25

Тестирование документации — это начальная стадия процесса тестирования, которая выступает как система раннего оповещения об ошибках. Процесс тестирования так или иначе начинается с документации и требований. Тестирование документации предполагает начало тестирования еще до разработки продукта. Тестировщик может указать на логические ошибки в постановке задачи, несоответствия в требованиях, а также составить чек-лист, список проверок по предоставленному требованию.

Как известно, хорошая документация должна обладать следующими **свойствами**:

* требования должны быть ***полными***, правильно и в полной мере описывать функцию, которую необходимо реализовать;
* ***однозначность*** — одинаковое восприятие требований всеми членами команды, никаких расхождений в трактовке быть не должно;
* ***непротиворечивость***: не должно быть противоречивых требований, конфликтующих между собой;
* ***необходимость***: требования должны отражать функциональности, действительно необходимые для пользователя, для удовлетворения пользователей;
* ***осуществимость:***насколько прописанные требования возможно реализовать;
* ***тестируемость***: возможность проверить все прописанные требования после их реализации, если требование является полным, одинаково воспринимается всеми участниками проекта, ни одна важная деталь не упущена — данное требование может быть протестировано.

**Тестирование документации и требований относится к разряду нефункционального тестирования. Существуют специальные техники для тестирования документации и требований:**

1. review требований

* беглый просмотр — это показ своей работы коллегам с целью получения обратной связи, вопросов и замечаний. Все отзывы и замечания помогут улучшить работу.
* технический просмотр — это вычитка документа группой специалистов, представляющих различные области. Документ/требования не могут быть качественными, пока хотя бы у одного из специалистов есть замечания.
* формальная инспекция — это редко используемая техника (например, при получении проекта, созданием которого занималась другая компания, на сопровождение и доработку), которая представляет собой систематизированный подход к анализу документации.

 2) вопросы — это одна из наиболее простых и эффективных техник выявления требований. Если что-то  в документах остается непонятным — задавайте вопросы. Для получения ответов на задаваемые вопросы, можно обратиться к менеджеру, более опытному специалисту, который ранее получил соответствующую информацию от заказчика, или к самому заказчику.

 3) тест-кейсы — требование должно быть проверяемым, значит должны существовать способы проверки корректности реализованного требования. Чем тщательнее продуман чек-лист, тем вероятнее определение проверяемости требований. Прежде, чем записывать возможные тест-кейсы, убедитесь в том, что вы понимаете требование. Для хорошего понимания конкретного требование поможет анализ других требований, которые вполне возможно будут каким либо образом связаны. Но если требование так и не удалось понять — скорее всего в нем есть неточность или ошибка. Когда же все требования будут хорошо сформулированы и протестированы, можно продолжать использование этой техники, совмещая разработку тест-кейсов с дополнительным тестированием.

 4) исследование поведения системы: тестировщик моделирует процесс работы  системой, созданной по тестируемым требованиям и ищет неоднозначные варианты поведения системы (чем-то мне напоминает исследовательское тестирование).

 5) графическое представление: для того, чтобы увидеть все требования полностью, очень удобно будет использовать рисунки, схемы, мокапы. На рисунках довольно просто заметить неточности, нестыковку элементов. После доработки всех неточностей, получится хорошее дополнение к текстовым требованиям.

22

PHP поддерживает несколько форм метапрограммирования, что реализуется с помощью таких механизмов, как Reflection API и Магические Методы. Доступно много Магических Методов, например: \_\_get(), \_\_set(), \_\_clone(), \_\_toString(), \_\_invoke(), и т.д., которые позволяют отслеживать поведение внутри класса.

Метод \_\_get() приминает имя свойства, к которому вы обращаетесь, в качестве аргумента. В приведенном выше примере сначала проверяется существование свойства в объекте и если оно существует, то возвращается его значение.  
  
Как и в примерах выше — вы не должны вызывать этот метод напрямую, PHP будет вызывать его каждый раз, при попытке получения доступа к не публичным свойствам класса.  
  
В обратной ситуации — если вы попытаетесь установить значение свойства, которое не является публичным — вы получите ошибку. И опять же, в PHP есть свой метод, который будет вызван при попытке установить в не публичное поле какое-либо значение. Данный метод принимает 2 параметра в качестве аргументов — свойство, в которое хотели записать значение, и само значение.  
  
Если вы хотите использовать данный метод, ваш класс получит свойство, на подобии этого:

*public function \_\_set($property, $value)*

*{*

*if (property\_exists($this, $property)) {*

*$this->$property = $value;*

*}*

*}*

*$tweet->text = 'Setting up my twttr';*

*echo $tweet->text; // 'Setting up my twttr'*

Если в классе не доступно какое-либо свойство напрямую(например с помощью модификатора доступа private) можно определить "обходной путь" - "магическую" функцию \_\_set(), в которую автоматически будет попадать любая попытка доступа к приватной переменной. Что делать с этими данными программист решает сам(реализует в теле метода).

22.2

* *Test Link*
* *Test IT*
* *Zephyr*
* *qTest*
* *PractiTest*
* *TestLodge*
* *TestRail*
* *Qase*
* *Tematoo*
* *Test Collab*
* *HP ALM*
* *Testuff*
* *XQual*

Выбор техник и программ всегда обусловлен сравнением текущих характеристик, стоимости и отзывов о данном продукте. Так, чтобы подобрать наиболее подходящий инструмент для управления тест-кейсами, можно руководствоваться такими принципами:

* **Управление тестированием** – создание и поддержка тестовых артефактов (тест-планов, тестовых случаев, тестовых сценариев или пользовательских историй).
* **Планирование цикла тестирования** – распределение ролей и обязанностей членов команды тестирования, возможность приоритизации тест-кейсов, установление четкой прослеживаемости между требованиями, тест-кейсами и дефектами.
* **Проведение тестирования** – удобный интерфейс для отметки результатов выполнения тест-кейсов, возможность добавлять новые тесты во время тестирования, сохранение пошаговой истории выполнения тест-кейсов.
* **Создание протокола тестирования** – возможность сбора метрик по количеству проделанных манипуляций, визуализация отчетов с помощью графиков.
* **Инструментарии выполнения задач** – возможность интеграции с системами отслеживания ошибок; создание, редактирование и отслеживание дефектов непосредственно с системы управления тест-кейсами.
* **Вспомогательные опции** – возможности экспорта и импорта данных, интеграция со сторонними приложениями, работа с crash-логами и прочее.

На основе всего вышеизложенного выделим десятку наиболее эффективных и популярных в 2019 году инструментов для управления тест-кейсами.

*21*

* Дымовое тестирование (smoke test139, intake test140, build verifi cation test141) направлено на проверку самой главной, самой важной, самой ключевой функциональности, неработоспособность которой делает бессмысленной саму идею использования приложения (или иного объекта, подвергаемого дымовому тестированию).
* Тестирование критического пути (critical path144 test) направлено на исследование функциональности, используемой типичными пользователями в типичной повседневной деятельности. Как видно из определения в сноске к англоязычной версии термина, сама идея позаимствована из управления проектами и трансформирована в контексте тестирования в следующую: существует большинство пользователей, которые чаще всего используют некое подмножество функций приложения (см. рисунок 2.3.g). Именно эти функции и нужно проверить, как только мы убедились, что приложение «в принципе работает» (дымовой тест прошёл успешно). Если по каким-то причинам приложение не выполняет эти функции или выполняет их некорректно, очень многие пользователи не смогут достичь множества своих целей. Пороговое значение метрики успешного прохождения «теста критического пути» уже немного ниже, чем в дымовом тестировании, но всё равно достаточно высоко (как правило, порядка 70–80–90 % — в зависимости от сути проекта).
* Расширенное тестирование (extended test145) направлено на исследование всей заявленной в требованиях функциональности — даже той, которая низко проранжирована по степени важности. При этом здесь также учитывается, какая функциональность является более важной, а какая — менее важной. Но при наличии достаточного количества времени и иных ресурсов тест-кейсы этого уровня могут затронуть даже самые низкоприоритетные требования.

20

**Конфигурационное тестирование**— ещё один из видов традиционного тестирования производительности. В этом случае вместо того, чтобы тестировать производительность системы с точки зрения подаваемой нагрузки, тестируется эффект влияния на производительность изменений в конфигурации. Хорошим примером такого тестирования могут быть эксперименты с различными методами балансировки нагрузки. Конфигурационное тестирование также может быть совмещено с нагрузочным, стресс или тестированием стабильности.

Для клиент-серверных приложений конфигурационное тестирование можно условно разделить на два уровня (для некоторых типов приложений может быть актуален только один):

1. Серверный
2. Клиентский

На первом (серверном) уровне, тестируется взаимодействие выпускаемого программного обеспечения с окружением, в которое оно будет установлено:

1. Аппаратные средства (тип и количество процессоров, объем памяти, характеристики сети / сетевых адаптеров и т.д.)
2. Программные средства (ОС, драйвера и библиотеки, стороннее ПО, влияющее на работу приложения и т.д.)

На следующем (клиентском) уровне, программное обеспечение тестируется с позиции его конечного пользователя и конфигурации его рабочей станции. На этом этапе будут протестированы следующие характеристики: удобство использования, функциональность. Для этого необходимо будет провести ряд тестов с различными конфигурациями рабочих станций:

1. Тип, версия и битность операционной системы (подобный вид тестирования называется **кросс-платформенное тестирование**)
2. Тип и версия Web барузера, в случае если тестируется Web приложение (подобный вид тестирования называется **кросс-браузерное тестирование**)
3. Тип и модель видео адаптера (при тестировании игр это очень важно)
4. Работа приложения при различных разрешениях экрана
5. Версии драйверов, библиотек и т.д. (для JAVA приложений версия JAVA машины очень важна, тоже можно сказать и для .NET приложений касательно версии .NET библиотеки)

**Порядок проведения тестирования**

Перед началом проведения конфигурационного тестирования рекомендуется:

* создавать матрицу покрытия (**матрица покрытия** - это таблица, в которую заносят все возможные конфигурации),
* проводить приоритезацию конфигураций (на практике, скорее всего, все желаемые конфигурации проверить не получится),
* шаг за шагом, в соответствии с расставленными приоритетами, проверяют каждую конфигурацию.

Уже на начальном этапе становится очевидно, что чем больше требований к работе приложения при различных конфигурациях рабочих станций, тем больше тестов нам необходимо будет провести. В связи с этим, рекомендуем, по возможности, автоматизировать этот процесс, так как именно при конфигурационном тестировании автоматизация реально помогает сэкономить время и ресурсы. Конечно же автоматизированное тестирование не является панацеей, но в данном случае оно окажется очень эффективным помощником.

19

**Абстрактный класс** — это **класс**, у которого не реализован один или больше методов (некоторые языки требуют такие методы помечать специальными ключевыми словами). **Интерфейс** — это **абстрактный класс**, у которого ни один метод не реализован, все они публичные и нет переменных **класса**.

**Интерфейс описывает только поведение. У него нет состояния. А у абстрактного класса состояние есть: он описывает и то, и другое.**

**Абстрактный класс связывает между собой и объединяет классы, имеющие очень близкую связь. В то же время, один и тот же интерфейс могут реализовать классы, у которых вообще нет ничего общего.**

**Классы могут реализовывать сколько угодно интерфейсов, но наследоваться можно только от одного класса.**

**19.3**

*существует достаточно простой алгоритм, позволяющий нам создавать эффективные проверки... Приступая к продумыванию чек-листа, тест-кейса или набора тест-кейсов, задайте себе следующие вопросы и получите чёткие ответы:*

1. *Что перед вами? Если вы не понимаете, что вам предстоит тестировать, вы не уйдёте дальше бездумных формальных проверок.*
2. *Кому и зачем оно нужно (и насколько это важно)? Ответ на этот вопрос позволит вам быстро придумать несколько характерных сценариев использования того, что вы собираетесь тестировать.*
3. *Как оно обычно используется? Это уже детализация сценариев и источник идей для позитивного тестирования (их удобно оформить в виде чек-листа).*
4. *Как оно может сломаться, т.е. начать работать неверно? Это также детализация сценариев использования, но уже в контексте негативного тестирования (их тоже удобно оформить в виде чек-листа).*

К этому алгоритму можно добавить ещё небольшой перечень универсальных рекомендаций, которые позволят вам проводить тестирование лучше: • Начинайте как можно раньше — уже с момента появления первых требований можно заниматься их тестированием и улучшением, можно писать чек-листы и тест-кейсы, можно уточнять план тестирования, готовить тестовое окружение и т. д. • Если вам предстоит тестировать что-то большое и сложное, разбивайте его на модули и подмодули, функциональность подвергайте функциональной декомпозиции307 — т. е. добейтесь такого уровня детализации, при котором вы можете без труда удержать в голове всю информацию об объекте тестирования. • Обязательно пишите чек-листы. Если вам кажется, что вы сможете запомнить все идеи и потом легко их воспроизвести, вы ошибаетесь. Исключений не бывает. • По мере создания чек-листов, тест-кейсов и т.д. прямо в текст вписывайте возникающие вопросы. Когда вопросов накопится достаточно, соберите их отдельно, уточните формулировки и обратитесь к тому, кто может дать ответы. • Если используемое вами инструментальное средство позволяет использовать косметическое оформление текста — используйте (так текст будет лучше читаться), но старайтесь следовать общепринятым традициям и не раскрашивать каждое второе слово в свой цвет, шрифт, размер и т. д. • Используйте технику беглого просмотра{50} для получения отзыва от коллег и улучшения созданного вами документа. • Планируйте время на улучшение тест-кейсов (исправление ошибок, доработку по факту изменения требований и т. д.). • Начинайте проработку (и выполнение) тест-кейсов с простых позитивных проверок наиболее важной функциональности. Затем постепенно повышайте сложность проверок, помня не только о позитивных{83}, но и о негативных{83} проверках. • Помните, что в основе тестирования лежит цель. Если вы не можете быстро и просто сформулировать цель созданного вами тест-кейса, вы создали плохой тест-кейс. • Избегайте избыточных, дублирующих друг друга тест-кейсов. Минимизировать их количество вам помогут техники классов эквивалентности{96}, граничных условий{96}, доменного тестирования.

Если показательность{142} тест-кейса можно увеличить, при этом не сильно изменив его сложность и не отклонившись от исходной цели, сделайте это. • Помните, что очень многие тест-кейсы требуют отдельной подготовки, которую нужно описать в соответствующем поле тест-кейса. • Несколько позитивных тест-кейсов{83} можно безбоязненно объединять, но объединение негативных тест-кейсов{83} почти всегда запрещено. • Подумайте, как можно оптимизировать созданный вами тест-кейс (набор тест-кейсов и т. д.) так, чтобы снизить трудозатраты на его выполнение. • Перед тем как отправлять финальную версию созданного вами документа, ещё раз перечитайте написанное (в доброй половине случаев найдёте опечатку или иную недоработку).

Есть простая логика: • Тесты ищут ошибки. • Но все ошибки найти невозможно. • Значит, наша задача — найти максимум ВАЖНЫХ ошибок за имеющееся время. Под важными ошибками здесь мы понимаем такие, которые приводят к нарушению важных для пользователя функций или свойств продукта. Функции и свойства разделены не случайно — безопасность, производительность, удобство и т.д. не относятся к функциям, но играют ничуть не менее важную роль в формировании удовлетворённости заказчика и конечных пользо ва телей.

17

https://gos-it.fandom.com/wiki/%D0%A2%D1%80%D0%B5%D0%B1%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F\_%D0%BA\_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%BC%D1%83\_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8E

## **Источники требований**

* Федеральное и муниципальное отраслевое законодательство (конституция, законы, распоряжения)
* Нормативное обеспечение организации (регламенты, положения, уставы, приказы)
* Текущая организация деятельности объекта автоматизации
* Модели деятельности (диаграммы бизнес-процессов)
* Представления и ожидания потребителей и пользователей системы
* Журналы использования существующих программно-аппаратных систем
* Конкурирующие программные продукты

**Требования к программному обеспечению** — совокупность утверждений относительно атрибутов, свойств или качеств [программной системы](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), подлежащей реализации. Создаются в процессе разработки требований к [программному обеспечению](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) (ПО), в результате [анализа требований](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7_%D1%82%D1%80%D0%B5%D0%B1%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B9).

Фаза разработки требований может быть разбита на выявление требований (сбор, понимание, рассмотрение и выяснение потребностей заинтересованных лиц), анализ (проверка целостности и законченности), спецификация (документирование требований) и проверка правильности.

14

#### **Отчёт о дефекте (и сам дефект вместе с ним) проходит определённые стадии жизненного цикла:**

* **Обнаружен (submitted)** — начальное состояние отчёта (иногда называется «Новый» (new)), в котором он находится сразу после создания. Некоторые средства также позволяют сначала создавать черновик (draft) и лишь потом публиковать отчёт.
* **Назначен (assigned)** — в это состояние отчёт переходит с момента, когда кто-то из проектной команды назначается ответственным за исправление дефекта. Назначение ответственного может производится решением лидера команды разработки, коллегиально, по добровольному принципу, иным принятым в команде способом или выполняется автоматически на основе определённых правил.
* **Исправлен (fixed)**— в это состояние отчёт переводит ответственный за исправление дефекта член команды после выполнения соответствующих действий по исправлению.
* **Проверен (verified)** — в это состояние отчёт переводит тестировщик, удостоверившись, что дефект на самом деле был устранён. Как правило, такую проверку выполняет тестировщик, изначально написавший отчёт о дефекте.
* **Закрыт (closed)** — состояние отчёта, означающее, что по данному дефекту не планируется никаких дальнейших действий. Здесь есть некоторые расхождения в жизненном цикле, принятом в разных инструментальных средствах управления отчётами о дефектах.
* **Открыт заново (reopened)** — в это состояние (как правило, из состояния «Исправлен») отчёт переводит тестировщик, удостоверившись, что дефект по-прежнему воспроизводится на билде, в котором он уже должен быть исправлен.
* **Рекомендован к отклонению (to be declined)** — в это состояние отчёт о дефекте может быть переведён из множества других состояний, чтобы вынести на рассмотрение вопрос об отклонении отчёта по той или иной причине. Если рекомендация является обоснованной, то отчёт переводится в состояние «Отклонён».
* **Отклонён (declined)**— в это состояние отчёт переводится в случаях, подробно описанных в пункте «Закрыт»: если средство управления отчётами о дефектах предполагает использование этого состояния вместо состояния «Закрыт» для тех или иных резолюций по отчёту.
* **Отложен (deferred)** — в это состояние отчёт переводится в случае, если исправление дефекта в ближайшее время является нерациональным или не представляется возможным, однако есть основания полагать, что в обозримом будущем ситуация исправится (выйдет новая версия библиотеки, вернётся из отпуска специалист по необходимой технологии, изменятся требования заказчика и т.д.)

Информация о дефектах, которая может быть полезна для улучшения качества, включает следующие вопросы:

* Что было не так? Решать нужно саму проблему, а не ее симптомы. Например, зацикливание - это симптом, а изменение индекса цикла - это проблема.
* Когда была создана эта проблема? Какое именно действие при разработке явилось ее источником? Это была проблема в требованиях? Проектировании системы? Коде? Тестировании?
* Когда проблема была выявлена? Может, она и не была сразу же устранена, но что нас интересует: сколько она существовала до того, как мы ее обнаружили?
* Каким образом была найдена эта проблема? Способ обнаружения можно внедрить в постоянно используемую практику.
* Можно ли было обнаружить ее раньше? Есть ли какой-либо процесс контроля качества, который мог бы ее выявить, будь он эффективнее?
* Сколько стоило устранение этой проблемы? Этот момент очень легко недооценить. Убедитесь, что Вы учли процесс диагностики проблемы и всю работу по ее устранению, которую Вам пришлось сделать, включая ре-дизайн, переписывание кода, ре-компиляцию, переработку тестов, повторное тестирование, повторный релиз, выпуск заплатки, формирование отчета по дефекту, отчет по статусу проекта и т.д. (не забудьте еще возможные затраты на исправление подпорченной репутации на рынке ПО).
* Какого рода была эта проблема? Когда у Вас огромное количество дефектов, то их категоризация облегчает анализ и обучение.

12

*Если коротко, то архитектурный стиль микросервисов — это подход, при котором единое приложение строится как набор небольших сервисов, каждый из которых работает в собственном процессе и коммуницирует с остальными используя легковесные механизмы, как правило HTTP. Эти сервисы построены вокруг бизнес-потребностей и развертываются независимо с использованием полностью автоматизированной среды. Существует абсолютный минимум централизованного управления этими сервисами. Сами по себе эти сервисы могут быть написаны на разных языках и использовать разные технологии хранения данных.*

**Свойства микросервисов**

Важнейшим свойством является **автономность**. Автономность распространяется на все, что касается разработки микросервисов:

* Автономность на уровне модели предметной области (**микросервисы строятся вокруг бизнес-концепций**)
* **Автономное развертывание**
* Автономное тестирование
* Независимое эволюционное развитие (**сокрытие внутренней реализации**)
* **Изоляция сбоев**

11

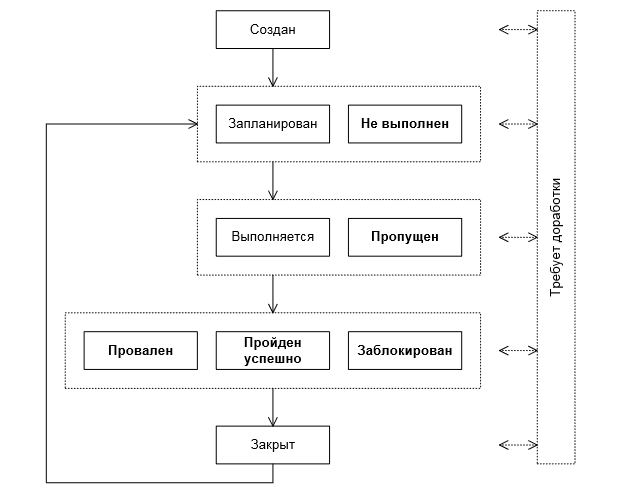


Рисунок 1 Жизненный цикл тест-кейса

**Создан (new)**- типичное начальное состояние практически любого артефакта. Тест-кейс автоматически переходит в это состояние после создания.

**Запланирован (planned, ready for testing)**- в этом состоянии тест-кейс находится, когда он или явно включён в план ближайшей итерации тестирования, или, как минимум, готов для выполнения.

**Не выполнен (not tested)**- в некоторых системах управления тест-кейсами это состояние заменяет собой предыдущее («запланирован»). Нахождение тест-кейса в данном состоянии означает, что он готов к выполнению, но ещё не был выполнен.

**Выполняется (work in progress)** - если тест-кейс требует длительное время для выполнения, то он может быть переведён в это состояние для подчёркивания того факта, что работа идёт, и скоро можно ожидать её результатов. Если выполнение тест-кейса занимает мало времени, это состояние, как правило, пропускается, а тест-кейс сразу переводится в одно из трёх следующих состояний - «провален», «пройден успешно» или «заблокирован».

**Пропущен (skipped)**- бывают ситуации, когда выполнение тест-кейса отменяется по соображениям нехватки времени или изменения логики тестирования.

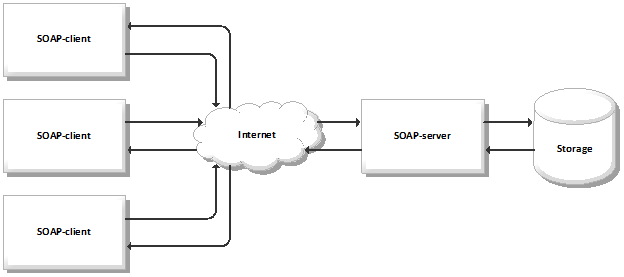
**Провален (failed)**- данное состояние означает, что в процессе выполнения тест-кейса был обнаружен дефект, заключающийся в том, что ожидаемый результат по как минимум одному шагу тест-кейса не совпадает с фактическим результатом. Если в процессе выполнения тест-кейса был «случайно» обнаружен дефект, никак не связанный с шагами тест-кейса и их ожидаемыми результатами, тест-кейс считается пройденным успешно (при этом, естественно, по обнаруженному дефекту создаётся отчёт о дефекте).

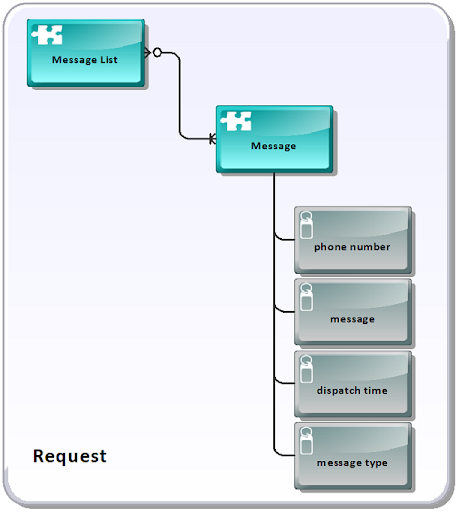
**Пройден успешно (passed)** - данное состояние означает, что в процессе выполнения тест-кейса не было обнаружено дефектов, связанных с расхождением ожидаемых и фактических результатов его шагов.

**Заблокирован (blocked)**- данное состояние означает, что по какой-то причине выполнение тест-кейса невозможно (как правило, такой причиной является наличие дефекта, не позволяющего реализовать некий пользовательский сценарий).

**Закрыт (closed)**- очень редкий случай, т.к. тест-кейс, как правило, оставляют в состояниях «провален / пройден успешно / заблокирован / пропущен». В некоторых системах управления тест-кейс переводят в данное состояние, чтобы подчеркнуть тот факт, что на данной итерации тестирования все действия с ним завершены.

**Требует доработки (not ready)**- как видно из схемы, в это состояние (или из него) тест-кейс может быть переведён в любой момент времени, если в нём будет обнаружена ошибка, если изменятся требования, по которым он был написан, или наступит иная ситуация, не позволяющая считать тест-кейс пригодным для выполнения и перевода в иные состояния.





10

Функциональное тестирование является одним из ключевых видов тестирования, задача которого – установить соответствие разработанного программного обеспечения (ПО) исходным функциональным требованиям компании клиента. То есть проведение функционального тестирования позволяет проверить способность информационной системы в определенных условиях решать задачи, нужные пользователям.

### Основные этапы функционального тестирования

**Подготовка** — Проводится анализ исходных документов о системе: функциональные и бизнес-требования, техническое задание, паспорт проекта. Также происходят разработка и согласование плана тестирования, тест-кейсов, согласование проектных сроков, числа итераций, оценка возможных рисков. Задачи по этому этапу выполняются совместно с представителями заказчика.

**Проведение** — Функциональное тестирование ведется вручную по подготовленным заранее тестовым сценариям с занесением всех найденных ошибок в багтрекинговую систему. В случае отсутствия такой системы у компании клиента, компания модератор краудтестинга может: предоставить систему управления тестированием на своей площадке; поставить компании клиенту лицензии; использовать имеющиеся у компании клиента средства; обходиться только офисным пакетом; поставить процесс тестирования у компании клиента на основе бесплатных средств.

**Отчет** — Происходит разработка и согласование отчетов о проведенном тестировании со списком обнаруженных отклонений и рекомендациями по улучшению системы. Если необходимо, проводится обучение пользователей.

9

Классификация тестирования по доступу к коду и архитектуре приложения.

• Метод белого ящика (white box testing119, open box testing, clear box testing, glass box testing) — у тестировщика есть доступ к внутренней структуре и коду приложения, а также есть достаточно знаний для понимания увиденного. Выделяют даже сопутствующую тестированию по методу белого ящика глобальную технику — тестирование на основе дизайна (design-based testing120). Для более глубокого изучения сути метода белого ящика рекомендуется ознакомиться с техниками исследования потока управления{98} или потока данных{98}, использования диаграмм состояний{98}. Некоторые авторы склонны жёстко связывать этот метод со статическим тестированием, но ничто не мешает тестировщику запустить код на выполнение и при этом периодически обращаться к самому коду (а модульное тестирование{77} и вовсе предполагает запуск кода на исполнение и при этом работу именно с кодом, а не с «приложением целиком»).

• Метод чёрного ящика (black box testing121, closed box testing, specifi cation-based testing) — у тестировщика либо нет доступа к внутренней структуре и коду приложения, либо недостаточно знаний для их понимания, либо он сознательно не обращается к ним в процессе тестирования. При этом абсолютное большинство перечисленных на рисунках 2.3.b и 2.3.c видов тестирования работают по методу чёрного ящика, идею которого в альтернативном определении можно сформулировать так: тестировщик оказывает на приложение воздействия (и проверяет реакцию) тем же способом, каким при реальной эксплуатации приложения на него воздействовали бы пользователи или другие приложения. В рамках тестирования по методу чёрного ящика основной информацией для создания тест-кейсов выступает документация (особенно — требования (requirements-based testing122)) и общий здравый смысл (для случаев, когда поведение приложения в некоторой ситуации не регламентировано явно; иногда это называют «тестированием на основе неявных требований», но канонического определения у этого подхода нет).

• Метод серого ящика (gray box testing123) — комбинация методов белого ящика и чёрного ящика, состоящая в том, что к части кода и архитектуры у тестировщика доступ есть, а к части — нет. На рисунках 2.3.b и 2.3.c этот метод обозначен особым пунктиром и серым цветом потому, что его явное упоминание — крайне редкий случай: обычно говорят о методах белого или чёрного ящика в применении к тем или иным частям приложения, при этом понимая, что «приложение целиком» тестируется по методу серого ящика.

8

Классификация тестирования по степени автоматизации.

Ручное тестирование (manual testing125) — тестирование, в котором тест-кейсы выполняются человеком вручную без использования средств автоматизации. Несмотря на то что это звучит очень просто, от тестировщика в те или иные моменты времени требуются такие качества, как терпеливость, наблюдательность, креативность, умение ставить нестандартные эксперименты, а также умение видеть и понимать, что происходит «внутри системы», т. е.  как внешние воздействия на приложение трансформируются в его внутренние процессы. • Автоматизированное тестирование (automated testing, test automation126) — набор техник, подходов и инструментальных средств, позволяющий исключить человека из выполнения некоторых задач в процессе тестирования. Тест-кейсы частично или полностью выполняет специальное инструментальное средство, однако разработка тест-кейсов, подготовка данных, оценка результатов выполнения, написания отчётов об обнаруженных дефектах — всё это и многое другое по-прежнему делает человек.

7

**Предметная область** – это мысленно ограниченная область реальной действительности, подлежащая описанию или моделированию и исследованию.

Методы обследования предметной области

1. метод наблюдения**;** **Данный метод не обеспечивает полного и достоверного изучения предметной области. Целесообразен, когда изучаемый вопрос не является трудным для понимания и требуется лишь уточнение некоторых деталей.**
2. метод опроса**;** **Данный метод в настоящее время наиболее распространен, хотя и не является полным и точным**
3. метод личного участия**;** **предполагает изучение предметной области «изнутри» путем выполнения определенных служебных функций с последующим их описанием. Он более достоверен и желателен по сравнению с предыдущими методами.**
4. метод анализа материалов; предполагает изучение существа вопроса на основе документов, отчетов, распоряжений, квитанций и т.д., имеющих место в соответствующей предметной области. Он наиболее точен и научно обоснован.
5. метод организационно– функционального анализа. заключает в себе наиболее полное обследование предметной области путем изучения отдельных функциональных задач и операций, выполняемых различными организационными звеньями предприятия или учреждения: их параметров и принципов взаимодействия.

3

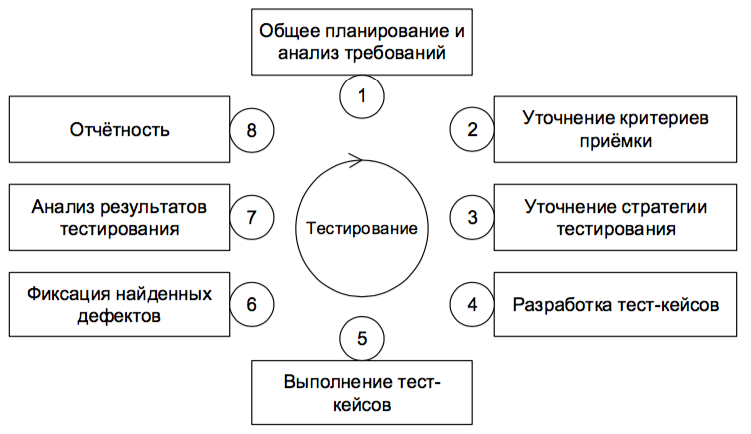


Рисунок 2 Жизненный цикл тестирования

**Стадия 1** (общее планирование и анализ требований) объективно необходима, как минимум, для того, чтобы иметь ответ на такие вопросы, как: что нам предстоит тестировать; как много будет работы; какие есть сложности; всё ли необходимое у нас есть и т.п. Как правило, получить ответы на эти вопросы невозможно без анализа требований, т.к. именно требования являются первичным источником ответов.

**Стадия 2** (уточнение критериев приёмки) позволяет сформулировать или уточнить метрики и признаки возможности или необходимости начала тестирования, приостановки и возобновления тестирования, завершения или прекращения тестирования.

**Стадия 3** (уточнение стратегии тестирования) представляет собой ещё одно обращение к планированию, но уже на локальном уровне: рассматриваются и уточняются те части стратегии тестирования, которые актуальны для текущей итерации.

**Стадия 4** (разработка тест-кейсов) посвящена разработке, пересмотру, уточнению, доработке, переработке и прочим действиям с тест-кейсами, наборами тест-кейсов, тестовыми сценариями и иными артефактами, которые будут использоваться при непосредственном выполнении тестирования.

**Стадия 5**(выполнение тест-кейсов) и **стадия 6** (фиксация найденных дефектов) тесно связаны между собой и фактически выполняются параллельно: дефекты фиксируются сразу по факту их обнаружения в процессе выполнения тест-кейсов. Однако зачастую после выполнения всех тест-кейсов и написания всех отчётов о найденных дефектах проводится явно выделенная стадия уточнения, на которой все отчёты о дефектах рассматриваются повторно с целью формирования единого понимания проблемы и уточнения таких характеристик дефекта, как важность и срочность.

**Стадия 7**(анализ результатов тестирования) и **стадия 8**(отчётность) также тесно связаны между собой и выполняются практически параллельно. Формулируемые на стадии анализа результатов выводы напрямую зависят от плана тестирования, критериев приёмки и уточнённой стратегии, полученных на стадиях 1, 2 и 3. Полученные выводы оформляются на стадии 8 и служат основой для стадий 1, 2 и 3 следующей итерации тестирования. Таким образом, цикл замыкается.