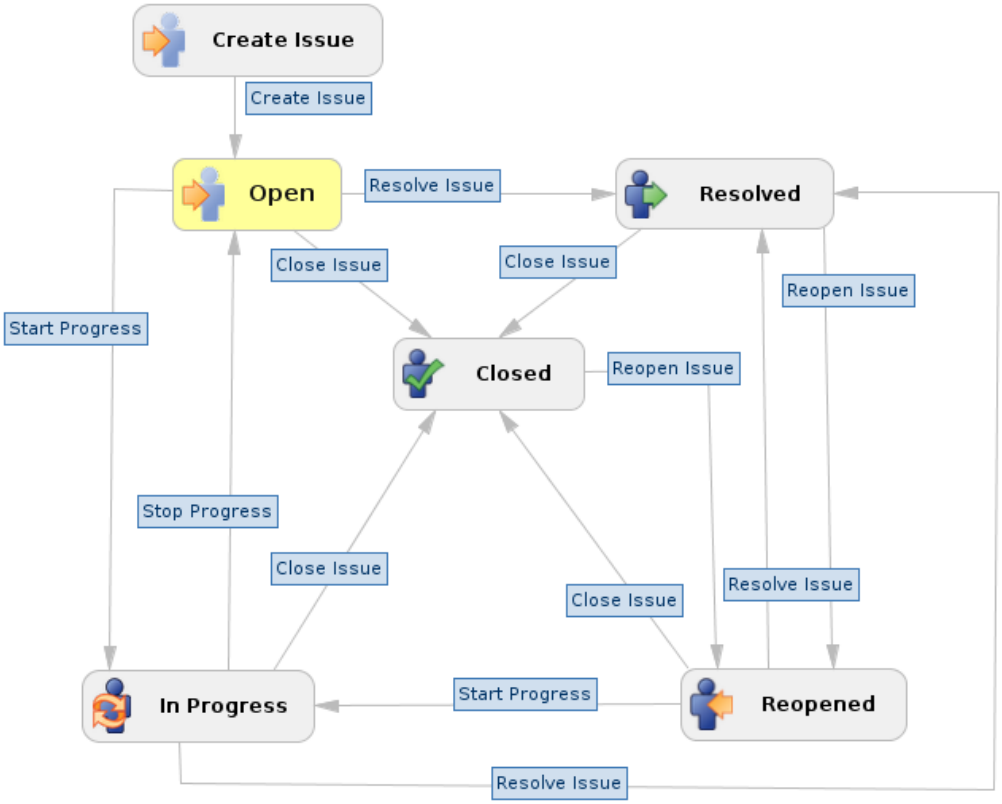

Рис. 20. Пример жизненного цикла дефекта

Жизненный цикл дефекта (bug workflow) – это последовательность этапов, которые проходит дефект с момента его создания до момента закрытия и завершения работ по нему. Зачастую жизненный цикл зависит от выбранной системы управления дефектами и специфики проекта.

Наиболее обобщенный жизненный цикл дефекта включает в себя следующие состояния:

- **«Открыт» (Open)** – статус присваивается автоматически после внесения баг-репорта, т. е. создания дефекта в системе управления дефектами;

- **«Исправлен» (Resolved / Fixed)** – данный статус присваивается специалистом разработки после того, как ошибка была устранена;
- **«Открыт повторно» (Reopened)** – данный статус присваивается тестировщиком при повторном возникновении ошибки после её предварительного исправления;
- **«Закрыт» (Closed)** – дефект получает данный статус после проведения проверки, которая выявила окончательное исправление.

В **жизненный цикл дефекта могут быть добавлены дополнительные статусы,** например **«In Progress»,** чтобы показать, что в данный момент дефект исправляется, или **«Customer verify»** – для отражения того, что дефект был проверен заказчиком и действительно исправлен. Также может присутствовать статус **«Отклонён» (Need More Info / Resolved как Invalid, Won't fix, Can't reproduce или Not a bug)** – ошибка была проанализирована разработчиком или другим участником процесса разработки и по результатам анализа выявлена недостаточность описания или дефект является дубликатом, не будет исправлен по согласованию заинтересованных сторон, не может быть воспроизведён или заведен по ошибке и не является дефектом согласно требованиям к системе.

Как видно из рис. 20, **в жизненный цикл дефекта могут быть добавлены дополнительные статусы,** например: **«In Progress»,** чтобы показать, что в данный момент дефект исправляется, а также **«Customer verify»** – для отражения того, что дефект был проверен заказчиком и действительно исправлен.

Для того чтобы дефект появился в системе управления ошибками, его необходимо правильно создать. Дефект – это такой же документ. Иными словами создание дефекта часто называют **баг-репортом.** Баг-репорт включает в себя информацию о последовательности действий, которая привела к некорректной работе тестируемой системы, описание самого некорректного поведения, а также подробное указание на ожидаемый корректный результат.

Обычно в баг-репорте или описании дефекта содержится следующая информация:

- **название проекта** – зачастую выставляется автоматически системой управления ошибками;
- **наименование найденной ошибки** – должно быть лаконичным и содержательным, чтобы по названию ошибки можно было понять ее суть в целом, не открывая подробное описание дефекта;

- **информация о тестировщике, создавшем баг-репорт**, – как правило, автоматически подставляется при создании дефекта;
- **информация о лице, на которого дефект назначается для исправления**, – выбирается при создании дефекта в системе управления ошибками;
- **severity (серьезность)** – степень негативного влияния дефекта на продукт, которая выставляется автором дефекта. Градация серьезности дефектов представлена в табл. 16;
- **priority (приоритет)** – порядок исправления дефекта, который может быть выставлен как тестировщиком при создании дефекта, так и откорректирован в соответствии с потребностями бизнеса другими заинтересованными лицами. Градация приоритетов дефекта описана в табл. 17;
- **версия ПО, в которой была найдена ошибка**, – является обязательной к указанию и необходима для отслеживания исправленных и неисправленных дефектов в той или иной версии программного продукта;
- **версия ПО, в которой планируется исправить найденный баг**, – зачастую это просто следующая версия, но иногда может быть и следующий релиз;
- **окружение, при котором проводилась проверка**, – настройки тестовой среды, данные тестового клиента;
- **шаги по воспроизведению ошибки** – должны быть сформулированы так, чтобы по ним можно было без затруднений воспроизвести ошибку. Слишком краткое и слишком подробное описание зачастую приводят к трудностям воспроизведения. Не нужно описывать лишние детали, которые никак не влияют на результат. Перед тем как создать дефект, рекомендуется проанализировать ошибку, а после формулировки шагов попытаться самостоятельно воспроизвести дефект по описанному. Если возникли затруднения, то шаги, скорее всего, описаны неверно и у разработчика возникнут трудности с воспроизведением и исправлением;
- **фактический результат** – описание некорректной работы программного продукта;
- **ожидаемый результат** – корректное поведение системы согласно требованиям. Можно оставлять номера документов/ссылки на требования;

- **приложения** – скриншоты работы системы, лог-файлы ошибок, видео и другая дополнительная информация, которая позволит разработчику понять причину, воспроизвести и быстро поправить дефект;
- **комментарии и дополнения** – вносятся в ходе последующей работы с ошибкой различными заинтересованными лицами.

Таблица 16

Градация серьезности дефектов

Степень серьезности	Описание
Блокирующая (Blocker)	• Дефект приводит к невозможности завершить выполнение бизнес-процесса; • дефект приводит к непреднамеренному завершению работы системы, либо к невозможности запустить систему; • производительность системы не позволяет выполнять базовые бизнес-процессы
Критическая (Critical)	• Дефект приводит к невозможности завершить выполнение бизнес-процесса, но возможно завершить этот процесс обходным путем; • система не учитывает ограничения в доступе или другие настройки безопасности; • дефект приводит к некорректным финансовым вычислениям или к потере данных в БД
Значительная (Major)	• Часто встречающийся дефект, который не ведёт к потере данных в БД; • система формирует некорректные сообщения об ошибке, либо не формирует когда это необходимо
Незначительная (Minor)	Дефекты пользовательского интерфейса, которые не влияют на функционирование системы (грамматические ошибки, лишние полосы прокрутки, обновления экрана и т. д.)
Тривиальная (Trivial)	Абсолютно незначительный дефект, например лишние пробелы в тексте, практически незаметные дефекты в дизайне и т. п.

Таблица 17

Градация приоритетов дефектов

Приоритет	Описание
Высокий (High)	Требуется срочное исправление
Средний (Medium)	Исправление важно, но не является срочным
Низкий (Low)	Исправление может быть отложено на достаточно длительный срок

Приведем пример того, как могут соотноситься серьезность и приоритет дефекта. Предположим, что на главной странице веб-сайта в заголовке имеется грамматическая ошибка. Какова серьезность такого дефекта? Согласно табл. 16 это незначительный дефект, так как он не влияет на функционирование системы никаким образом. Кроме того, исправление такого дефекта – это дело нескольких минут, как и последующая проверка. Теперь необходимо выставить приоритет данному дефекту. Обратив внимание на то, что грамматическая ошибка присутствует на главной странице непосредственно в заголовке веб-сайта, мы будем вынуждены поставить высокий приоритет. Таким образом, дефект не является критичным для работы пользователя в системе, но требует быстрого исправления [15].

В результате работы тестировщиков по поиску дефектов ежедневно может заводиться, переоткрываться, закрываться по нескольку десятков дефектов. Поэтому, как и в случае с тестами, требуется удобная система, которая позволит управлять большим количеством баг-репортов и их переходами по статусам жизненного цикла.

Для таких целей предусмотрены так называемые **системы багтрекинга (bug tracking system, или BTS). Такие инструменты выполняют, как правило, следующие основные функции:**

- документирование дефектов и инцидентов, а также вопросов и предложений по улучшению программного продукта;
- отслеживание хода работ по дефектам и инцидентам;
- сбор статистики.

Кроме того, с помощью систем багтрекинга выполняются и такие действия, как:

- фиксация времени, потраченного на исправление дефекта;

- оформление не только дефектов, но и задач внутри проекта;
- составление отчетности по проекту и т. д.

Описание различных систем управления дефектами представлено в табл. 18.

Таблица 18

Системы багтрекинга

Bug tracking system	Описание
Atlassian JIRA	Гибко настраиваемая платная система управления проектами. Имеется большое количество всевозможных плагинов, которые позволяют адаптировать JIRA под нужды конкретного проекта. Есть возможность делать запросы к системе различными способами, сохранять запросы в качестве персональных фильтров. При определенной настройке JIRA способна рассылать уведомления о создаваемых дефектах/задачах и других действиях участников проектной команды по почте. Помимо всего прочего имеется приложение для iPhone, позволяющее в удаленном режиме отслеживать статус работ по проекту. JIRA может быть интегрирована с системами контроля версий **Подробнее:** https://www.atlassian.com/software/jira
Bugzilla	Является предшественником JIRA и свободно распространяется. Имеет доступ через веб-интерфейс. Имеется возможность импортировать список багов, отслеживать изменения по дефектам, создавать отчеты и новые дефекты с выбором нужных полей. Также имеется интеграция с системами управления версиями **Подробнее:** http://www.bugzilla.org/
Redmine	Открытое серверное веб-приложение для отслеживания ошибок. С помощью данного инструмента можно управлять не только дефектами, но и проектом в целом, создавая задачи внутри проекта. Как и в других подобных системах, имеется возможность настройки почтовых уведомлений **Подробнее:** http://www.redmine.org/

Окончание табл. 18

Bug tracking system	Описание
Mantis	Практически чистая BTS, которая распространяется по специальной лицензии. Для работы системы требуется web-сервер. Исходный код системы открыт. Имеется функциональность базового управления проектами без ограничения на их количество и дефектами. Документация к системе довольно сложна, но настроить под себя систему все-таки можно. Данная система написана на PHP и является кроссплатформенной **Подробнее:** https://www.mantisbt.org/
Microsoft Test Manager	Система является частью комплексного продукта Team Foundation Server (TFS). Главное преимущество – наличие связки «задача – дефекты – затраченное время». Имеется возможность записи прохождения тестов и автосбора информации по дефектам, что отличает данную систему от других подобных систем **Подробнее:** https://msdn.microsoft.com/ru-ru/library/jj635157.aspx
Trac	Система написана на Python и является кроссплатформенной. Исходный код открыт. Система имеет возможность работать с различными базами данных. Изначально в систему включен только базовый функционал, но можно настроить ее под себя при помощи плагинов **Подробнее:** https://trac.edgewall.org/
BugNet	Бесплатная система для внесения и последующего отслеживания ошибок с открытым исходным кодом. Имеется возможность настройки почтовых уведомлений и экспорта списков оформленных дефектов, а также функции составления отчётности **Подробнее:** http://www.bugnetproject.com/
Phabricator: Maniphest	Бесплатная система отслеживания ошибок, которая является частью платформы Phabricator, предназначенной для разработки ПО. Поддерживается работа через web-интерфейс. Задачи можно создавать через e-mail или URL [14] **Подробнее:** http://phabricator.org/

На рис. 21 представлена форма создания дефекта в системе Atlassian JIRA.

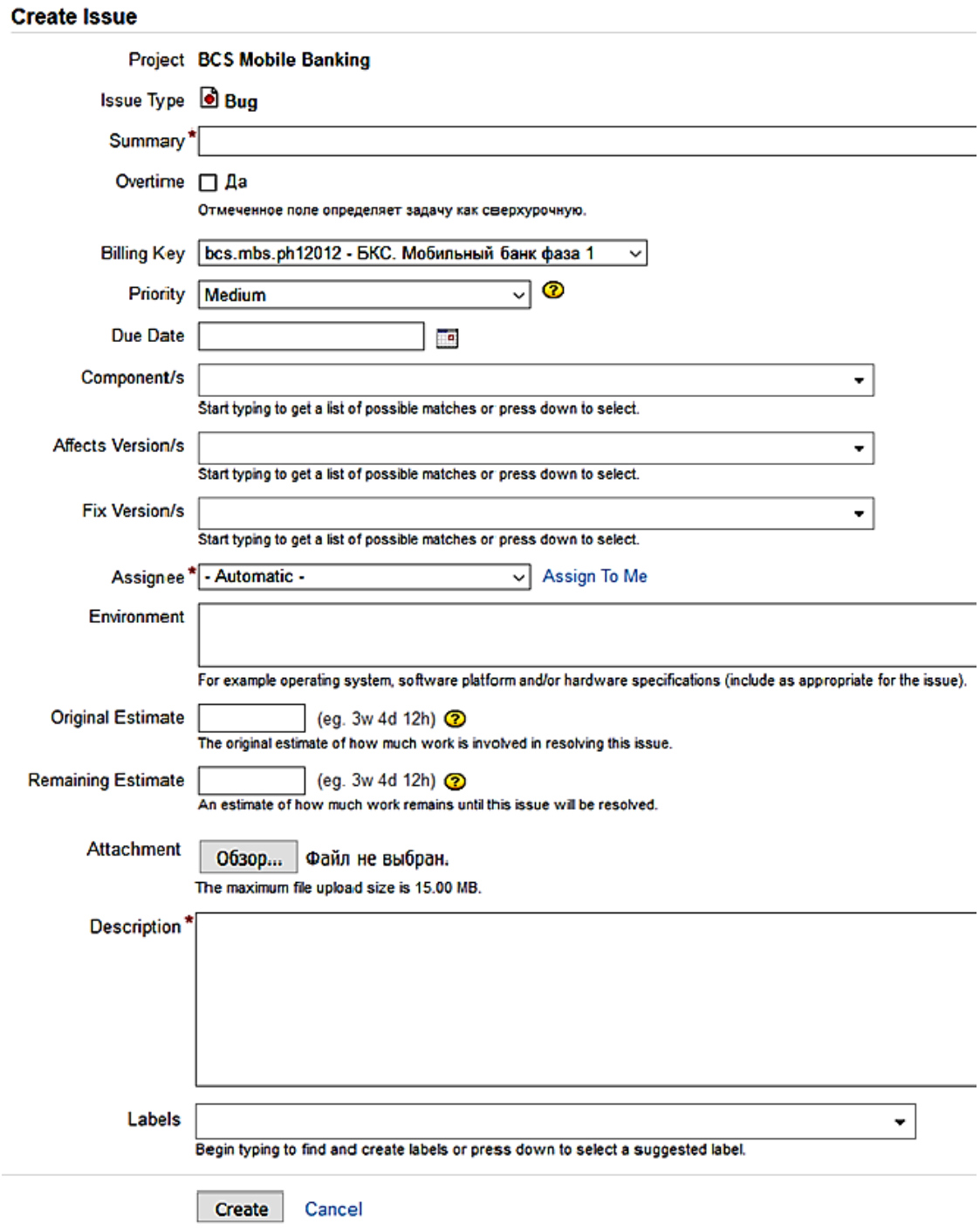

Рис. 21. Форма создания дефекта в системе Atlassian JIRA

Рис. 21 иллюстрирует систему JIRA, которая уже была настроена под нужды конкретного проекта, а потому имеются многочисленные дополнительные поля, а также поля, предзаполненные данными проекта.

Контрольные вопросы

1. Какие группы лиц, заинтересованных в тестировании, вам известны?
2. Приведите примеры заинтересованных лиц, относящихся к каждой группе.
3. Почему важно не просто определить круг заинтересованных в тестировании лиц, но и понять их цели?
4. Что необходимо сделать для того, чтобы процесс тестирования стал понятен всем заинтересованным лицам?
5. Назовите основные задачи тест-менеджера.
6. Кто может выполнять функции тест-менеджера?
7. Назовите основные задачи тестировщика.
8. Какие этапы составляют процесс планирования тестирования?
9. Что такое тест-план и для чего он необходим?
10. Что такое стратегия тестирования?
11. Как оцениваются трудозатраты проекта?
12. Какими бывают риски?
13. Что такое критерий входа?
14. Что такое критерий выхода?
15. Когда принято составлять график тестирования?
16. Как вы понимаете понятия мониторинга и контроля процесса тестирования?
17. Приведите примеры метрик тестирования.
18. Что такое матрица трассировки и как ее строят?
19. Для чего необходима отчётность по тестированию?
20. Как правильно завершать процесс тестирования?
21. Для чего необходима система управления тестами?
22. Какие системы управления тестами вам известны?
23. Каковы основные функции системы управления дефектами?
24. Что такое жизненный цикл дефекта и какие состояния он включает?
25. Чем отличается серьезность дефекта от его приоритета?
26. Какие системы управления дефектами вам известны?
27. Позволяют ли системы управления дефектами управлять проектом в целом?
28. Почему дефект может быть отклонен?
29. Что происходит с дефектом, если он был исправлен, но после проверки выяснилось, что он повторяется?
30. Для чего используются плагины в системе управления дефектами, например в Atlassian JIRA?

ЗАКЛЮЧЕНИЕ

Тестирование программных продуктов и информационных систем – сложный и трудоемкий процесс, который включает в себя различные этапы, регламентируется большим количеством стандартов и призван улучшить качество системы для конечных пользователей. Тестирование способно указать на наличие дефектов в системе, но не может подтвердить их полного отсутствия. Невозможно протестировать все сценарии и варианты, поэтому при построении процесса тестирования необходимо анализировать риски, брать за основу опыт предыдущих проектов или компаний и осуществлять планирование тестирования наиболее тщательным образом.

Качество и успех тестирования напрямую зависит от того, на каком этапе жизненного цикла программного продукта тестирование будет начато. Как известно, тестирование стоит начинать как можно раньше. Помимо непосредственной проверки готовой системы, подвергать тестированию следует документацию, чтобы уже на самых ранних стадиях предотвратить попадание дефектов в программный код.

Существует огромное количество видов тестирования, а также методик разработки тестовых сценариев. Методики позволяют отбирать тесты из бесконечного их числа таким образом, чтобы обеспечить наибольшее покрытие системы и уложиться в сроки, отведенные на проверку продукта.

Важен тот факт, что некоторые процессы проверки программных продуктов являются повторяющимися, например регрессионное тестирование. Кроме того, имеется большое количество сценариев, которые трудно проверять вручную, так как высока вероятность совершения тестировщиком ошибки. Здесь на помощь приходит автоматизация тестирования. На самом начальном этапе автоматизации можно столкнуться с трудностями выбора подходящего инструмента. Здесь на помощь снова приходит опыт предыдущих проектов, а также опыт коллег или конкурентов.

Тестирование, как и почти любой другой процесс, невозможно без контроля и управления. Тестирование порождает большое количество артефактов. Для удобной работы с ними существуют системы отслеживания ошибок, системы управления тестами и другие удобные инструменты, способные облегчить работу тестировщика, тест-менеджера и других участников процесса.

Обязательным является мониторинг и контроль тестирования, анализ его результатов, извлечение уроков. Для получения общей картины тестирования используют различные метрики, например: процент покрытия требований тестами, количество дефектов и их распределение по модулям системы, количество дефектов документации, процент прохождения тестов на каждом конкретном этапе тестирования. Нельзя просто остановить тестирование. Его необходимо грамотно завершить. Для этого следует проанализировать критерии выхода.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Тэллес, М.* Наука отладки [Электронный ресурс] / М. Тэллес, Ю. Хсих ; пер. с англ. С. Лунин. – Режим доступа: http://citforum.ru/programming/digest/scofdebug/index.shtml#4_4 (дата обращения: 17.02.2016).

2. Стандартный глоссарий терминов, используемых в тестировании программного обеспечения [Электронный ресурс] / А. Александров [и др.]. – Режим доступа: http://www.rstqb.org/fileadmin/user_upload/redaktion/rstqb_ru/downloads/ISTQB_Glossary_Russian_v2_2.pdf (дата обращения: 25.02.2016).

3. Сертифицированный тестировщик. Программа обучения базового уровня [Электронный ресурс] / А. Александров [и др.]. – Режим доступа: http://www.rstqb.org/fileadmin/user_upload/redaktion/rstqb_ru/downloads/ISTQB_CTFL_Syllabus_2011_RU.pdf (дата обращения: 25.02.2016).

4. Системное тестирование [Электронный ресурс]. – Режим доступа: http://www.protesting.ru/testing/levels/system.html (дата обращения: 25.02.2016).

5. Структурное тестирование [Электронный ресурс]. – Режим доступа: http://studopedia.ru/3_80803_strukturnoe-testirovanie.html (дата обращения: 25.02.2016).

6. ISO/IEC 12207:2008 [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/ISO/IEC_12207:2008 (дата обращения: 10.03.2016).

7. *Кулямин, В. В.* Методы верификации программного обеспечения [Электронный ресурс] / В. В. Кулямин. – Режим доступа: http://www.ict.edu.ru/ft/005645/62322e1-st09.pdf (дата обращения: 15.03.2016).

8. Стандарты документирования программных средств [Электронный ресурс]. – Режим доступа: http://www.slideserve.com/nigel-tillman/6468916 (дата обращения: 10.03.2016).

9. Обзор международного стандарта ISO/IEC 20000:2005. Процессы, сертификация ISO 20000 [Электронный ресурс]. – Режим доступа: http://www.itexpert.ru/rus/biblio/iso20k/ (дата обращения: 10.03. 2016).

10. Стандарты. ISO-20000 [Электронный ресурс]. – Режим доступа: http://tmguru.ru/baza-znanij/protsess-testirovaniya/standarty/ (дата обращения: 10.03.2016).

11. Идеи Деминга и ITIL [Электронный ресурс]. – Режим доступа: http://www.itexpert.ru/rus/ITEMS/77-24/ (дата обращения: 15.03. 2016).

12. *Graham, Bath*. The Software Test Engineer's Handbook / Graham Bath, Judy McKay. – USA : Rockynook, 2008. – 398 p. – ISBN 978-1-933952-24-6.

13. Автоматизированное тестирование программного обеспечения – основные понятия [Электронный ресурс]. – Режим доступа: http://www.protesting.ru/automation/ (дата обращения: 01.03.2016).

14. Управление тестированием [Электронный ресурс]. – Режим доступа: http://tmguru.ru/baza-znanij/ (дата обращения: 01.03.2016).

15. *Москаленко, Е.* Что такое Severity и Priority? Примеры из жизни [Электронный ресурс] / Е. Москаленко. – Режим доступа: http://evgmoskalenko.com/testing/severity-i-priority-primery.html (дата обращения: 08.03.2016).

ОГЛАВЛЕНИЕ

Учебное издание

ПЕРОЦКАЯ Вероника Николаевна
ГРАДУСОВ Денис Александрович

ОСНОВЫ ТЕСТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

Редактор Е. А. Лебедева

Технический редактор С. Ш. Абдуллаева

Корректор Т. В. Евстюничева

Компьютерная верстка Л. В. Макаровой

Подписано в печать 20.04.17.
Формат 60×84/16. Усл. печ. л. 5,81. Тираж 60 экз.
Заказ
Издательство
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых.
600000, Владимир, ул. Горького, 87.