Лекция №7 Нефукциональное тестирование

### **Введение**

Нефункциональное тестирование - это тестирование, которое проводится для оценки **характеристик** программного обеспечения. Проверяется не корректность работы функций приложения, а сопутствующие характеристики (надежность, скорость работы, совместимость с другим ПО или оборудованием, и т.д.).

#### **Виды нефункционального тестирования**

* Тестирование производительности;
* [Тестирование безопасности](https://dist.belstu.by/mod/assign/view.php?id=57440);
* Тестирование эргономичности (usability testing);
* Тестирование совместимости;
* UI тестирование;
* Тестирование локализации и интернационализации;
* A/B тестирование;
* Тестирование на отказ и восстановление;
* Тестирование на соответствие стандартам;
* Тестирование на прерывания (работы мобильного ПО);
* Тестирование соединения (работы мобильного ПО).

### **Тестирование производительности**

**Тестирование производительности в инженерии программного обеспечения** — тестирование, которое проводится с целью определения, как быстро работает вычислительная система или её часть под определённой нагрузкой. Также может служить для проверки и подтверждения других атрибутов качества системы, таких как масштабируемость, надёжность и потребление ресурсов.

**Тестирование производительности** — это одна из сфер деятельности развивающейся в области информатики инженерии производительности, которая стремится учитывать производительность на стадии моделирования и проектирования системы, перед началом основной стадии кодирования.

**Направления тестирования производительности:**

* нагрузочное (load)
* стресс (stress)
* тестирование стабильности (endurance or soak or stability)
* конфигурационное (configuration)

Возможны два подхода к тестированию производительности программного обеспечения:

1. в терминах рабочей нагрузки: программное обеспечение подвергается тестированию в ситуациях, соответствующих различным сценариям использования;
2. в рамках бета-тестирования, когда система испытывается реальными конечными пользователями.

### **Нагрузочное тестирование**

**Нагрузочное тестирование** — это простейшая форма тестирования производительности. Нагрузочное тестирование обычно проводится для того, чтобы оценить поведение приложения под заданной ожидаемой нагрузкой. Этой нагрузкой может быть, например, ожидаемое количество одновременно работающих пользователей приложения, совершающих заданное число транзакций за интервал времени. Такой тип тестирования обычно позволяет получить время отклика всех самых важных бизнес-транзакций. В случае наблюдения за базой данных, сервером приложений, сетью и т. д., этот тип тестирования может также идентифицировать некоторые узкие места приложения.

#### **Стресс-тестирование**

**Стресс-тестирование** обычно используется для понимания пределов пропускной способности приложения. Этот тип тестирования проводится для определения надёжности системы во время экстремальных или диспропорциональных нагрузок и отвечает на вопросы о достаточной производительности системы в случае, если текущая нагрузка сильно превысит ожидаемый максимум.

#### **Тестирование стабильности**

**Тестирование** **стабильности**проводится с целью убедиться в том, что приложение выдерживает ожидаемую нагрузку в течение длительного времени. При проведении этого вида тестирования осуществляется наблюдение за потреблением приложением памяти, чтобы выявить потенциальные утечки. Кроме того, такое тестирование выявляет деградацию производительности, выражающуюся в снижении скорости обработки информации и/или увеличении времени ответа приложения после продолжительной работы по сравнению с началом теста.

#### **Конфигурационное тестирование**

**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.

В общих случаях тестирование производительности может служить разным целям:

* С целью демонстрации того, что система удовлетворяет критериям производительности.
* С целью определения, производительность какой из двух или нескольких систем лучше.
* С целью определения, какой элемент нагрузки или часть системы приводит к снижению производительности.

Многие тесты на производительность делаются без попытки осмыслить их реальные цели. Перед началом тестирования всегда должен быть задан бизнес-вопрос: «Какую цель мы преследуем, тестируя производительность?». Ответы на этот вопрос являются частью технико-экономического обоснования (или business case) тестирования. Цели могут различаться в зависимости от технологий, используемых приложением, или его назначения, однако, они всегда включают что-то из нижеследующего:

**Параллелизм / Пропускная способность**

Если конечными пользователями приложения считаются пользователи, выполняющие логин в систему в любой форме, то в этом случае крайне желательно достижение параллелизма. По определению это максимальное число параллельных работающих пользователей приложения, поддержка которого ожидается от приложения в любой момент времени. Модель поведения пользователя может значительно влиять на способность приложения к параллельной обработке запросов, особенно если он включает в себя периодически вход и выход из системы.

Если концепция приложения не заключается в работе с конкретными конечными пользователями, то преследуемая цель для производительности будет основана на максимальной пропускной способности или числе транзакций в единицу времени. Хорошим примером в данном случае будет являться просмотр веб-страниц, например, на портале Wikipedia.

**Время ответа сервера**

Эта концепция строится вокруг времени ответа одного узла приложения на запрос, посланный другим. Простым примером является HTTP 'GET' запрос из браузера рабочей станции на веб-сервер. Практически все приложения, разработанные для нагрузочного тестирования работают именно по этой схеме измерений. Иногда целесообразно ставить задачи по достижению производительности времени ответа сервера среди всех узлов приложения.

**Время отображения**

Время отображения — одно из самых сложных для приложения для нагрузочного тестирования понятий, так как в общем случае они не используют концепцию работы с тем, что происходит на отдельных узлах системы, ограничиваясь только распознаванием периода времени в течение которого нет сетевой активности. Для того, чтобы замерить время отображения, в общем случае требуется включать функциональные тестовые сценарии в тесты производительности, но большинство приложений для тестирования производительности не включают в себя такую возможность.

#### **Требования к производительности**

Очень важно детализировать требования к производительности и документировать их в каком-либо плане тестирования производительности. В идеальном случае это делается на стадии разработки требований при разработке системы, до проработки деталей её дизайна. См. Инженерия производительности.

Однако тестирование производительности часто не проводится согласно спецификации, так как нет зафиксированного понимания о максимальном времени ответа для заданного числа пользователей. Тестирование производительности часто используется как часть процесса профайлинга производительности. Его идея заключается в том, чтобы найти «слабое звено» — такую часть системы, соптимизировав время реакции которой, можно улучшить общую производительность системы. Определение конкретной части системы, стоящей на этом критическом пути, иногда очень непростая задача, поэтому некоторые приложения для тестирования включают в себя (или могут быть добавлены с помощью add-on’ов) инструменты, запущенные на сервере (агенты) и наблюдающие за временем выполнения транзакций, временем доступа к базе данных, оверхедами сети и другими показателями серверной части системы, которые могут быть проанализированы вместе с остальной статистикой по производительности.

Тестирование производительности может проводиться с использованием глобальной сети и даже в географически удаленных местах, если учитывать тот факт, что скорость работы сети Интернет зависит от местоположения. Оно также может проводиться и локально, но в этом случае необходимо настроить сетевые маршрутизаторы таким образом, чтобы появилась задержка, присутствующая во всех публичных сетях. Нагрузка, прилагаемая к системе, должна совпадать с реальным положением дел. Так например, если 50 % пользователей системы для доступа к системе используют сетевой канал шириной 56К, а другая половина использует оптический канал, то компьютеры, создающие тестовую нагрузку на систему должны использовать те же соединения (идеальный вариант) или эмулировать задержки вышеуказанных сетевых соединений, следуя заданным профайлам пользователей.

Требования к производительности должны адресовать следующие, как минимум, вопросы:

* Что охватывается тестом производительности? Какие подсистемы, компоненты, интерфейсы и т. д. должны быть протестированы?
* Если в тест включаются пользовательские интерфейсы, то сколько одновременно работающих в системе пользователей ожидается для каждого интерфейса (необходимо определить пиковые и нормальные значения)
* Как выглядит аппаратная составляющая тестируемой системы? (Необходимо описать все сервера и сетевое оборудование)
* Каков сценарий использования каждого компонента системы? (например, 20 % запросов составляет вход в систему, 40 % — поиск, 30 % — выбор элемента, 10 % — выход из системы)
* Каков сценарий использования системы? [в одном тесте на производительность могут быть задействованы разные сценарии использования каждого компонента]
* Каковы требования ко времени выполнения серии операций серверной части приложения?

#### **Основные показатели (метрики) производительности**

Одним из результатов, получаемых при нагрузочном тестировании и используемых в дальнейшем для анализа, являются показатели производительности приложения. Основные из них разобраны ниже.

**1. Потребление ресурсов центрального процессора (CPU, %)**

Метрика, показывающая сколько времени из заданного определённого интервала было потрачено процессором на вычисления для выбранного процесса. В современных системах важным фактором является способность процесса работать в нескольких потоках, для того, чтобы процессор мог производить вычисления параллельно. Анализ истории потребления ресурсов процессора может объяснять влияние на общую производительность системы потоков обрабатываемых данных, конфигурации приложения и операционной системы, многопоточности вычислений, и других факторов.

**2. Потребление оперативной памяти (Memory usage, Mb)**

Метрика, показывающая количество памяти, использованной приложением. Использованная память может делиться на три категории:

**Virtual** — объём виртуального адресного пространства, которое использует процессор. Этот объём не обязательно подразумевает использование соответствующего дискового пространства или оперативной памяти. Виртуальное пространство конечно и процесс может быть ограничен в возможности загружать необходимые библиотеки.

**Private**— объём адресного пространства, занятого процессом и не разделяемого с другими процессами.

**Working Set**— набор страниц памяти, недавно использованных процессом. В случае, когда свободной памяти достаточно, страницы остаются в наборе, даже если они не используются. В случае когда, свободной памяти остается мало, использованные страницы удаляются.

При работе приложения память заполняется ссылками на объекты, которые, в случае неиспользования, могут быть очищены специальным автоматическим процессом, называемым «сборщиком мусора» (англ. Garbage Collector). Время затрачиваемое процессором на очистку памяти таким способом может быть значительным, в случае, когда процесс занял всю доступную память (в Java — так называемый «постоянный Full GC») или когда процессу выделены большие объёмы памяти, нуждающиеся в очистке. На время, требующееся для очистки памяти, доступ процесса к страницам выделенной памяти может быть заблокирован, что может повлиять на конечное время обработки этим процессом данных.

**3. Потребление сетевых ресурсов**

Эта метрика не связана непосредственно с производительностью приложения, однако её показатели могут указывать на пределы производительности системы в целом.

**4. Работа с дисковой подсистемой (I/O Wait)**

Работа с дисковой подсистемой может значительно влиять на производительность системы, поэтому сбор статистики по работе с диском может помогать выявлять узкие места в этой области. Большое количество чтений или записей может приводить к простаиванию процессора в ожидании обработки данных с диска и в итоге увеличению потребления CPU и увеличению времени отклика.

**5. Время выполнения запроса (request response time, ms)**

Время выполнения запроса приложением остаётся одним из самых главных показателей производительности системы или приложения. Это время может быть измерено на серверной стороне, как показатель времени, которое требуется серверной части для обработки запроса; так и на клиентской, как показатель полного времени, которое требуется на сериализацию / десериализацию, пересылку и обработку запроса. Надо заметить, что не каждое приложение для тестирования производительности может измерить оба этих времени.

### **Другие виды нефункционального тестирования**

#### **Тестирование эргономичности (usability testing)**

Исследование, выполняемое с целью определения, удобен ли некоторый искусственный объект (такой как веб-страница, пользовательский интерфейс или устройство) для его предполагаемого применения. Это метод оценки удобства продукта в использовании, основанный на привлечении пользователей в качестве тестировщиков, испытателей и суммировании полученных от них выводов.

#### **Тестирование совместимости (compatibility testing)**

Тестирование, целью которого является проверка корректной работы приложения в определенном окружении.

Может проверяться совместимость с:

* Аппаратная платформа;
* Сетевые устройства;
* Периферия (принтеры, CD/DVD-приводы, веб-камеры и пр.);
* Операционная система (Unix, Windows, MacOS, ...);
* Базы данных (Oracle, MS SQL, MySQL, ...);
* Системное программное обеспечение (веб-сервер, файрволл, антивирус, ...);
* Браузеры (Internet Explorer, Firefox, Opera, Chrome, Safari, и др.)

#### **Тестирование графического пользовательского интерфейса**

Тестирование, проверяющее соответствие внешнего вида продукта заявленным дизайнам и требованиям.

#### **Тестирование глобализации**

**Локализация программного обеспечения**

Процесс адаптации программного обеспечения к культуре какой-либо страны. Как частность — перевод пользовательского интерфейса, документации и сопутствующих файлов программного обеспечения с одного языка на другой.

**Интернационализация программного обеспечения**

Технологические приёмы разработки, упрощающие адаптацию продукта к языковым и культурным особенностям региона (регионов), отличного от того, в котором разрабатывался продукт.

**Отличие локализации от интернационализации:**

Интернационализация производится на начальных этапах разработки, в то время как локализация — для каждого целевого языка.

**Примеры:**

* Наличия перевода во всем приложении;
* Отображения букв/символов языков;
* Корректность сортировки и фильтрации в разных локализациях;
* Графическое отображение текста;
* Отображение аудио и видео материалов;
* Форматы даты, времени, чисел, валюты, меры весов, расстояния, и т.д.

#### **A/B тестирование**

Метод маркетингового исследования, суть которого заключается в том, что контрольная группа элементов сравнивается с набором тестовых групп, в которых один или несколько показателей были изменены, для того, чтобы выяснить, какие из изменений улучшают целевой показатель.

#### **Тестирование на отказ и восстановление системы**

Тестирование, которое проверяет продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками программного обеспечения, отказами оборудования или проблемами связи. Целью данного вида тестирования является проверка систем восстановления, которые, в случае возникновения сбоев, обеспечат сохранность и целостность данных тестируемого продукта.

#### **Тестирование на соответствие стандартам**

Процесс тестирования для определения соответствия компонента или системы стандартам, нормам и правилам.

**Примеры:**

* ISO/IEC 25010:2011
* Apple guidelines
* Google Play guidelines

#### **Тестирование мобильного ПО на прерывание работы**

Тестирование мобильного программного обеспечения на корректность обработки различных прерываний в работе приложения.

**Примеры:**

* Входящие звонки;
* СМС сообщения;
* Системные сообщения (напр. низкий уровень заряда батареи)
* Screen-lock;
* Уход в спящий режим.

#### **Тестирование мобильного ПО на соединение с Интернет**

Тестирование мобильного программного обеспечения на корректность работы на различных типах соединений.

**Примеры:**

* 2G;
* 3G;
* 4G;
* Wi-Fi;
* EDGE;
* Отсутствие Интернет.

## Лекция №8 Автоматизация тестирования

### **Введение**

**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.

**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.

**Инструмент для автоматизированного тестирования** (Automation Test Tool) - это программное обеспечение, посредством которого специалист по автоматизированному тестированию осуществляет создание, отладку, выполнение и анализ результатов прогона тест скриптов.

**Тест Скрипт**(Test Script) - это набор инструкций, для автоматической проверки определенной части программного обеспечения.

**Тестовый набор** (Test Suite) - это комбинация тест скриптов, для проверки определенной части программного обеспечения, объединенной общей функциональностью или целями, преследуемыми запуском данного набора.

**Тесты для запуска**(Test Run) - это комбинация тест скриптов или тестовых наборов для последующего совместного запуска (последовательного или параллельного, в зависимости от преследуемых целей и возможностей инструмента для автоматизированного тестирования).

**Уровни автоматизации тестирования**:

* Уровень модульного тестирования (unit tests);
* Уровень функционального тестирование (non-UI tests);
* Уровень тестирования через пользовательский интерфейс (UI tests).

Начало формы

Модульное тестирование

Конец формы

### **Модульное тестирование**

Модульное тестирование

Модульное тестирование, иногда блочное тестирование или юнит-тестирование (англ. unit testing) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы, наборы из одного или более программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Этот тип тестирования обычно выполняется программистами.

#### **Цели модульного тестирования**

Цель модульного тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

**Поощрение изменений**

Модульное тестирование позже позволяет программистам проводить рефакторинг, будучи уверенными, что модуль по-прежнему работает корректно (регрессионное тестирование). Это поощряет программистов к изменениям кода, поскольку достаточно легко проверить, что код работает и после изменений.

**Упрощение интеграции**

Модульное тестирование помогает устранить сомнения по поводу отдельных модулей и может быть использовано для подхода к тестированию «снизу вверх»: сначала тестируя отдельные части программы, а затем программу в целом.

**Документирование кода**

Модульные тесты можно рассматривать как «живой документ» для тестируемого класса. Клиенты, которые не знают, как использовать данный класс, могут использовать юнит-тест в качестве примера.

**Отделение интерфейса от реализации**

Поскольку некоторые классы могут использовать другие классы, тестирование отдельного класса часто распространяется на связанные с ним. Например, класс пользуется базой данных; в ходе написания теста программист обнаруживает, что тесту приходится взаимодействовать с базой. Это ошибка, поскольку тест не должен выходить за границу класса. В результате разработчик абстрагируется от соединения с базой данных и реализует этот интерфейс, используя свой собственный mock-объект. Это приводит к менее связанному коду, минимизируя зависимости в системе.

#### **Когда модульное тестирование не работает**

**Сложный код**

Тестирование программного обеспечения — комбинаторная задача. Например, каждое возможное значение булевской переменной потребует двух тестов: один на вариант TRUE, другой — на вариант FALSE. В результате на каждую строку исходного кода потребуется 3−5 строк тестового кода.

Как и любая технология тестирования, модульное тестирование не позволяет отловить все ошибки программы. В самом деле, это следует из практической невозможности трассировки всех возможных путей выполнения программы, за исключением простейших случаев.

**Результат известен лишь приблизительно**

Например, в математическом моделировании. Бизнес-приложения зачастую работают с конечными и счётными множествами, научные — с континуальными. Поэтому сложно подобрать тесты для каждой из ветвей программы, сложно сказать, верен ли результат, выдерживается ли точность, и т. д. А во многих случаях качество моделирования определяется «на глаз», и последний результат записывается как «опорный». Если найдено расхождение, новый результат проверяют вручную и выясняют, какой качественнее: старый или новый.

**Код, взаимодействующий с системой**

Код, взаимодействующий с портами, таймерами и прочими «нестабильными» частями системы, крайне сложно проверить в изолированном окружении.

Но это не значит, что модульное тестирование здесь полностью непригодно: оно вынуждает программиста перейти от файлов и портов, например, на абстрактные потоки. Это делает код более общим (например, без проблем можно перейти с файлов на сетевые сокеты), более тестируемым (можно даже проверить ситуацию «пропала связь»), ограничивает те части, которые не подлежат модульному тестированию.

**Ошибки интеграции и производительности**

При выполнении юнит-тестов происходит тестирование каждого из модулей по отдельности. Это означает, что ошибки интеграции, системного уровня, функций, исполняемых в нескольких модулях, не будут определены. Кроме того, данная технология бесполезна для проведения тестов на производительность. Таким образом, модульное тестирование более эффективно при использовании в сочетании с другими методиками тестирования.

**При общей низкой культуре программирования**

Для получения выгоды от модульного тестирования требуется строго следовать технологии тестирования на всём протяжении процесса разработки программного обеспечения. Нужно хранить не только записи обо всех проведённых тестах, но и обо всех изменениях исходного кода во всех модулях. С этой целью следует использовать систему контроля версий ПО. Таким образом, если более поздняя версия ПО не проходит тест, который был успешно пройден ранее, будет несложным сверить варианты исходного кода и устранить ошибку. Также необходимо убедиться в неизменном отслеживании и анализе неудачных тестов. Игнорирование этого требования приведёт к лавинообразному увеличению неудачных тестовых результатов.

**Проблемы с объектами-заглушками**

За исключением простейших случаев, тестируемый объект должен взаимодействовать с другими объектами. Этих «товарищей по взаимодействию» — объекты-заглушки — делают предельно простыми: либо крайне упрощёнными (память вместо БД), либо рассчитанными на конкретный тест и механически повторяющими сессию обмена. Вопросы начинаются, когда протокол обмена меняется; надо отыскивать эти заглушки во всех тестах и переводить под новый протокол.

#### **Правила написания модульных тестов**

* Тесты должны быть:
  + достоверными;
  + не зависеть от окружения, на котором они выполняются;
  + легко поддерживаться;
  + легко читаться и быть простыми для понимания (даже новый разработчик должен понять что именно тестируется);
  + соблюдать единую конвенцию именования;
  + запускаться регулярно в автоматическом режиме.
* Один тест должен проверять только одну сущность;
* Тесты должны храниться в системе контроля версий;
* Названия методов должны быть «говорящими»

    [Тестируемый метод]\_[Сценарий]\_[Ожидаемое поведение]

class CalculatorTests

{

public void Sum\_2Plus5\_7Returned()

        {

    // …

         }

}

* Необходимо придерживаться единого стиля написания тела теста

class CalculatorTests

{

public void Sum\_2Plus5\_7Returned()

{

// arrange

var calc = new Calculator();

// act

var res = calc.Sum(2,5);

// assert

Assert.AreEqual(7, res);

}

}

### **Интеграционные автоматизированные тесты (Non-UI tests)**

Процесс в программировании, позволяющий проверить работоспособность приложения используя программный интерфейс приложения (API).

**Пример теста:**

package CountriesRestTests;

…

public class GetTest {

 @Test

public void getRequestFindCapital() throws JSONException {

 // выполняем запрос get для доступа ко всем параметрам ответа

Response resp = get("http://restcountries.eu/rest/v1/name/belarus");

 JSONArray jsonResponse = new JSONArray(resp.asString());

 // получение параметра capital (столицы Беларуси)

String capital = jsonResponse.getJSONObject(0).getString("capital");

 // проверка, что столицей является Минск

AssertJUnit.assertEquals(capital, “Minsk");

}

}

**Инструменты для автоматизации тестирования API:**

Для автоматизации тестирования API необходим фреймворк, который предоставит наборы правил и рекомендаций для написания тестов, а также позволит интегрироваться с другими необходимыми фреймворками.

На данный момент существует множество фреймворков, имеющие возможности из перечисленных выше. Чаще фреймворк выбирается исходя из технологии, с помощью которой создано приложение (TestNG, JUNit, Mocha, PyTest, и др.)

Большинство из этих библиотек позволяют отправлять и получать данные в формате JSON, XML, CSV, Text, Image, form-data, encoded-data с несколькими поддерживаемыми стандартами авторизации.

Они также позволяют обрабатывать код статуса ответа HTTP и проверять, пришел ли требуемый код статуса ответа или нет. В дополнение к этому, тестовые фреймворки позволят вам использовать все возможности языка, на котором они основаны:

* База данных: можно легко создать соединения БД для чтения данных.
* Внешний файл: можно прочитать внешний текст, JSON, CSV или любые другие файлы.
* Случайные данные: можно использовать библиотеки, такие как faker, для генерации случайных тестовых данных на лету.
* Данные из ответа API : во время тестирования бывает необходимо передать ответ одного API в качестве данных запроса другому.  ️

Для проверки правильности ответов необходима библиотека под названием Assertion library. Многие тестовые среды поставляются в комплекте с библиотеками assertion, что дает возможность писать assert-ы на простом английском языке, таком как синтаксис. Они также позволяют проверить JSON-схему ответа.

Также большинство этих фреймворков предоставляют базовый HTML-отчет о тестовом прогоне, который можно скачать и отправить всем заинтересованным лицам.

### **UI тестирование**

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

GUI-автоматизация развивалась в течение 4 поколений инструментов и техник:

1. **Утилиты записи и воспроизведения** (англ. capture/playback tools) записывают действия тестировщика во время ручного тестирования. Они позволяют выполнять тесты без прямого участия человека в течение продолжительного времени, значительно увеличивая продуктивность и устраняя «тупое» повторение однообразных действий во время ручного тестирования. В то же время, любое малое изменение тестируемого ПО требует перезаписи ручных тестов. Поэтому это первое поколение инструментов не эффективно и не масштабируемо.

2. **Написание сценария** (англ. scripting) — форма программирования на языках, специально разработанных для автоматизации тестирования ПО — смягчает многие проблемы инструментов записи и воспроизведения. Но разработкой занимаются программисты высокого уровня, которые работают отдельно от тестировщиков, непосредственно запускающих тесты. К тому же скрипты более всего подходят для тестирования GUI и не могут быть внедренными, пакетными или вообще каким-либо образом объединены в систему. Наконец, изменения в тестируемом ПО требуют сложных изменений в соответствующих скриптах, и поддержка все возрастающей библиотеки тестирующих скриптов становится в конце концов непреодолимой задачей.

3. **Управляемое данными тестирование** (англ. Data-driven testing) — методология, которая используется в автоматизации тестирования. Особенностью является то, что тестовые скрипты выполняются и верифицируются на основе данных, которые хранятся в центральном хранилище данных или базе данных. Роль базы данных могут выполнять ODBC-ресурсы, csv или xls файлы и т. д. Управляемое данными тестирование — это объединение нескольких взаимодействующих тестовых скриптов и их источников данных во фреймворк, используемый в методологии. В этом фреймворке переменные используются как для входных значений, так и для выходных проверочных значений: в тестовом скрипте обычно закодированы навигация по приложению, чтение источников данных, ведение логов тестирования. Таким образом, логика, которая будет выполнена в скрипте, также зависит от данных.

4. **Тестирование по ключевым словам** (англ. Keyword-based) автоматизация подразумевает разделение процесса создания кейсов на 2 этапа: этап планирования и этап реализации. В этом случае конечный тест представляет собой не программный код, а описание последовательности действий с их параметрами (например, «завести в базе данных пользователя с логином XXX и паролем YYY»). При этом фреймворк отвечает за непосредственную реализацию ключевых слов (действий), а дизайнеру тестов достаточно иметь представление о всём наборе действий, реализованных во фреймворке. Это даёт возможность создавать тесты людям, не имеющим навыков программирования.

Также сейчас широко используется такой подход как **Behavior-driven development**.  Это методология разработки программного обеспечения, являющаяся ответвлением от методологии разработки через тестирование (TDD).

Основной идеей данной методологии является совмещение в процессе разработки чисто технических интересов и интересов бизнеса, позволяя тем самым управляющему персоналу и программистам говорить на одном языке. Для общения между этими группами персонала используется предметно-ориентированный язык, основу которого представляют конструкции из естественного языка, понятные неспециалисту, обычно выражающие поведение программного продукта и ожидаемые результаты.

Считается, что данный подход эффективен, когда предметная область, в которой работает программный продукт, описывается очень комплексно.

BDD фокусируется на следующих вопросах:

* С чего начинается процесс.
* Что нужно тестировать, а что нет.
* Сколько проверок должно быть совершено за один раз.
* Что можно назвать проверкой.
* Как понять, почему тест не прошёл.

Исходя из этих вопросов, BDD требует, чтобы имена тестов были целыми предложениями, которые начинаются с глагола в сослагательном наклонении и следовали бизнес целям. Описание приемочных тестов должно вестись на гибком языке пользовательской истории, например,

Как [*роль того, чьи бизнес интересы удовлетворяютс*я] я хочу, чтобы [*описание функциональности так, как она должна работать*], для того чтобы [*описание выгоды*].

Критерии приёмки должны быть описаны через сценарий, который реализует пользователь, чтобы достигнуть результата.

**Принципы BDD**

Как уже было отмечено, тесты для некоторой единицы программного обеспечения должны быть описаны с точки зрения желаемого поведения программируемого устройства. Под желаемым поведением здесь понимается такое, которое имеет ценность для бизнеса. Описание желаемого поведения даётся с помощью спецификации поведения (англ. behavioral specification).

Спецификация поведения строится в полуформальной форме. В настоящее время в практике BDD устоялась следующая структура:

* Заголовок (англ. Title). В сослагательной форме должно быть дано описание бизнес-цели.
* Описание (англ. Narrative). В краткой и свободной форме должны быть раскрыты следующие вопросы:
* Кто является заинтересованным лицом данной истории;
* Что входит в состав данной истории;
* Какую ценность данная история предоставляет для бизнеса.
* Сценарии (англ. Scenarios). В одной спецификации может быть один и более сценариев, каждый из которых раскрывает одну из ситуаций поведения пользователя, тем самым конкретизируя описание спецификации. Каждый сценарий обычно строится по одной и той же схеме:
* Начальные условия (одно или несколько);
* Событие, которое инициирует начало этого сценария;
* Ожидаемый результат или результаты.

BDD не предоставляет каких-либо формальных правил, но настаивает на том, чтобы использовался ограниченный стандартный набор фраз, который включал бы все элементы спецификации поведения. В 2007 году Дэном Нортом был предложен шаблон для спецификации, который получил популярность и впоследствии стал известен как язык Gherkin.