**1. Тестирование, качество, баг.**

**Тести́рование програ́ммного обеспе́че́ния** — процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определенным образом.

**Качество -** свойство, которое показывает, что продукт удовлетворяет требованиям либо превосходит их.

**Баг** - ошибка, дефект, сбой или ошибка в компьютерной программе или системе, которая создает неверный или неожиданный результат или приводит к неправильной работе. Компоненты бага:

1.Заголовок.

2.Описание.

3. Предусловия.

4. Шаги для выполнения.

5. Результат теста.

6. Ожидаемый результат.

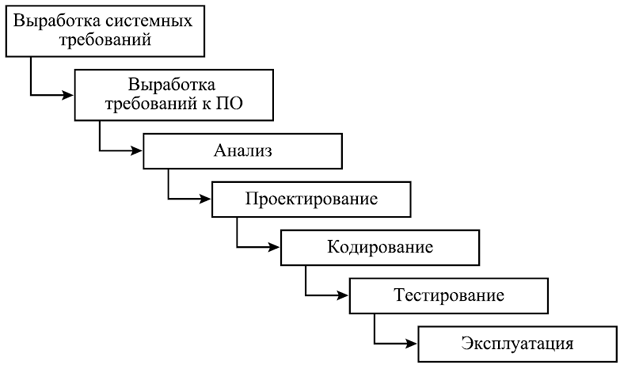
7. Вложения.

**Тест кейс** - конкретный тестовый сценарий, который содержит совокупность шагов, условий и ожидаемых результатов. Компоненты тест кейса:

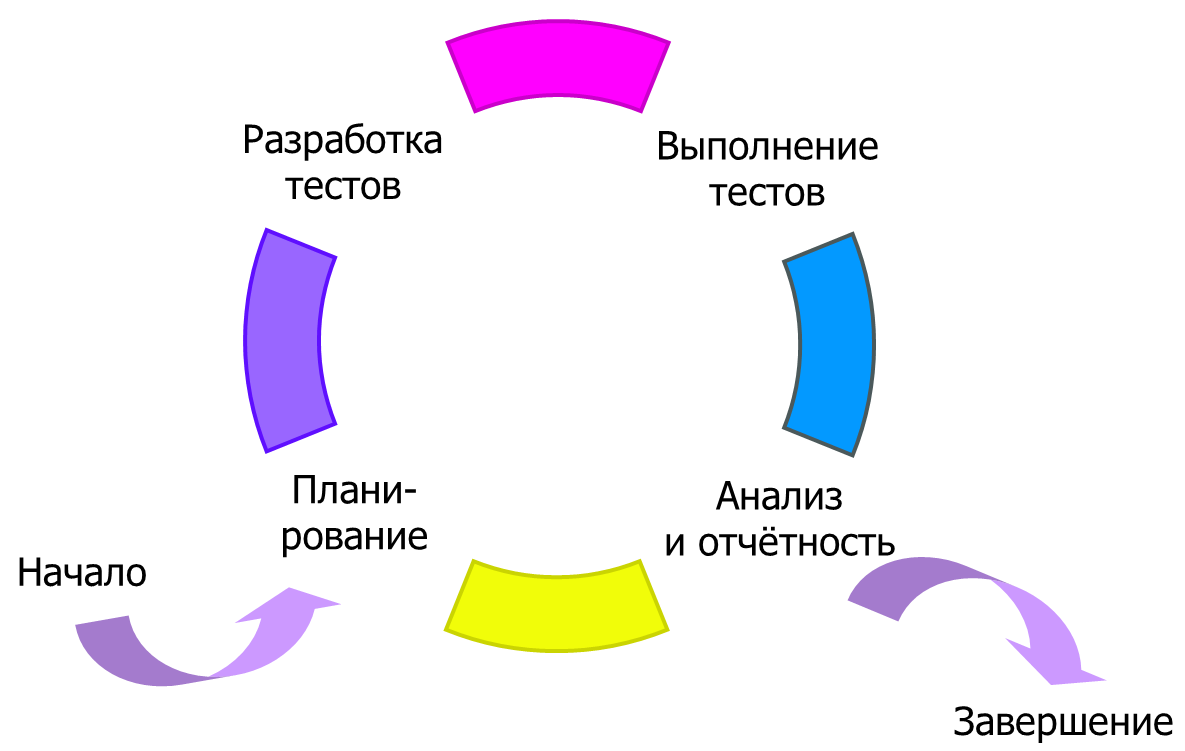
* Number
* Module/Functionality
* Preconditions
* **Summary/Description**
* **Steps**
* **Expected results**
* Actual results

**2. Жизненный цикл ПО, тестирования**

**Жизненный цикл программного обеспечения** (ПО) — период времени, который начинается с момента принятия решения о необходимости создания программного продукта и заканчивается в момент его полного изъятия из эксплуатации.



Жизненный цикл тестирования:



**3.Виды и типы тестирования.**

Функциональное тестирование (functional testing)

Проверка функциональности (тестирование методом «черного ящика») – проверка соответствия программного обеспечения требованиям, заявленным в спецификации. Может проводиться как полное тестирование заявленной функциональность, так и проверка только базовой функциональности.

**Тестирование «белого ящика» (white box)** Тестирование на соответствие программного продукта требованиям со знанием внутренней структуры реализации системы (есть в наличии исходный код и технические спецификации).

**Тестирование «черного ящика» (black box)** Тестирование на соответствие программного продукта требованиям без знания внутренней структуры реализации системы.

Системное тестирование (system testing)

Высокоуровневая проверка функционала всей программы или системы в целом.

Тестирование проиводительности (performance testing)

Тестирование, которое проводится с целью определения, как быстро работает система или её часть под определённой нагрузкой.

* **Нагрузочное тестирование (load testing)**

Тестирование предназначено для проверки работоспособности системы при стандартных нагрузках и для определения максимально возможного пика, при котором система работает правильно.

* **Стресс тестирование (stress testing)**

Тестирование предназначено для проверки работоспособности системы при *не*стандартных нагрузках и для определения максимально возможного пика, при котором система работает правильно. Так же предназначено для выявления результатов, при которых система переходит в нерабочее состояние.

Регрессионное тестирование (regression testing)

Регрессионное тестирование проводится с целью проверить, не влияют ли новые функции, улучшения и исправленные дефекты на существующую функциональность продукта и не возникают ли старые дефекты.

Модульное тестирование (unit testing)

Каждая сложная программная система состоит из отдельных частей - модулей, выполняющих ту или иную функцию в составе системы. Для того, чтобы удостовериться в корректной работе всей системы, необходимо вначале протестировать каждый модуль системы по отдельности. В случае возникновения проблем при тестировании системы в целом это позволяет проще выявить модули, вызвавшие проблему, и устранить соответствующие дефекты в них. Такое тестирование модулей по отдельности получило называние модульного тестирования.

Тестирование безопасности (security testing)

Тестирование безопасности - это стратегия тестирования, используемая для проверки безопасности системы, а также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения, атак хакеров, вирусов, несанкционированного доступа к конфиденциальным данным.

Тестирование локализации (localization testing)

Тестирование локализации - это процесс тестирования локализованной версии программного продукта. Проверка правильности перевода элементов интерфейса пользователя, проверка правильности перевода системных сообщений и ошибок, проверка перевода раздела "Помощь"/"Справка" и сопроводительной документации.

Юзабилити тестирование (usability testing)

Тестирование удобства пользования - это метод тестирования, направленный на установление степени удобства использования, обучаемости, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий. Выявлять проблемы, связанные со специфическим механизмом интерфейса определять, существуют ли проблемы с удобностью интерфейса для навигации, использования основного функционала.

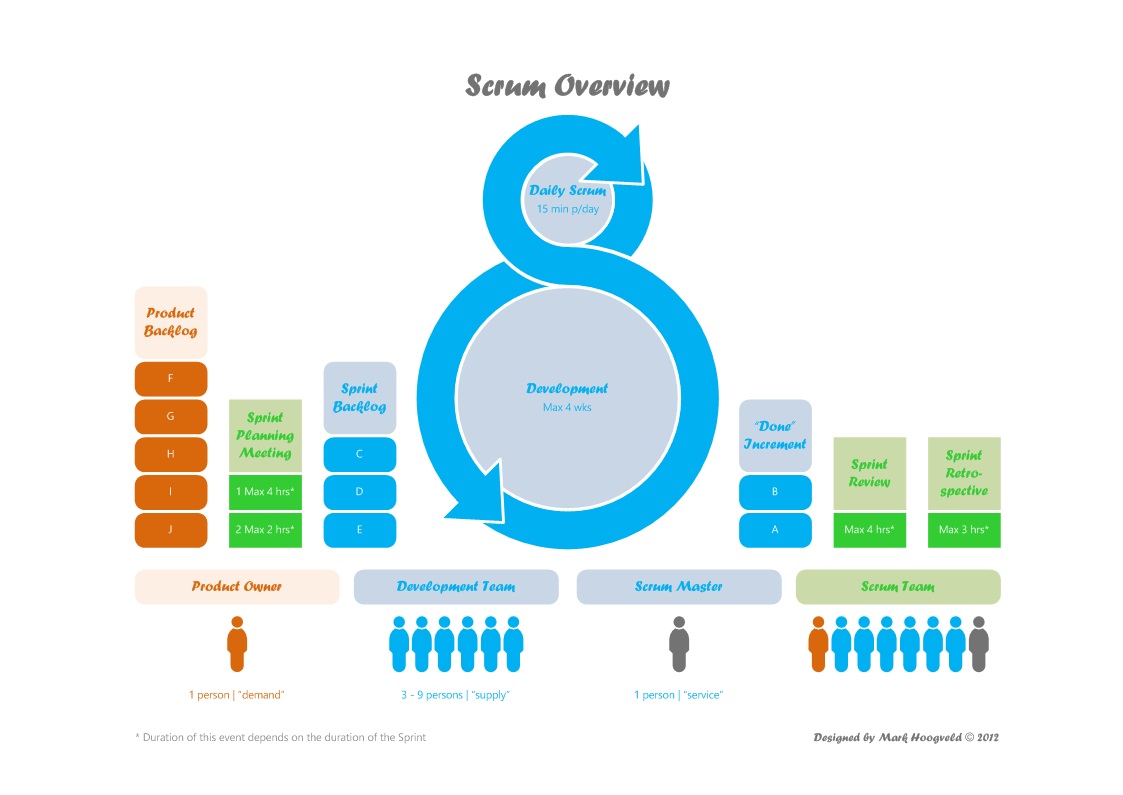
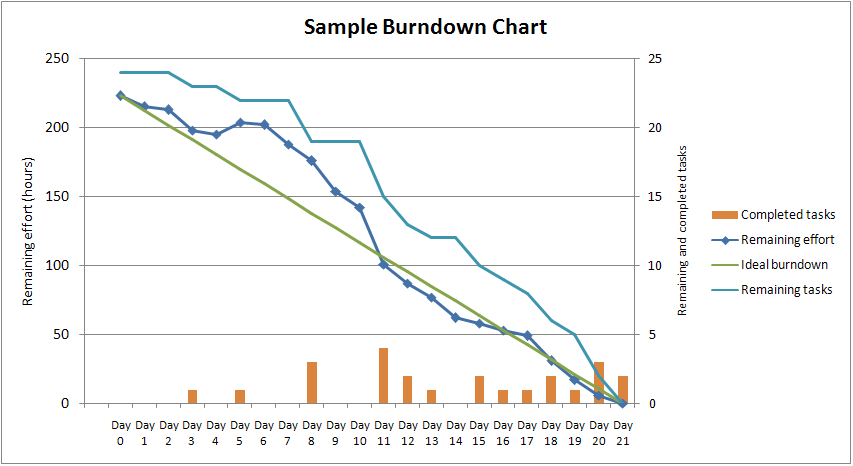
Приемочный тест (Smoke test) – первый и самый короткий тест, призванный проводить проверку основных элементов программного продукта и его работоспособности в целом. В случае функционального тестирования – проверяется основной функционал приложения. Тест занимает 1-4 часа в зависимости от сложности тестируемого продукта. На основе результатов данного теста принимается решение о приемке версии программного продукта и продолжении тестирования текущей версии продукта более серьезными тестовыми испытаниями.

Критический тест (Critical path test) – основной тип тестовых испытаний, во время которого значимые элементы и функции приложения проверяются на предмет правильности работы при стандартном их использовании. Как правило, на данном уровне тестирования проверяется основная масса требований к продукту.

Расширенный тест (Extended test) – вид углубленного тестирования, при котором проверяется нестандартное использование программного продукта, границы переполнения массивов данных, ввод специальных символов и т.п.

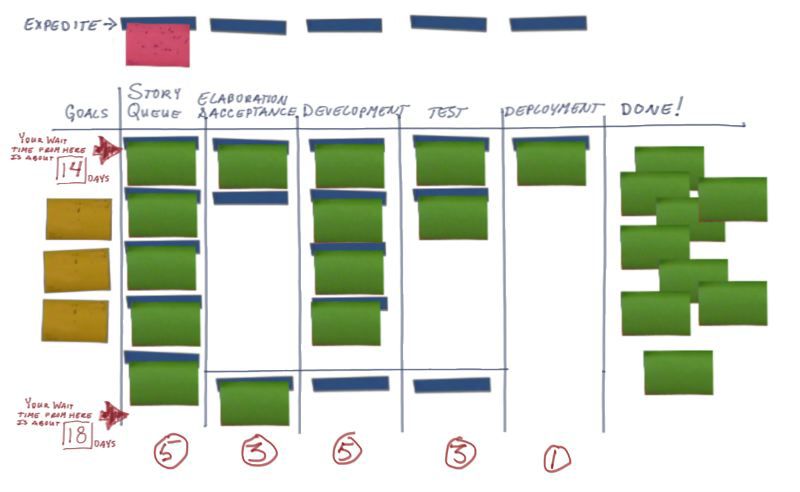
**4.Методологии тестирования.**

## SCRUM

[](http://calvinx.com/wp-content/uploads/2014/05/scrum-overview-mark-hoogveld.jpg)  
**Scrum (Скрам)** — это не аббревиатура, этот термин взят из регби, который обозначает схватку вокруг мяча.  
Сам термин Scrum, я бы определила так — это методология управления проектами, которая построена на принципах тайм-менеджмета. Основной ее особенностью является вовлеченность в процесс всех участников, причем у каждого участника есть своя определенная роль. Суть в том, что не только команда работает над решением задачи, но все те, кому интересно решение задачи, не просто поставили ее и расслабились, а постоянно «работают» с командой, и эта работа не означает только постоянный контроль.  
Основные термины, которые используются в методологии:  
**Владелец продукта (Product owner)** — человек, который имеет непосредственный интерес в качественном конечном продукте, он понимает, как это продукт должен выглядеть/работать. Этот человек не работает в команде, он работает на стороне заказчика/клиента (это может быть как другая компания, так и другой отдел), но этот человек работает с командой. И это тот человек, который расставляет приоритеты для задач.  
**Scrum-мастер** — это человек, которого можно назвать руководителем проекта, хотя это не совсем так. Главное, что это человек, «зараженный Scrum-бациллой» на столько, что несет ее как своей команде, так и заказчику, и соответственно следит за тем, чтобы все принципы Scrum соблюдались.  
**Scrum-команда** — это команда, которая принимает все принципы Scrum и готова с ними работать.  
**Спринт** — отрезок времени, который берется для выполнения определенного (ограниченного) списка задач. Рекомендуется брать 2-4 недели (длительность определяется командой один раз).  
**Бэклог (backlog)** — это список всех работ. Можно сказать, что это ежедневник общего пользования   
Различают 2 вида бэклогов: Product-бэклог и спринт-бэклог.  
**Product-бэклог** — это полный список всех работ, при реализации которых мы получим конечный продукт.  
**Спринт-бэклог** — это список работ, который определила команда и согласовала с Владельцем продукта, на ближайший отчетный период (спринт). Задания в спринт-бэклог берутся из product-бэклога.  
**Планирование спринта** — это совещание, на котором присутствуют все (команда, Scrum-мастер, Владелец продукта). В течение этого совещания Владелец продукта определяет приоритеты заданий, которые он хотел бы увидеть выполнеными по истечении спринта. Команда оценивает по времени, сколько из желаемого они могут выполнить. В итоге получается список заданий, который не может меняться в течение спринта и к концу спринта должен быть полностью выполнен.  
  
Пример работы PR-агентства. Как бы это могло выглядеть, если бы они работали по Scrum.  
Компания клиент «Икс» хочет провести через 2 месяца масштабное мероприятие для своих партнеров и журналистов. Услуги по организации такого мероприятия компания «Икс» заказала у агентства «Зет». Компанию «Икс» представляет PR-менеджер, который отвечает за организацию мероприятия со стороны клиента. В терминологии Scrum — этот человек называется Владелец продукта. Со стороны агентства за организацию мероприятия отвечает account-менеджер (Scrum-мастер), в подчинении которого находится команда (Scrum-команда). На совместном совещании (планировании спринта) компания и агентство решают, что они будут отчитываться-планировать каждые 2 недели (длина спринта). На первые 2 недели они запланировали список задач (спринт-бэклог), однако команда оценила, что не все из этого списка они успеют выполнить. Тогда PR-менеджер (он же Владелец продукта), говорит какие из этого списка задач более приоритетные на ближайшие 2 недели, после чего команда берется за выполнение заданий. Единственное что здесь должно быть учтено, что на момент планирования первого спринта должен быть спланирован весь список заданий на 2 месяца (product-бэклог), чтобы не получилось так, что к моменту проведения мероприятия что-то не выполнено.  
  
**Жизненный цикл спринта**  
  
**Планирование спринта**  
В начале каждого спринта проводится планирование спринта. В планировании спринта участвуют заказчики, пользователи, менеджмент, Product Owner, Скрам Мастер и команда.  
Планирование спринта состоит из двух последовательных митингов.  
Планирование спринта, митинг первый  
Участники: команда, Product Owner, Scrum Master, пользователи, менеджемент  
Цель: Определить цель спринта (Sprint Goal) и Sprint Backlog -функциональность, которая будет разработана в течение следующего спринта для достижения цели спринта.  
Артефакт: Sprint Backlog  
Планирование спринта, митинг второй  
Участники: Скрам Мастер, команда  
Цель: определить, как именно будет разрабатываться определенная функциональность для того, чтобы достичь цели спринта. Для каждого элемента Sprint Backlog определяется список задач и оценивается их продолжительность.  
Артефакт: в Sprint Backlog появляются задачи  
Если в ходе спринта выясняется, что команда не может успеть сделать запланированное на спринт, то Скрам Мастер, Product Owner и команда встречаются и выясняют, как можно сократить scope работ и при этом достичь цели спринта.  
**Остановка спринта (Sprint Abnormal Termination)**  
Остановка спринта производится в исключительных ситуациях. Спринт может быть остановлен до того, как закончатся отведенные 30 дней. Спринт может остановить команда, если понимает, что не может достичь цели спринта в отведенное время. Спринт может остановить Product Owner, если необходимость в достижении цели спринта исчезла.  
После остановки спринта проводится митинг с командой, где обсуждаются причины остановки спринта. После этого начинается новый спринт: производится его планирование и стартуются работы.  
**Daily Scrum Meeting**  
Этот митинг проходит каждое утро в начале дня. Он предназначен для того, чтобы все члены команды знали, кто и чем занимается в проекте. Длительность этого митинга строго ограничена и не должна превышать 15 минут. Цель митинга — поделиться информацией. Он не предназначен для решения проблем в проекте. Все требующие специального обсуждения вопросы должны быть вынесены за пределы митинга.  
Скрам митинг проводит Скрам Мастер. Он по кругу задает вопросы каждому члену команды  
• Что сделано вчера?  
• Что будет сделано сегодня?  
• С какими проблемами столкнулся?  
Скрам Мастер собирает все открытые для обсуждения вопросы в виде Action Items, например в формате что/кто/когда, например  
• Обсудить проблему с отрисовкой контрола  
• Петя и Вася  
• Сразу после скрама  
**Диаграмма сгорания задач (Burndown chart)**  
[](https://upload.wikimedia.org/wikipedia/commons/0/05/SampleBurndownChart.png)

Диаграмма, показывающая количество сделанной и оставшейся работы. Обновляется ежедневно с тем, чтобы в простой форме показать подвижки в работе над спринтом. График должен быть общедоступен.  
Существуют разные виды диаграммы:  
• диаграмма сгорания работ для спринта — показывает, сколько уже задач сделано и сколько ещё остаётся сделать в текущем спринте.  
• диаграмма сгорания работ для выпуска проекта — показывает, сколько уже задач сделано и сколько ещё остаётся сделать до выпуска продукта (обычно строится на базе нескольких спринтов).  
Ретроспектива   
В конце каждого Спринта, Скрам Команда собирается на **Ретроспективу**. Цель Ретроспективы пересмотреть качество существующих процессов, взаимоотношения людей и применяемые инструменты. Команда определяет, что прошло хорошо, а что не очень, а также выявляет потенциальные возможности для улучшений. Они создают план улучшений на будущее.

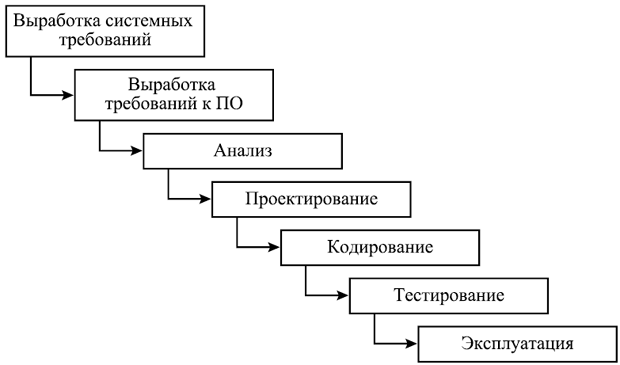
## Канбан

Термин Канбан имеет дословный перевод: «Кан» значит видимый, визуальный, и «бан» значит карточка или доска.  
На заводах Тойота карточки Канбан используются повсеместно для того, чтобы не загромождать склады и рабочие места заранее созданными запчастями. Например, представьте, что вы ставите двери на Тойоты Короллы. У вас около рабочего места находится пачка из 10 дверей. Вы их ставите одну за другой на новые машины и, когда в пачке остается 5 дверей, то вы знаете, что пора заказать новые двери. Вы берете карточку Канбан, пишете на ней заказ на 10 дверей и относите ее тому, кто делает двери. Вы знаете, что он их сделает как раз к тому моменту, как у вас закончатся оставшиеся 5 дверей. И именно так и происходит — когда вы ставите последнюю дверь, прибывает пачка из 10 новых дверей. И так постоянно — вы заказываете новые двери только тогда, когда они вам нужны.  
А теперь представьте, что такая система действует на всём заводе. Нигде нет складов, где запчасти лежат неделями и месяцами. Все работают только по запросу и производят именно столько запчастей, сколько запрошено. Если вдруг заказов стало больше или меньше — система сама легко подстраивается под изменения.  
  
Основная задача карт Канбан в этой системе — это уменьшать количество «выполняющейся в данный момент работы» (work in progress).  
Например, на всю производственную линию может быть выделено ровно 10 карточек для дверей. Это значит, что в каждый момент времени на линии не будет больше 10 готовых дверей. Когда заказывать новые двери и сколько — это задача для того, кто их устанавливает. Только он знает свои потребности, и только он может помещать заказы производителю дверей, но он всегда ограничен числом 10.  
Этот метод Бережливого производства (Lean manufacturing) был придуман в Тойоте и сейчас многие производственные компании по всему миру его внедряют или уже внедрили.  
  
Но это всё относится к производству, а не к разработке программного обеспечения.  
А что же такое Канбан разработка применительно к ПО, и чем она отличается от других гибких методологий, буть то SCRUM или XP?  
  
Во-первых, нужно сразу понять, что Канбан — это не конкретный процесс, а система ценностей. Как, впрочем, и SCRUM с XP. Это значит, что никто вам не скажет что и как делать по шагам.  
Во-вторых, весь Канбан можно описать одной простой фразой — «Уменьшение выполняющейся в данный момент работы (work in progress)».  
В-третьих, Канбан — это даже еще более «гибкая» методология, чем SCRUM и XP. Это значит, что она не подойдет всем командам и для всех проектов. И это также значит, что команда должна быть еще более готовой к гибкой работе, чем даже команды, использующие SCRUM и XP.  
  
**Разница между Канбан и SCRUM:**  
— В Канбан нет таймбоксов ни на что (ни на задачи, ни на спринты)  
— В Канбан задачи больше и их меньше  
— В Канбан оценки сроков на задачу опциональные или вообще их нет  
— В Канбан «скорость работы команды» отсутствует и считается только среднее время на полную реализацию задачи  
Канбан разработка отличается от SCRUM в первую очередь ориентацией на задачи. Если в SCRUM основная ориентация команды — это успешное выполнение спринтов (надо признать, что это так), то в Канбан на первом месте задачи.  
Спринтов никаких нет, команда работает над задачей с самого начала и до завершения. Деплоймент задачи делается тогда, когда она готова. Презентация выполненной работы — тоже. Команда не должна оценивать время на выполнение задачи, ибо это имеет мало смысла и почти всегда ошибочно вначале.  
Если менеджер верит команде, то зачем иметь оценку времени? Задача менеджера — это создать приоритизированный пул задач, а задача команды — выполнить как можно больше задач из этого пула. Всё. Никакого контроля не нужно. Всё, что нужно от менеджера — это добавлять задачи в этот пул или менять им приоритет. Именно так он управляет проектом.  
Команда для работы использует Канбан-доску. Например, она может выглядеть так:  
[](http://habrastorage.org/files/2b5/ec1/918/2b5ec19187ef4c47821de19248fa63b1.jpg)

Столбцы слева направо:  
**Цели проекта:**  
Необязательный, но полезный столбец. Сюда можно поместить высокоуровневые цели проекта, чтобы команда их видела и все про них знали. Например, «Увеличить скорость работы на 20%» или «Добавить поддержку Windows 7».  
**Очередь задач:**  
Тут хранятся задачи, которые готовы к тому, чтобы начать их выполнять. Всегда для выполнения берется верхняя, самая приоритетная задача и ее карточка перемещается в следующий столбец.  
**Проработка дизайна:**  
этот и остальные столбцы до «Закончено» могут меняться, т.к. именно команда решает, какие шаги проходит задача до состояния «Закончено».  
Например, в этом столбце могут находиться задачи, для которых дизайн кода или интерфейса еще не ясен и обсуждается. Когда обсуждения закончены, задача передвигается в следующий столбец.  
**Разработка:**  
Тут задача висит до тех пор, пока разработка фичи не завершена. После завершения она передвигается в следующий столбец.  
Или, если архитектура не верна или не точна — задачу можно вернуть в предыдущий столбец.  
**Тестирование:**  
В этом столбце задача находится, пока она тестируется. Если найдены ошибки — возвращается в Разработку. Если нет — передвигается дальше.  
**Деплоймент:**  
У всех проектов свой деплоймент. У кого-то это значит выложить новую версию продукта на сервер, а у кого-то — просто закомитить код в репозиторий.  
**Закончено:**  
Сюда стикер попадает только тогда, когда все работы по задаче закончены полностью.

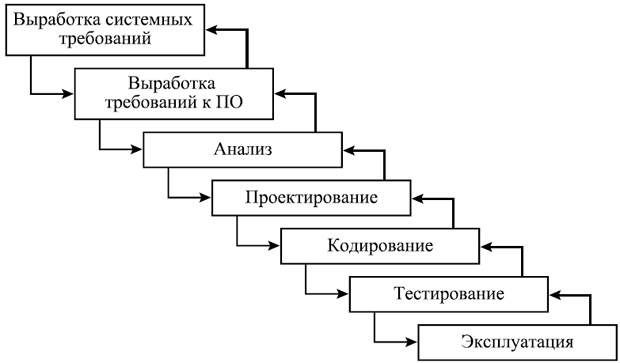
В любой работе случаются срочные задачи. Запланированные или нет, но такие, которые надо сделать прямо сейчас. Для таких можно выделить специальное место (на картинке отмечено, как «Expedite»). В Expedite можно поместить одну срочную задачу и команда должна начать ее выполнять немедленно и завершить как можно быстрее. Но может быть только одна такая задача! Если появляется еще одна — она должна быть добавлена в «Очередь задач».  
  
А теперь самое важное. Видите цифры под каждым столбцом? Это число задач, которые могут быть одновременно в этих столбцах. Цифры подбираются экспериментально, но считается, что они должны зависеть от числа разработчиков в команде.  
Например, если вы имеете 8 программистов в команде, то в строку «Разработка» вы можете поместить цифру 4. Это значит, что одновременно программисты будут делать не более 4-х задач, а значит у них будет много причин для общения и обмена опытом. Если вы поставите туда цифру 2, то 8 программистов, занимающихся двумя задачами, могут заскучать или терять слишком много времени на обсуждениях. Если поставить 8, то каждый будет заниматься своей задачей и некоторые задачи будут задерживаться на доске надолго, а ведь главная задача Канбан — это уменьшение времени прохождения задачи от начала до стадии готовности.  
Никто не даст точный ответ, какие должны быть эти лимиты, но попробуйте для начала разделить число разработчиков на 2 и посмотреть, как это работает в вашей команде. Потом эти числа можно подогнать под вашу команду.  
Под «разработчиками» я понимаю не только программистов, но и других специалистов. Например, для столбца «Тестирование» разработчики — это тестеры, т.к. тестирование — это их обязаность.

## Каскадная модель (waterfall)

[](http://fresh2l.com/sites/default/files/dev-cycle-waterfall.gif)

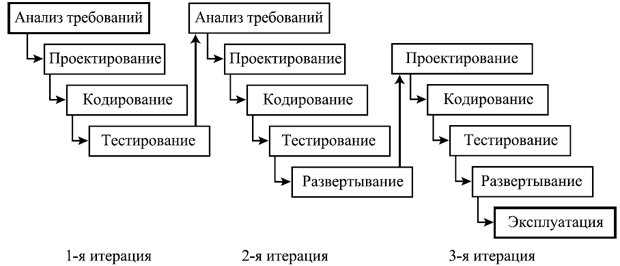
— высокий уровень формализации процессов;  
— большое количество документации;  
— жесткая последовательность этапов жизненного цикла без возможности возврата на предыдущий этап.  
Минусы  
• Watrefall проект должен постоянно иметь актуальную документацию. Обязательная актуализация проектной документации. Избыточная документация  
• Очень не гибкая методологии  
• Может создать ошибочное впечатление о работе над проектом (например фраза «45% выполнено» не несёт за собой никакой полезной информации, а является всего лишь инструментов для менеджера проекта)  
• У Заказчика нет возможности ознакомиться с системой заранее и даже с «Пилотом» системы  
• У Пользователя нет возможности привыкать к продукту постепенно  
• Все требования должны быть известны в начале жизненного цикла проекта  
• Возникает необходимость в жёстком управлении и регулярном контроле, иначе проект быстро выйдет из графиков  
• Отсутствует возможность учесть переделку, весь проект делается за один раз  
**Плюсы**  
• Высокая прозрачность разработки и фаз проекта  
• Чёткая последовательность  
• Стабильность требований  
• Строгий контроль менеджмента проекта  
• Облегчает работу по составлению плана проекта и сбора команды проекта  
• Хорошо определяет процедуру по контроля качества

## «Водоворот» или каскадная модель с промежуточным контролем

В этой модели предусмотрен промежуточный контроль за счет обратных связей. Но это достоинство порождает и недостатки. Затраты на реализацию проекта при таком подходе возрастают практически в 10 раз. Эта модель, как вы уже поняли, является незначительной модификацией предыдущей и относится к первой группе.  
[](http://fresh2l.com/sites/default/files/dev-cycle-whirlpool.gif)

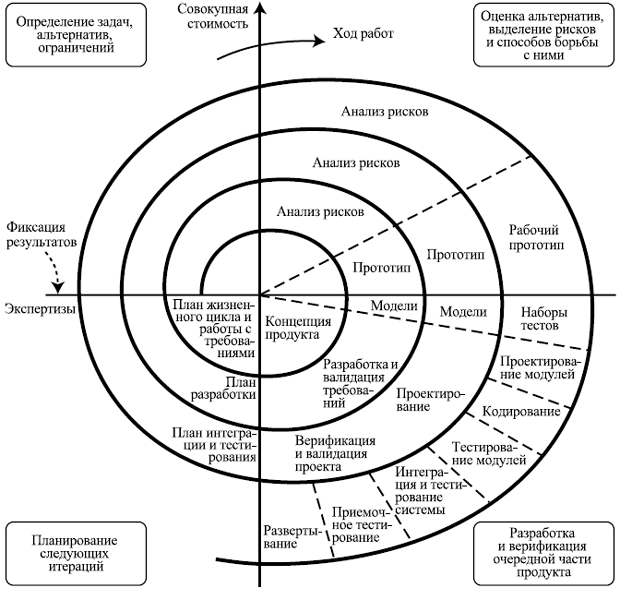
При реальной работе в соответствии с моделью, допускающей движение только в одну сторону, обычно возникают проблемы при обнаружении недоработок и ошибок, сделанных на ранних этапах. Но еще более тяжело иметь дело с изменениями окружения, в котором разрабатывается ПО (это могут быть изменения требований, смена подрядчиков, изменения политик разрабатывающей или эксплуатирующей организации, изменения отраслевых стандартов, появление конкурирующих продуктов и пр.).

## Итеративная модель

Итеративные или инкрементальные модели (известно несколько таких моделей) предполагают разбиение создаваемой системы на набор кусков, которые разрабатываются с помощью нескольких последовательных проходов всех работ или их части.  
[](http://fresh2l.com/sites/default/files/dev-cycle-iter.gif)

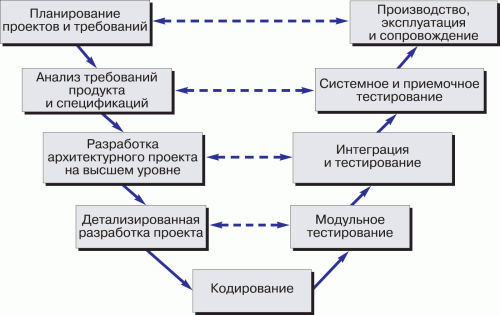
Каскадная модель с возможностью возвращения на предшествующий шаг при необходимости пересмотреть его результаты, становится итеративной.  
Итеративный процесс предполагает, что разные виды деятельности не привязаны намертво к определенным этапам разработки, а выполняются по мере необходимости, иногда повторяются, до тех пор, пока не будет получен нужный результат.  
Вместе с гибкостью и возможностью быстро реагировать на изменения, итеративные модели привносят дополнительные сложности в управление проектом и отслеживание его хода. При использовании итеративного подхода значительно сложнее становится адекватно оценить текущее состояние проекта и спланировать долгосрочное развитие событий, а также предсказать сроки и ресурсы, необходимые для обеспечения определенного качества результата.

## Спиральная модель жизненного цикла программного обеспечения

Данная модель прекрасно сочетает в себе постадийное прототипирование и проектирование. И из восходящей и нисходящей концепций в эту модель было взято все лучшее.  
[](http://fresh2l.com/sites/default/files/dev-cycle-spiral.gif)

**Преимущества модели:**  
1. Результат достигается в кратчайшие сроки.  
2. Конкурентоспособность достаточно высокая.  
3. При изменении требований, не придется начинать все с «нуля».  
Но у этой модели есть один существенный недостаток: невозможность регламентирования стадий выполнения.

## V модель — разработка через тестирование

Данная модель имеет более приближенный к современным методам алгоритм, однако все еще имеет ряд недостатков. Является одной из основных практик экстремального программирования и предполагает регулярное тестирование продукта во время разработки.  
[](http://fresh2l.com/sites/default/files/dev-cycle-v.png)

V-модель обеспечивает поддержку в планировании и реализации проекта. В ходе проекта ставятся следующие задачи:  
• Минимизация рисков: V-образная модель делает проект более прозрачным и повышает качество контроля проекта путём стандартизации промежуточных целей и описания соответствующих им результатов и ответственных лиц. Это позволяет выявлять отклонения в проекте и риски на ранних стадиях и улучшает качество управления проектов, уменьшая риски.  
• Повышение и гарантии качества: V-Model — стандартизованная модель разработки, что позволяет добиться от проекта результатов желаемого качества. Промежуточные результаты могут быть проверены на ранних стадиях. Универсальное документирование облегчает читаемость, понятность и проверяемость.  
• Уменьшение общей стоимости проекта: Ресурсы на разработку, производство, управление и поддержку могут быть заранее просчитаны и проконтролированы. Получаемые результаты также универсальны и легко прогнозируются. Это уменьшает затраты на последующие стадии и проекты.  
• Повышение качества коммуникации между участниками проекта: Универсальное описание всех элементов и условий облегчает взаимопонимание всех участников проекта. Таким образом, уменьшаются неточности в понимании между пользователем, покупателем, поставщиком и разработчиком.

## Модель на основе разработки прототипа

Данная модель основывается на разработке прототипов и прототипирования продукта и относится ко второй группе.  
Прототипирование используется на ранних стадиях жизненного цикла программного обеспечения:  
o Прояснить не ясные требования (прототип UI)  
o Выбрать одно из ряда концептуальных решений (реализация сцинариев)  
o Проанализировать осуществимость проекта  
Классификация протопипов:  
o Горизонтальные прототипы — моделирует исключительно UI не затрагивая логику обработки и базу данных.  
o Вертикальные прототипы — проверка архитектурных решений.  
o Одноразовые прототипы — для быстрой разработки.  
o Эволюционные прототипы — первое приближение эволюционной системы.

## Модель Хаоса

Вкратце Стратегия хаоса — это стратегия разработки программного обеспечения основанная на модели хаоса. Главное правило — это,всегда решать наиболее важную задачу первой.

## Экстремальное программирование (XP)

Двенадцать основных приёмов экстремального программирования (по первому изданию книги Extreme programming explained) могут быть объединены в четыре группы:  
• Короткий цикл обратной связи (Fine-scale feedback)  
• Разработка через тестирование (Test-driven development)  
• Игра в планирование (Planning game)  
• Заказчик всегда рядом (Whole team, Onsite customer)  
• Парное программирование (Pair programming)  
• Непрерывный, а не пакетный процесс  
• Непрерывная интеграция (Continuous integration)  
• Рефакторинг (Design improvement, Refactoring)  
• Частые небольшие релизы (Small releases)  
• Понимание, разделяемое всеми  
• Простота (Simple design)  
• Метафора системы (System metaphor)  
• Коллективное владение кодом (Collective code ownership) или выбранными шаблонами проектирования (Collective patterns ownership)  
• Стандарт кодирования (Coding standard or Coding conventions)  
• Социальная защищенность программиста (Programmer welfare):  
• 40-часовая рабочая неделя (Sustainable pace, Forty-hour week)

## RATIONAL UNIFIED PROCESS (RUP)

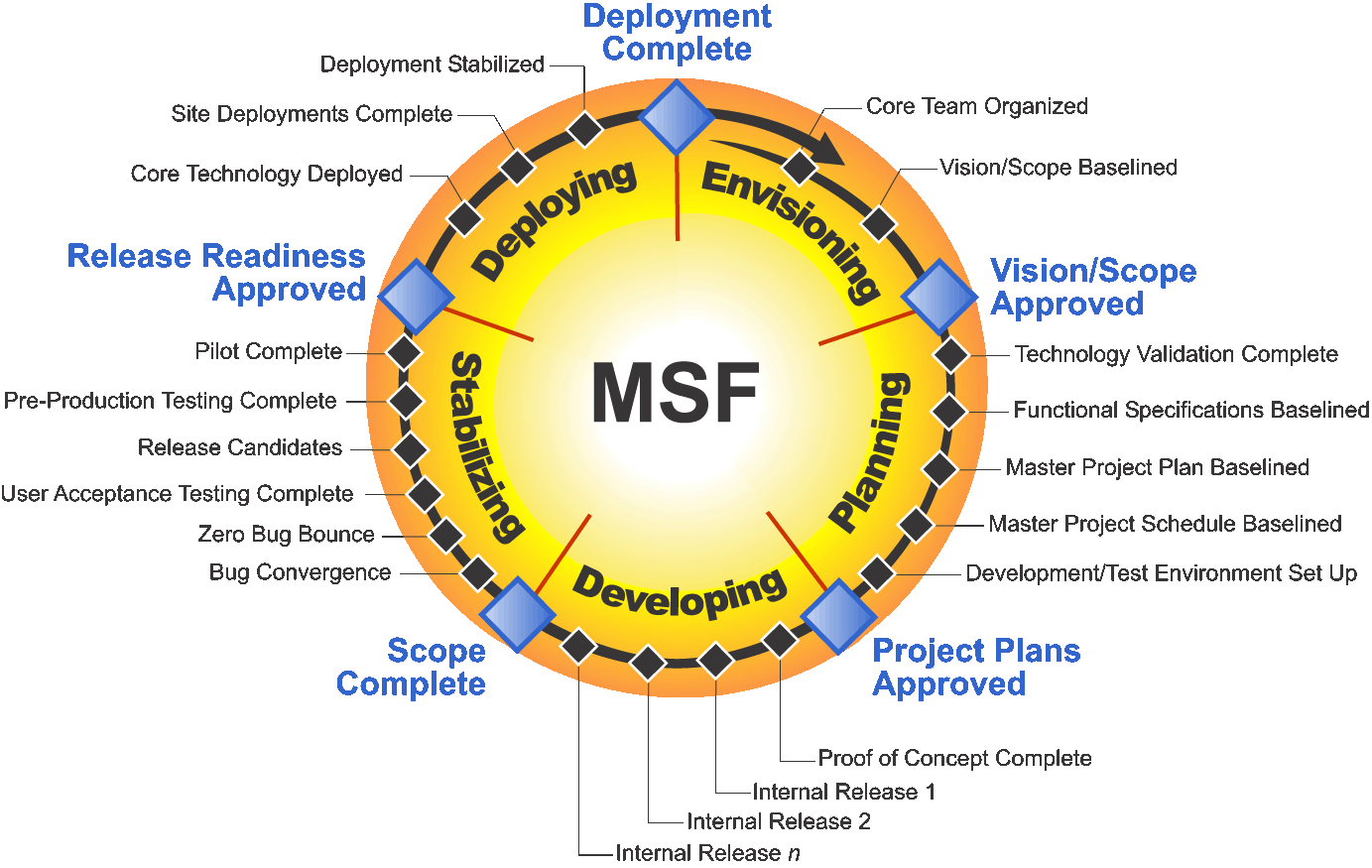
 — методология разработки программного обеспечения, созданная компанией Rational Software.   
  
[](https://upload.wikimedia.org/wikipedia/commons/1/19/Development-iterative.png)

В основе методологии лежат 6 основных принципов:

• компонентная архитектура, реализуемая и тестируемая на ранних стадиях проекта;  
• работа над проектом в сплочённой команде, ключевая роль в которой принадлежит архитекторам;  
• ранняя идентификация и непрерывное устранение возможных рисков;  
• концентрация на выполнении требований заказчиков к исполняемой программе;  
• ожидание изменений в требованиях, проектных решениях и реализации в процессе разработки;  
• постоянное обеспечение качества на всех этапах разработки проекта.

Использование методологии RUP направлено на итеративную модель разработки. Особенность методологии состоит в том, что степень формализации может меняться в зависимости от потребностей проекта. Можно по окончании каждого этапа и каждой итерации создавать все требуемые документы и достигнуть максимального уровня формализации, а можно создавать только необходимые для работы документы, вплоть до полного их отсутствия. За счет такого подхода к формализации процессов методология является достаточно гибкой и широко популярной. Данная методология применима как в небольших и быстрых проектах, где за счет отсутствия формализации требуется сократить время выполнения проекта и расходы, так и в больших и сложных проектах, где требуется высокий уровень формализма, например, с целью дальнейшей сертификации продукта. Это преимущество дает возможность использовать одну и ту же команду разработчиков для реализации различных по объему и требованиям.

## MICROSOFT SOLUTIONS FRAMEWORK

 — методология разработки программного обеспечения, предложенная корпорацией Microsoft. MSF опирается на практический опыт Microsoft и описывает управление людьми и рабочими процессами в процессе разработки решения.  
[](http://blog.pa-resultant.ru/wp-content/uploads/2014/01/mof.gif)

Базовые концепции и принципы модели процессов MSF:

• единое видение проекта — все заинтересованные лица и просто участники проекта должны чётко представлять конечный результат, всем должна быть понятна цель проекта;  
• управление компромиссами — поиск компромиссов между ресурсами проекта, календарным графиком и реализуемыми возможностями;  
• гибкость — готовность к изменяющимся проектным условиям;  
• концентрация на бизнес-приоритетах — сосредоточенность на той отдаче и выгоде, которую ожидает получить потребитель решения;  
• поощрение свободного общения внутри проекта;  
• создание базовых версии — фиксация состояния любого проектного артефакта, в том числе программного кода, плана проекта, руководства пользователя, настройки серверов и последующее эффективное управление изменениями, аналитика проекта.  
  
MSF предлагает проверенные методики для планирования, проектирования, разработки и внедрения успешных IT-решений. Благодаря своей гибкости, масштабируемости и отсутствию жестких инструкций MSF способен удовлетворить нужды организации или проектной группы любого размера. Методология MSF состоит из принципов, моделей и дисциплин по управлению персоналом, процессами, технологическими элементами и связанными со всеми этими факторами вопросами, характерными для большинства проектов.

**5. Continuous delivery, Continuous Deployment и Continuous Integration.**

### Что значит непрерывный?

Прежде, чем мы начнём разбираться в различных концепциях **DevOps**, следует понять, что значит “непрерывный” в области программного обеспечения. Проще говоря, термин “непрерывности” относится к изменениям программного обеспечения, которые происходят в течении всего процесса разработки **ПО**.

Конечно же есть доля лукавства в термине “непрерывный”. На самом деле после реализации функционала может пройти некоторое время до того, как код попадёт в продакшен, но это время всё-таки гораздо меньше, чем было до появления DevOps.

## Continuous delivery (непрерывная доставка)

В большинстве случаев **непрерывная доставка** — это серия практик, направленных на то, чтобы обновления программного обеспечения происходили практически постоянно. Данные методы гарантируют быстрое развёртывание на продакшене не меняя существующий функционал. Непрерывная доставка осуществима благодаря различным оптимизациям на ранних этапах процесса разработки.

Разработчик, сделав какую-либо фичу, отправляет её QA-инженерам для тестирования. Тестировщикам легче досконально оттестировать небольшой новый функционал и написать к нему тест-кейсы. Как только все проверки – прошли, новая фича попадает на дальнейшее тестирование авто-тестами и потом уже в релизный брэнч в системе контроля версий.

**Continuous delivery** поставляет бизнесу каждый функционал постепенно. Это позволяет получить сразу отклик от клиента и, при необходимости, сделать некоторые изменения.

#### Другие преимущества Continuous delivery:

1. Внесение нового функционала в back-end для проверки совместимости с системой;
2. Быстрое реагирование на потребности рынка;
3. Возможность подстраивания под изменение бизнес-стратегии;
4. Низкое количество потенциальных ошибок.

## Continuous deployment(непрерывное развёртываение)

**Continuous deployment** часто путают с continuous delivery, хотя между ними существуют чёткие различия, которые следует знать и понимать.

Как выше было уже сказано непрерывная доставка обеспечивает постоянный выпуск обновлений пользователям. А **непрерывное развёртывание** отвечает за то, чтобы весь новый функционал после тестирования сразу же попал в основную программу без ручного вмешательства инженеров DevOps.

Тот же Docker создан для неприрывного развёртывания. DevOps инженеры могут обновлять контейнеры и разворачивать их сразу на продакшене в автоматическом режиме. Такой процесс является ключом к непрерывной доставке, т.к. весь процесс может занять всего лишь несколько минут.

Не всегда непрерывное развёртывание имеет смысл. Использование фича-тоглинга сводит на нет все преимущества. Всегда надо исходить из потребностей бизнеса и процессов внедрения нового функционала.

## Continuous integration (непрерывная интеграция)

**Непрерывная интеграция** является ключевым компонентом практики Agile Development. Основой данной практики является постоянное попадание кода в центральный репозиторий после успешного запуска тестов. Основные цели **continuous integration** – поиск и устранение потенциальных проблем как можно быстрее, улучшение качества ПО и сокращение время для выпуска обновлений.

До того, как непрерывная интеграция стала широко распространённой, разработчики обычно работали изолировано, а только по окончанию работы объедениняли свои наработки. Порой это был очень трудоёмкий и длительный процесс.

При непрерывной интеграции разработчики часто заливают свои изменения в центральный репозиторий, выполняя до этого unit – тесты. Затем система контроля версий автоматически проверяет код на возможно безопасной интеграции с существующим в репозитории. При этом идёт постоянное поступление кода, что облегчает тестирование и сводит к минимуму возможные риски.

## Как это всё работает вместе?

Перед тем как вы дойдёте до непрерываного развёртывания вам предстоит длинный путь. Сначала вы должны будете многое автоматизировать и пройти фазы непрерываной интеграции и доставки.

**6. Виды тестовой документации.**

Внешняя документация:

* Замечание – короткая записка, комментарий о небольшой неточности в реализации продукта.
* Баг-репорт – описание выявленного случая несоответствия производимого продукта требованиям, к нему выдвигаемым – ошибки или ее проявления. Он обязательно должен содержать следующие элементы:   
  + Идею тестового случая, вызвавшего ошибку.
  + Описание исходного состояния системы для выполнения кейса.
  + Шаги, необходимые для того, чтобы выявить ошибку или ее проявление.
  + Ожидаемый результат, т. е. то, что должно было произойти в соответствии с требованиями.
  + Фактический результат, т. е. то, что произошло на самом деле.
  + Входные данные, которые использовались во время воспроизведения кейса.
  + Прочую информацию, без которой повторить кейс не получится.
  + Критичность и/или приоритет.
  + Экранный снимок (скрин).
  + Версию, сборку, ресурс и другие данные об окружении.
* Запрос на изменение (улучшение) – описание неявных/некритичных косвенных требований, которые не были учтены при планировании/реализации продукта, но несоблюдение, которых может вызвать неприятие у конечного потребителя. И пути/рекомендации по модификации продукта для соответствия им.
* Отчет о тестировании (тест репорт) – документ, предоставляющий сведения о соответствии/ несоответствии продукта требованиям. Может так же содержать описание некоторых подробностей проведенной сессии тестирования, например, затраченное время, использованные виды тестирования, перечень проверенных случаев и т. п. В идеальном варианте фраза вида «Тест пройден. Ошибка не воспроизводится/Функционал работает корректно/Соответствует требованиям» означает, что продукт или его часть полностью соответствует требованиям прямым и косвенным (в производстве ПО).

Внутренняя документация:

* Тест-план (план тестирования) – формализованное и укрупненное описание одной сессии тестирования по одному или нескольким направлениям проверок. Т.е. перечень направлений проверок, которые должны быть проведены в рамках сессии тестирования (и, сообразных этим направлениям, требований). Также может содержать в себе необходимую информацию об окружении, методике, прочих условиях важных для показательности данной сессии тестирования. Под направлением проверок также может пониматься более детализированная тестовая документация (в виде ссылки на нее): чек листы, тестовые комплекты, тестовые сценарии, на которую необходимо опираться при проведении сессии тестирования. Основная цель документа – описать границы сессии тестирования, стабилизировать показательность данной сессии.
* Тестовый сценарий – последовательность действий над продуктом, которые связаны единым ограниченным бизнес-процессом использования, и сообразных им  проверок корректности поведения продукта в ходе этих действий. Может содержать информацию об исходном состоянии продукта для запуска сценария, входных данных и прочие сведения, имеющие определяющее значение для успешного и показательного проведения проверок по сценарию. Особенностью является линейность действий и проверок, т.е. зависимость последующих действий и проверок от успешности предыдущих. Цель документа – стабилизация покрытия аспектов продукта, необходимых для выполнения функциональной задачи, показательными необходимыми и достаточными проверками. Фактически при успешном прохождении всего тестового сценария мы можем сделать заключение о том, что продукт может выполнять ту или иную возложенную на него функцию.
* Тестовый комплект – некоторый набор формализованных тестовых случаев объединенных между собой по общему логическому признаку.
* Чек-лист (лист проверок) – перечень формализованных тестовых случаев в виде удобном для проведения проверок. Тестовые случаи в чек-листе не должны быть зависимыми друг от друга. Обязательно должен содержать в себе информацию о: идеях проверок, наборах входных данных, ожидаемых результатах, булевую отметку о прохождении/непрохождении тестового случая, булевую отметку о совпадении/несовпадении фактического и ожидаемого результата по каждой проверке. Может так же содержать шаги для проведения проверки, данные об особенностях окружения и прочую информацию необходимую для проведения проверок. Цель – обеспечить стабильность покрытия требований проверками необходимыми и достаточными для заключения о соответствии им продукта. Особенностью является то, что чек-листы компонуются теми тестовыми случаями, которые показательны для определенного требования.
* Тестовый случай (тест-кейс) – формализованное описание одной показательной проверки на соответствие требованиям прямым или косвенным. Обязательно должен содержать следующую информацию:   
  + Идея проверки.
  + Описание проверяемого требования или проверяемой части требования.
  + Используемое для проверки тестовое окружение.
  + Исходное состояние продукта перед началом проверки.
  + Шаги для приведения продукта в состояние, подлежащее проверке.
  + Входные данные для использования при воспроизведении шагов.
  + Ожидаемый результат.
  + Прочую информацию, необходимую для проведения проверки.

Цель – зафиксировать сгенерированную и отобранную показательную проверку в виде, позволяющем тестировщику любой квалификации ее провести и суметь проанализировать полученные результаты.  
  
Как видно, каждый последующий вид внутренней тестовой документации в определенной мере детализирует предыдущий. У каждого документа есть свое назначение и все вместе они – инструмент для облегчения генерации, отбора и воспроизведения тестовых случаев. Кроме того хорошо структурированная, поддерживаемая, читаемая, организованная и доступная тестовая документация позволяет в долгосрочной перспективе:

* Обеспечить стабильность покрытия требований проверками.
* Обеспечить показательность всех проводимых проверок.
* Обеспечить необходимость и достаточность проводимых проверок.
* Сэкономить время на этапах тестирования, сводя их к проведению проверок и анализу  и передаче результатов.
* Снизить входной уровень квалификации тестировщика для проведения проверок.
* Повысить прогнозируемость сессий тестирования в части затрат времени и ресурсов.
* Повысить прозрачность процесса тестирования для других участников процесса производства продукта.
* Обеспечить базу знаний о продукте и истории его развития.

Но следует учитывать, что есть и свои недостатки:

* Стабильность покрытия. Со стремящейся к бесконечности долей вероятности, если проводится тестирование по документации, то будут проведены только те проверки, которые есть в данной документации. Вероятность пропуска ошибки (чаще всего несоответствие косвенному требованию, непокрытому документацией) возрастает.
* Плохая локализация ошибки тестировщиком. Либо полное отсутствие локализации. Фактический результат не совпал с ожидаемым – ошибка. А что это на самом деле: ошибка; проявление ошибки; инцидент, уже описанной ошибки, тестировщик не проверит (в подавляющем количестве случаев).
* Высокий требуемый уровень квалификации специалиста для создания и поддержания тестовой документации.
* Большие временные затраты на создание и поддержание тестовой документации.
* Слабо прогнозируемое время актуальности тестовой документации.

Списки как плюсов так минусов можно продолжать, я указал только те, которые лежат на поверхности. Но понимание хотя бы этого списка крайне важно для нынешнего или будущего специалиста по тестированию. Вопрос, касающийся тестовой документации, преодолевает очень малый процент соискателей.  
  
Естественно, видов документации в тестировании гораздо больше, но без понимания назначения и особенностей этих документов работать в этой профессии очень сложно. И чтобы совсем не увеличивать размеры статьи, рассмотрим последний (для этой статьи) вопрос.

**7. Принципы ООП**

Абстра́кция  — в объектно-ориентированном программировании это придание объекту характеристик, которые отличают его от всех других объектов, четко определяя его концептуальные границы. Основная идея состоит в том, чтобы отделить способ использования составных объектов данных от деталей их реализации в виде более простых объектов, подобно тому, как функциональная абстракция разделяет способ использования функции и деталей её реализации в терминах более примитивных функций, таким образом, данные обрабатываются функцией высокого уровня с помощью вызова функций низкого уровня.

Такой подход является основой объектно-ориентированного программирования. Это позволяет работать с объектами, не вдаваясь в особенности их реализации. В каждом конкретном случае применяется тот или иной подход: инкапсуляция, полиморфизм или наследование. Например, при необходимости обратиться к скрытым данным объекта, следует воспользоваться инкапсуляцией, создав, так называемую, функцию доступа или свойство.

Абстракция данных — популярная и в общем неверно определяемая техника программирования. Фундаментальная идея состоит в разделении несущественных деталей реализации подпрограммы и характеристик существенных для корректного ее использования. Такое разделение может быть выражено через специальный «интерфейс», сосредотачивающий описание всех возможных применений программы[1].

С точки зрения теории множеств, процесс представляет собой организацию для группы подмножеств своего множества. См. также Закон обратного отношения между содержанием и объемом понятия.

Инкапсуля́ция — свойство языка программирования, позволяющее пользователю не задумываться о сложности реализации используемого программного компонента (что у него внутри?), а взаимодействовать с ним посредством предоставляемого интерфейса (публичных методов и членов), а также объединить и защитить жизненно важные для компонента данные. При этом пользователю предоставляется только спецификация (интерфейс) объекта.

Пользователь может взаимодействовать с объектом только через этот интерфейс. Реализуется с помощью ключевого слова: public.

Пользователь не может использовать закрытые данные и методы. Реализуется с помощью ключевых слов: private, protected, internal.

Инкапсуляция — один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с абстракцией, полиморфизмом и наследованием).

Сокрытие реализации целесообразно применять в следующих случаях:

предельная локализация изменений при необходимости таких изменений,

прогнозируемость изменений (какие изменения в коде надо сделать для заданного изменения функциональности) и прогнозируемость последствий изменений.

Насле́дование — один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с инкапсуляцией, полиморфизмом и абстракцией), позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом.

Другими словами, класс-наследник реализует спецификацию уже существующего класса (базовый класс). Это позволяет обращаться с объектами класса-наследника точно так же, как с объектами базового класса.

Простое наследование:

Класс, от которого произошло наследование, называется базовым или родительским (англ. base class). Классы, которые произошли от базового, называются потомками, наследниками или производными классами (англ. derived class).

В некоторых языках используются абстрактные классы. Абстрактный класс — это класс, содержащий хотя бы один абстрактный метод, он описан в программе, имеет поля, методы и не может использоваться для непосредственного создания объекта. То есть от абстрактного класса можно только наследовать. Объекты создаются только на основе производных классов, наследованных от абстрактного. Например, абстрактным классом может быть базовый класс «сотрудник вуза», от которого наследуются классы «аспирант», «профессор» и т. д. Так как производные классы имеют общие поля и функции (например, поле «год рождения»), то эти члены класса могут быть описаны в базовом классе. В программе создаются объекты на основе классов «аспирант», «профессор», но нет смысла создавать объект на основе класса «сотрудник вуза».

Множественное наследование

При множественном наследовании у класса может быть более одного предка. В этом случае класс наследует методы всех предков. Достоинства такого подхода в большей гибкости. Множественное наследование реализовано в C++. Из других языков, предоставляющих эту возможность, можно отметить Python и Эйфель. Множественное наследование поддерживается в языке UML.

Множественное наследование — потенциальный источник ошибок, которые могут возникнуть из-за наличия одинаковых имен методов в предках. В языках, которые позиционируются как наследники C++ (Java, C# и др.), от множественного наследования было решено отказаться в пользу интерфейсов. Практически всегда можно обойтись без использования данного механизма. Однако, если такая необходимость все-таки возникла, то, для разрешения конфликтов использования наследованных методов с одинаковыми именами, возможно, например, применить операцию расширения видимости — «::» — для вызова конкретного метода конкретного родителя.

Попытка решения проблемы наличия одинаковых имен методов в предках была предпринята в языке Эйфель, в котором при описании нового класса необходимо явно указывать импортируемые члены каждого из наследуемых классов и их именование в дочернем классе.

Большинство современных объектно-ориентированных языков программирования (C#, Java, Delphi и др.) поддерживают возможность одновременно наследоваться от класса-предка и реализовать методы нескольких интерфейсов одним и тем же классом. Этот механизм позволяет во многом заменить множественное наследование — методы интерфейсов необходимо переопределять явно, что исключает ошибки при наследовании функциональности одинаковых методов различных классов-предков.

Полиморфи́зм — возможность объектов с одинаковой спецификацией иметь различную реализацию.

Язык программирования поддерживает полиморфизм, если классы с одинаковой спецификацией могут иметь различную реализацию — например, реализация класса может быть изменена в процессе наследования[1].

Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций».

Полиморфизм — один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с абстракцией, инкапсуляцией и наследованием).

Полиморфизм позволяет писать более абстрактные программы и повысить коэффициент повторного использования кода. Общие свойства объектов объединяются в систему, которую могут называть по-разному — интерфейс, класс. Общность имеет внешнее и внутреннее выражение:

внешняя общность проявляется как одинаковый набор методов с одинаковыми именами и сигнатурами (именем методов и типами аргументов и их количеством);

внутренняя общность — одинаковая функциональность методов. Её можно описать интуитивно или выразить в виде строгих законов, правил, которым должны подчиняться методы. Возможность приписывать разную функциональность одному методу (функции, операции) называется перегрузкой метода (перегрузкой функций, перегрузкой операций).

 Формы полиморфизма

Используя Параметрический полиморфизм можно создавать универсальные базовые типы. В случае параметрического полиморфизма, функция реализуется для всех типов одинаково и таким образом функция реализована для произвольного типа. В параметрическом полиморфизме рассматриваются параметрические методы и типы.

Параметрические методы.

Если полиморфизм включения влияет на наше восприятие объекта, то параметрические полиморфизм влияет на используемые методы, так как можно создавать методы родственных классов, откладывая объявление типов до времени выполнения. Для избежания написания отдельного метода каждого типа применяется параметрический полиморфизм, при этом тип параметров будет являться таким же параметром, как и операнды.

Параметрические типы.

Вместо того, чтобы писать класс для каждого конкретного типа следует создать типы, которые будут реализованы во время выполнения программы то есть мы создаем параметрический тип.

Полиморфизм переопределения.

Абстрактные методы часто относятся к отложенным методам. Класс, в котором определен этот метод может вызвать метод и полиморфизм обеспечивает вызов подходящей версии отложенного метода в дочерних классах. Специальный полиморфизм допускает специальную реализацию для данных каждого типа.

Полиморфизм-перегрузка

- это частный случай полиморфизма. С помощью перегрузки одно и то же имя может обозначать различные методы, причем методы могут различаться количеством и типом параметров, то есть не зависят от своих аргументов. Метод может не ограничиваться специфическими типами параметров многих различных типов.

**8. Maven**

**Apache Maven** — [фреймворк](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA) для [автоматизации сборки](https://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_%D1%81%D0%B1%D0%BE%D1%80%D0%BA%D0%B8) проектов на основе описания их структуры в файлах на языке [POM](https://en.wikipedia.org/wiki/Project_Object_Model), являющемся подмножеством [XML](https://ru.wikipedia.org/wiki/XML)[[3]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-3). Проект Maven издаётся сообществом Apache Software Foundation, где формально является частью [Jakarta Project](https://ru.wikipedia.org/wiki/Apache_Jakarta_Project).

Название программы, *maven*, — является словом из языка [идиш](https://ru.wikipedia.org/wiki/%D0%98%D0%B4%D0%B8%D1%88), смысл которого можно примерно выразить как «собиратель знания»[[4]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-4).

Maven обеспечивает [декларативную](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BB%D0%B0%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), а не [императивную](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) (в отличие от средства автоматизации сборки [Apache Ant](https://ru.wikipedia.org/wiki/Apache_Ant)) сборку проекта. В файлах описания проекта содержится его спецификация, а не отдельные команды выполнения. Все задачи по обработке файлов, описанные в спецификации, Maven выполняет посредством их обработки последовательностью встроенных и внешних плагинов.

Maven используется для построения и управления проектами, написанными на [Java](https://ru.wikipedia.org/wiki/Java), [C#](https://ru.wikipedia.org/wiki/C_Sharp), [Ruby](https://ru.wikipedia.org/wiki/Ruby), [Scala](https://ru.wikipedia.org/wiki/Scala_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)), и других языках[[5]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-5).

Информация для сборки проекта, поддерживаемого Apache Maven, содержится в [XML](https://ru.wikipedia.org/wiki/Extensible_Markup_Language)-файле с названием pom.xml. При запуске Maven проверяет, содержит ли конфигурационный файл все необходимые данные и все ли данные синтаксически правильно записаны.

Пример файла pom.xml:

**<project>**

*<!-- версия модели для POM-ов Maven 2.x всегда 4.0.0 -->*

**<modelVersion>**4.0.0**</modelVersion>**

*<!-- координаты проекта, то есть набор значений, который*

*позволяет однозначно идентифицировать этот проект -->*

**<groupId>**com.mycompany.app**</groupId>**

**<artifactId>**my-app**</artifactId>**

**<version>**1.0**</version>**

*<!-- зависимости от библиотек -->*

**<dependencies>**

**<dependency>**

*<!-- координаты необходимой библиотеки -->*

**<groupId>**junit**</groupId>**

**<artifactId>**junit**</artifactId>**

**<version>**3.8.1**</version>**

*<!-- эта библиотека используется только для запуска и компилирования тестов -->*

**<scope>**test**</scope>**

**</dependency>**

**</dependencies>**

**</project>**

Минимальная конфигурация включает версию конфигурационного файла, имя проекта, его автора и версию[[9]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-:0-9). С помощью pom.xml конфигурируются зависимости от других проектов, индивидуальные фазы процесса построения проекта (build process), список [плагинов](https://ru.wikipedia.org/wiki/%D0%9F%D0%BB%D0%B0%D0%B3%D0%B8%D0%BD), реализующих порядок сборки[[9]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-:0-9).

Крупные проекты могут быть поделены на несколько модулей, или подпроектов, каждый со своим собственным POM. Операции над модулями могут выполняться через общий корневой POM единой командой.

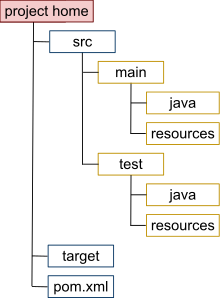
POM файлы подпроектов могут наследовать конфигурацию от других файлов конфигурации. В то же время все файлы конфигурации обязательно наследуются от «Супер POM» файла[[10]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-10) по умолчанию. Супер POM обеспечивает конфигурацию по умолчанию, например, стандартная структура каталогов, используемые по умолчанию плагины, привязка к фазам жизненного цикла и прочее.

## Основные концепции[[править](https://ru.wikipedia.org/w/index.php?title=Apache_Maven&veaction=edit&section=3) | [править код](https://ru.wikipedia.org/w/index.php?title=Apache_Maven&action=edit&section=3)]

### Соглашения по конфигурации[[править](https://ru.wikipedia.org/w/index.php?title=Apache_Maven&veaction=edit&section=4) | [править код](https://ru.wikipedia.org/w/index.php?title=Apache_Maven&action=edit&section=4)]

Maven поддерживает принцип [соглашения по конфигурации](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%B3%D0%BB%D0%B0%D1%88%D0%B5%D0%BD%D0%B8%D1%8F_%D0%BF%D0%BE_%D0%BA%D0%BE%D0%BD%D1%84%D0%B8%D0%B3%D1%83%D1%80%D0%B0%D1%86%D0%B8%D0%B8), заключающийся в том, что рассматриваемые аспекты нуждаются в конфигурации тогда и только тогда, когда этот аспект *не* удовлетворяет некоторой спецификации. Как следствие, это позволяет сократить количество требуемой конфигурации без потери гибкости. Одним из следствий применения данного принципа является то, что отсутствует необходимость указывать пути к файлам в явном виде, что упрощает содержимое *pom.xml*. Однако, почти все стандарты, на которые опирается Maven, могут быть изменены индивидуальной конфигурацией[[11]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-11)[[12]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-12).

### Архетипы[[править](https://ru.wikipedia.org/w/index.php?title=Apache_Maven&veaction=edit&section=5) | [править код](https://ru.wikipedia.org/w/index.php?title=Apache_Maven&action=edit&section=5)]

[](https://commons.wikimedia.org/wiki/File:Maven_CoC.svg?uselang=ru)

Maven автоматически генерирует представленную структуру для Java-проекта.

Maven использует принцип *Maven-архетипов* (англ. *Archetypes*). Архетип — это инструмент шаблонов, каждый из которых определён паттерном или моделью, по аналогии с которой создаются производные.[[13]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-13)

Стандартная структура каталогов — одна из реализаций принципа архетипов в Maven. Следующая структура показывает важнейшие каталоги для проекта на Java[[14]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-:2-14):

* Корневой каталог проекта: файл *pom.xml* и все дальнейшие подкаталоги
  + *src*: все исходные файлы
    - *src/main*: исходные файлы собственно для продукта
      * *src/main/java*: [Java](https://ru.wikipedia.org/wiki/Java)-исходный текст
      * *src/main/resources*: другие файлы, которые используются при компиляции или исполнении, например Properties-файлы
    - *src/test*: исходные файлы, необходимые для организации автоматического тестирования
      * *src/test/java*: [JUnit](https://ru.wikipedia.org/wiki/JUnit)-тест-задания для автоматического тестирования
  + *target*: все создаваемые в процессе работы Мавена файлы
    - *target/classes*: компилированные Java-классы

### Жизненный цикл[[править](https://ru.wikipedia.org/w/index.php?title=Apache_Maven&veaction=edit&section=6) | [править код](https://ru.wikipedia.org/w/index.php?title=Apache_Maven&action=edit&section=6)]

Жизненный цикл Maven проекта — это список поименованных *фаз*, определяющий порядок действий при его построении. Жизненный цикл Maven содержит три независимых порядка выполнения:[[15]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-15)

* clean — жизненный цикл для очистки проекта. Содержит следующие фазы:
  1. pre-clean
  2. clean
  3. post-clean
* default — основной жизненный цикл, содержащий следующие фазы:
  1. validate - выполняется проверка, является ли структура проекта полной и правильной.
  2. generate-sources
  3. process-sources
  4. generate-resources
  5. process-resources
  6. compile - компилируются исходные тексты.
  7. process-test-sources
  8. process-test-resources
  9. test-compile
  10. test - собранный код тестируется заранее подготовленным набором тестов.
  11. package - упаковка откомпилированных классов и прочих ресурсов. Например, в JAR-файл.
  12. integration-test - программное обеспечение в целом или его крупные модули подвергаются интеграционному тестированию. Проверяется взаимодействие между составными частями программного продукта.
  13. install - установка программного обеспечения в локальный Maven-репозиторий, чтобы сделать его доступным для других проектов текущего пользователя.
  14. deploy - стабильная версия программного обеспечения распространяется на удаленный Maven-репозиторий, чтобы сделать его доступным для других пользователей.
* site — жизненный цикл генерации проектной документации. Состоит из фаз:
  1. pre-site
  2. site
  3. post-site
  4. site-deploy

Стандартные жизненные циклы могут быть дополнены функционалом с помощью Maven-плагинов. Плагины позволяют вставлять в стандартный цикл новые шаги (например, распределение на сервер приложений) или расширять существующие шаги.

### Архитектура[[править](https://ru.wikipedia.org/w/index.php?title=Apache_Maven&veaction=edit&section=7) | [править код](https://ru.wikipedia.org/w/index.php?title=Apache_Maven&action=edit&section=7)]

Maven базируется на [plugin](https://ru.wikipedia.org/wiki/Plugin)-архитектуре, которая позволяет применять плагины для различных задач (*compile, test, build, deploy, checkstyle, pmd, scp-transfer*) для данного проекта, без необходимости их в явном виде инсталлировать. Это возможно за счет того, что информация поступает плагину через стандартный вход, а результаты пишутся в его стандартный выход. Теоретически, это позволяет кому угодно писать плагины для взаимодействия со средствами построения проекта (компиляторы, средства тестирования, и т. п.) для любого другого языка. В реальности, поддержка других языков кроме Java сейчас минимальна. Существует плагин для .NET-фреймворка[[16]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-16), а также плагины для [C](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))/[C++](https://ru.wikipedia.org/wiki/C%2B%2B) [[17]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-17)[[18]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-18).

Количество доступных плагинов в настоящее время очень велико и включает, в том числе, плагины, позволяющие непосредственно из Maven запускать web-приложение для тестирования его в браузере; плагины, позволяющие тестировать или создавать банки данных; плагины, позволяющие генерировать Web Services. Задачей разработчика в такой ситуации является найти и применить наиболее подходящий набор плагинов.

Плагин обеспечивает достижение ряда целей с помощью следующего синтаксиса:

mvn [имя плагина]:[имя цели]

Например, Java-проект может быть скомпилирован плагином-компилятором[[19]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-19) путём выполнения команды: mvn compiler:compile

Существуют Maven-плагины для построения, тестирования, контроля исходного текста, запуска web-сервера, генерации [Eclipse](https://ru.wikipedia.org/wiki/Eclipse_(%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8))-проектных файлов и множество других.[[20]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-20) Плагины перечисляются и конфигурируются в секции <plugins> файла *pom.xml*. Некоторая базовая группа плагинов включается в каждый проект по умолчанию.

### Зависимости[[править](https://ru.wikipedia.org/w/index.php?title=Apache_Maven&veaction=edit&section=8) | [править код](https://ru.wikipedia.org/w/index.php?title=Apache_Maven&action=edit&section=8)]

В файле pom.xml задаются зависимости, которые имеет управляемый с помощью Maven проект. Менеджер зависимостей основан на нескольких основных принципах:

* **Репозитории**. Maven ищет необходимые файлы в локальных каталогах или в локальном Maven-репозитории. Если зависимость не может быть локально разрешена, Maven подключается к указанному Maven-репозиторию в сети и копирует в локальный репозиторий. По умолчанию Maven использует Maven Central Repository[[21]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-maven2repo-21), но разработчик может конфигурировать и другие публичные Maven-репозитории, такие, как Apache, Ibiblio, Codehaus или Java.Net.
* **Транзитивные зависимости**. Необходимые библиотеки подгружаются в проект автоматически. При разрешении конфликта версий используется принцип «ближайшей» зависимости, то есть выбирается зависимость, путь к которой через список зависимых проектов является наиболее коротким.
* **Исключение зависимостей**. Файл описания проекта предусматривает возможность исключить зависимость в случае обнаружения цикличности или отсутствия необходимости в определённой библиотеке.
* **Поиск зависимостей**. Поиск зависимостей (open-source-библиотек и модулей) ведётся по их координатам (groupId, artifactId и version). Эти координаты могут быть определены с помощью специальных поисковых машин, например, Maven search engine[[22]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-autogenerated1-22). Например, по поисковому признаку «pop3» поисковая машина предоставляет результат с groupId="com.sun.mail" и artifactId="pop3".
* **Менеджеры репозиториев**. Репозитории реализуются с помощью *менеджеров репозиториев Maven* (Maven Repository Manager), таких как Apache Archiva, Nexus (ранее Proximity), Artifactory, Codehaus Maven Proxy или Dead Simple Maven Proxy[[23]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-23).

Область распространения зависимости позволяет включать зависимости только на определённую стадию построения проекта. Существует 6 возможных областей:[[24]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-24)

1. compile. Область по умолчанию. Зависимость доступна во всех путях поиска классов в проекте. Распространяется на зависимые проекты.
2. provided. Область аналогична compile, за исключением того, что JDK или контейнер сам предоставит зависимость во время выполнения программы.
3. runtime. Зависимость не нужна для компиляции, но нужна для выполнения.
4. test. Зависимость не нужна для нормальной работы приложения, а нужна только для компиляции и запуска тестов.
5. system. Область аналогична provided за исключением того, что содержащий зависимость JAR указывается явно. Артефакт не ищется в репозитории.
6. import (начиная с версии Maven 2.0.9) используется только с зависимостью типа pom в секции <dependencyManagement>. Зависимости текущего POM заменяются на зависимости из указанного POM.

## Приведение проекта к виду Maven[[править](https://ru.wikipedia.org/w/index.php?title=Apache_Maven&veaction=edit&section=9) | [править код](https://ru.wikipedia.org/w/index.php?title=Apache_Maven&action=edit&section=9)]

Проект, поддерживающийся с помощью Maven, должен удовлетворять некоторым условиям для возможности его чтения утилитой, последующего анализа и возможности сборки. Это накладывает некоторые ограничения на структуру директорий и требует дополнительных действий, если проект изначально имеет отличную структуру.[[25]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-25)

Для того, чтобы Maven распознал проект, как подлежащий обработке, он должен содержать установленную структуру каталогов. Все файлы с исходным кодом должны находиться по относительному пути «*\src\main\java*»[[14]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-:2-14).

Файл конфигурации web-проекта web.xml должен находиться в директории «*\src\main\webapp\WEB-INF*».[[14]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-:2-14)

Файл конфигурации Maven-проекта pom.xml должен находиться в корневом каталоге проекта. В нём согласно его предназначению могут быть указаны удаленный репозиторий, плагины для создания архивов, плагин компилятора и прочее. Для web-проекта также необходимо добавлять дополнительные зависимости, как например javaee.jar.

Таким образом, выходной файл конфигурации web-проекта, согласованный с Maven, выглядит следующим образом:

**<project** xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"**>**

**<modelVersion>**4.0.0**</modelVersion>**

**<groupId>**com.mkyong**</groupId>**

**<artifactId>**servletdemo**</artifactId>**

**<packaging>**war**</packaging>**

**<version>**1.0-SNAPSHOT**</version>**

**<name>**servletdemo**</name>**

**<url>**http://maven.apache.org**</url>**

**<repositories>**

**<repository>**

**<id>**java.net**</id>**

**<url>**http://download.java.net/maven/2**</url>**

**</repository>**

**</repositories>**

**<build>**

**<plugins>**

**<plugin>**

**<groupId>**org.apache.maven.plugins**</groupId>**

**<artifactId>**maven-war-plugin**</artifactId>**

**<configuration>**

**<webResources>**

**<resource>**

**<directory>**${basedir}/src/main/java**</directory>**

**<targetPath>**WEB-INF/classes**</targetPath>**

**<includes>**

**<include>**\*\*/\*.properties**</include>**

**<include>**\*\*/\*.xml**</include>**

**<include>**\*\*/\*.css**</include>**

**<include>**\*\*/\*.html**</include>**

**</includes>**

**</resource>**

**</webResources>**

**</configuration>**

**</plugin>**

**<plugin>**

**<artifactId>**maven-compiler-plugin**</artifactId>**

**<configuration>**

**<source>**1.6**</source>**

**<target>**1.6**</target>**

**</configuration>**

**</plugin>**

**</plugins>**

**</build>**

**</project>**

После выполнения всех указанных требований Maven-проект готов для выполнения фаз жизненного цикла, таких, как компиляция, сборка архива и генерация документации[[26]](https://ru.wikipedia.org/wiki/Apache_Maven#cite_note-26).

Пример выходного лога сообщений при выполнении команды mvn war:war :

E:\workspace\servletdemo>mvn war:war

[INFO] Scanning for projects...

.......

[INFO] Processing war project

[INFO] Copying webapp resources[E:\workspace\servletdemo]

[INFO] Webapp assembled in[47 msecs]

[INFO] Building war: E:\workspace\servletdemo\target\servletdemo-1.0-SNAPSHOT.war

[INFO] -----------------------------------------------

[INFO] BUILD SUCCESSFUL

[INFO] -----------------------------------------------

**9. Ant**

**Apache Ant** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *ant* — муравей и [акроним](https://ru.wikipedia.org/wiki/%D0%90%D0%BA%D1%80%D0%BE%D0%BD%D0%B8%D0%BC) — «Another Neat Tool») — [утилита](https://ru.wikipedia.org/wiki/%D0%A3%D1%82%D0%B8%D0%BB%D0%B8%D1%82%D0%B0) для автоматизации процесса сборки программного продукта. Является платформонезависимым аналогом утилиты [make](https://ru.wikipedia.org/wiki/Make), где все команды записываются в [XML](https://ru.wikipedia.org/wiki/XML)-формате.

Ant был создан в рамках проекта [Jakarta](https://ru.wikipedia.org/wiki/Apache_Jakarta_Project), сегодня — самостоятельный проект первого уровня [Apache Software Foundation](https://ru.wikipedia.org/wiki/Apache_Software_Foundation).

Первая версия была разработана инженером [Sun Microsystems](https://ru.wikipedia.org/wiki/Sun_Microsystems) [Джеймсом Дэвидсоном](https://ru.wikipedia.org/w/index.php?title=%D0%94%D1%8D%D0%B2%D0%B8%D0%B4%D1%81%D0%BE%D0%BD,_%D0%94%D0%B6%D0%B5%D0%B9%D0%BC%D1%81_%D0%94%D1%83%D0%BD%D0%BA%D0%B0%D0%BD&action=edit&redlink=1) ([James Davidson](https://en.wikipedia.org/wiki/James_Duncan_Davidson) (англ.)[русск.](https://ru.wikipedia.org/w/index.php?title=James_Duncan_Davidson&action=edit&redlink=1)), который нуждался в утилите, подобной *make*, разрабатывая первую референтную реализацию [J2EE](https://ru.wikipedia.org/wiki/J2EE).

Ant, в отличие от другого сборщика проектов [Apache Maven](https://ru.wikipedia.org/wiki/Apache_Maven), обеспечивает [императивную](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), а не [декларативную](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BB%D0%B0%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)сборку проекта.

В отличие от make, утилита Ant полностью независима от платформы, требуется лишь наличие на применяемой системе установленной рабочей среды [Java](https://ru.wikipedia.org/wiki/Java) — [JRE](https://ru.wikipedia.org/wiki/JRE). Отказ от использования команд [операционной системы](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) и формат [XML](https://ru.wikipedia.org/wiki/XML) обеспечивают переносимость сценариев.

Управление процессом сборки происходит посредством *XML*-сценария, также называемого Build-файлом. В первую очередь этот файл содержит определение проекта, состоящего из отдельных целей (***Targets***). Цели сравнимы с [процедурами](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) в языках программирования и содержат вызовы команд-заданий (***Tasks***). Каждое задание представляет собой неделимую, атомарную команду, выполняющую некоторое элементарное действие.

Между целями могут быть определены зависимости — каждая цель выполняется только после того, как выполнены все цели, от которых она зависит (если они уже были выполнены ранее, повторного выполнения не производится).

Типичными примерами целей являются ***clean*** (удаление промежуточных файлов), ***compile*** ([компиляция](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) всех классов), ***deploy*** (развёртывание приложения на сервере). Конкретный набор целей и их взаимосвязи зависят от специфики проекта.

Ant позволяет определять собственные типы заданий путём создания [Java-классов](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), реализующих определённые [интерфейсы](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

## Часто применяемые задания (Tasks)[[править](https://ru.wikipedia.org/w/index.php?title=Apache_Ant&veaction=edit&section=2) | [править код](https://ru.wikipedia.org/w/index.php?title=Apache_Ant&action=edit&section=2)]

В поставку входит множество библиотек, затрагивающие широкий набор задач. Кроме этого на веб-сайте проекта доступны дополнительные библиотеки, расширяющие функциональность продукта, например, для интеграции со сторонними системами. Стандартная версия содержит более 150 типов задач. Примеры таких команд представлены ниже.

|  |  |
| --- | --- |
| **Код** | **Действие** |
| *javac* | компиляция Java-кода |
| *copy* | копирование файлов |
| *delete* | удаление файлов и директорий |
| *move* | перемещение файлов и директорий |
| *replace* | замещение [фрагментов](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B0%D0%B3%D0%BC%D0%B5%D0%BD%D1%82) текста в файлах |
| *junit* | автоматический запуск [юнит-тестов](https://ru.wikipedia.org/wiki/%D0%AE%D0%BD%D0%B8%D1%82-%D1%82%D0%B5%D1%81%D1%82) |
| *exec* | выполнение внешней команды |
| *zip* | создание архива в формате [Zip](https://ru.wikipedia.org/wiki/Zip) |
| *cvs* | выполнение [CVS](https://ru.wikipedia.org/wiki/CVS)-команды |
| *mail* | отправка [электронной почты](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BE%D1%87%D1%82%D0%B0) |
| [*xslt*](https://ru.wikipedia.org/wiki/Xslt) | наложение [XSLT](https://ru.wikipedia.org/wiki/XSLT)-преобразования |

## Пример сценария[[править](https://ru.wikipedia.org/w/index.php?title=Apache_Ant&veaction=edit&section=3) | [править код](https://ru.wikipedia.org/w/index.php?title=Apache_Ant&action=edit&section=3)]

Данный сценарий определяет две цели — *clean* и *build*. Первая выполняет удаление временных файлов, вторая — компиляцию и создание [JAR](https://ru.wikipedia.org/wiki/JAR)-архива. Цель *build* зависит от *clean*, поэтому при её запуске будет сначала выполнена очистка и только затем — задания собственно сборки. *Build* также выбрана в качестве цели по умолчанию. Это означает, что она будет выполнена при обработке данного сценария без явного указания цели.

<?xml version="1.0"?>

**<project** default="build" basedir="."**>**

**<property** name="name" value="AntBuildJar"**/>**

**<property** name="src.dir" location="${basedir}/src"**/>**

**<property** name="build" location="${basedir}/build"**/>**

**<property** name="build.classes" location="${build}/classes"**/>**

**<path** id="libs.dir"**>**

**<fileset** dir="lib" includes="\*\*/\*.jar"**/>**

**</path>**

*<!-- Сборка приложения -->*

**<target** name="build" depends="clean" description="Builds the application"**>**

*<!-- Создание директорий -->*

**<mkdir** dir="${build.classes}"**/>**

*<!-- Компиляция исходных файлов -->*

**<javac** srcdir="${src.dir}"

destdir="${build.classes}"

debug="false"

deprecation="true"

optimize="true" **>**

**<classpath** refid="libs.dir"**/>**

**</javac>**

*<!-- Копирование необходимых файлов -->*

**<copy** todir="${build.classes}"**>**

**<fileset** dir="${src.dir}" includes="\*\*/\*.\*" excludes="\*\*/\*.java"**/>**

**</copy>**

*<!-- Создание JAR-файла -->*

**<jar** jarfile="${build}/${name}.jar"**>**

**<fileset** dir="${build.classes}"**/>**

**</jar>**

**</target>**

*<!-- Очистка -->*

**<target** name="clean" description="Removes all temporary files"**>**

*<!-- Удаление файлов -->*

**<delete** dir="${build.classes}"**/>**

**</target>**

**</project>**

**10. Selenium**

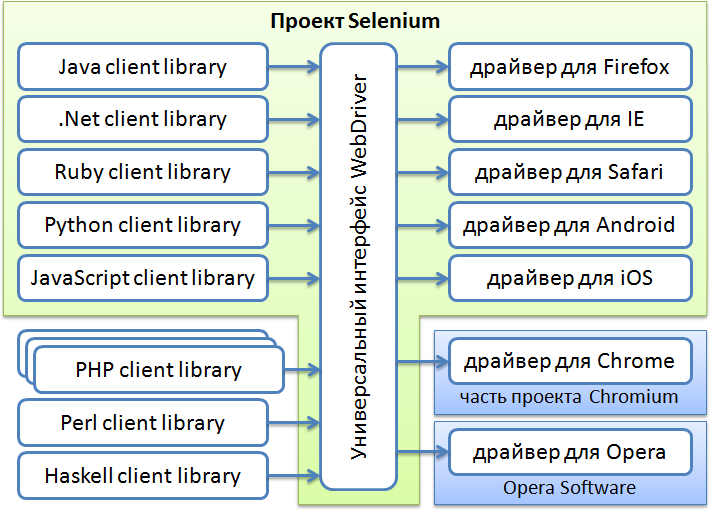
## Selenium

**Selenium – это проект, в рамках которого разрабатывается серия программных продуктов** с открытым исходным кодом (open source):

* Selenium WebDriver,
* Selenium RC,
* Selenium Server,
* Selenium Grid,
* Selenium IDE.

Называть просто словом Selenium любой из этих пяти продуктов, вообще говоря, неправильно, хотя так часто делают, если из контекста понятно, о каком именно из продуктов идёт речь, или если речь идёт о нескольких продуктах одновременно, или обо всех сразу.

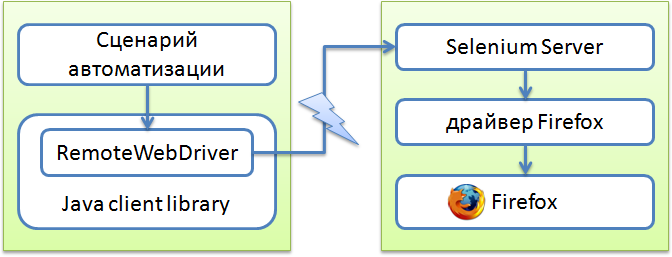
## Selenium WebDriver

**Selenium WebDriver – это программная библиотека для управления браузерами**. Часто употребляется также более короткое название WebDriver.  
  
Иногда говорят, что это «драйвер браузера», но на самом деле это целое семейство драйверов для различных браузеров, а также набор клиентских библиотек на разных языках, позволяющих работать с этими драйверами.  
  
Это основной продукт, разрабатываемый в рамках проекта Selenium.  
  
Selenium WebDriver называется также Selenium 2.0, причина этого будет объяснена ниже.  
  
Как уже было сказано, WebDriver представляет собой семейство драйверов для различных браузеров плюс набор клиентских библиотек для этих драйверов на разных языках программирования:  
  
  
  
В рамках проекта Selenium разрабатываются драйверы для браузеров Firefox, Internet Explorer и Safari, а также драйверы для мобильных браузеров Android и iOS. Драйвер для браузера Google Chrome разрабатывается в рамках проекта Chromium, а драйвер для браузера Opera (включая мобильные версии) разрабатывается компанией Opera Software. Поэтому они формально не являются частью проекта Selenium, распространяются и поддерживаются независимо. Но логически, конечно, можно считать их частью семейства продуктов Selenium.  
  
Аналогичная ситуация и с клиентскими библиотеками – в рамках проекта Selenium разрабатываются библиотеки для языков Java, .Net (C#), Python, Ruby, JavaScript. Все остальные реализации не имеют отношения к проекту Selenium, хотя, возможно, в будущем, какие-то из них могут влиться в этот проект.

## Selenium RC

**Selenium RC – это предыдущая версия библиотеки для управления браузерами**. Аббревиатура RC в названии этого продукта расшифровывается как Remote Control, то есть это средство для «удалённого» управления браузером.  
  
Эта версия с функциональной точки зрения значительно уступает WebDriver. Сейчас она находится в законсервированном состоянии, не развивается и даже известные баги не исправляются. А всем, кто сталкивается с ограничениями Selenium RC, предлагается переходить на использование WebDriver.  
  
Иногда Selenium RC называется также Selenium 1.0, тогда как WebDriver называется Selenium 2.0. Хотя на самом деле дистрибутив версии 2.0 включает в себя одновременно обе реализации – и Selenium RC, и WebDriver. А вот когда выйдет версия 3.0 – в ней останется только WebDriver.  
  
С технической точки зрения WebDriver не является результатом эволюционного развития Selenium RC, они построены на совершенно разных принципах и у них практически нет общего кода. Объединяет их лишь тот факт, что обе реализации были сделаны в рамках проекта Selenium. Ну, или если быть совсем точным, WebDriver сначала был самостоятельным проектом, но в 2008 году произошло слияние и сейчас WebDriver представляет собой основной вектор развития проекта Selenium.

## Selenium Server

**Selenium Server – это сервер, который позволяет управлять браузером с удалённой машины, по сети**. Сначала на той машине, где должен работать браузер, устанавливается и запускается сервер. Затем на другой машине (технически можно и на той же самой, конечно) запускается программа, которая, используя специальный драйвер RemoteWebDriver, соединяется с сервером и отправляет ему команды. Он в свою очередь запускает браузер и выполняет в нём эти команды, используя драйвер, соответствующий этому браузеру:  
  
  
  
Selenium Server поддерживает одновременно два набора команд – для новой версии (WebDriver) и для старой версии (Selenium RC).

## Selenium Grid

**Selenium Grid – это кластер, состоящий из нескольких Selenium-серверов**. Он предназначен для организации распределённой сети, позволяющей параллельно запускать много браузеров на большом количестве машин.  
  
Selenium Grid имеет топологию «звезда», то есть в его составе имеется выделенный сервер, который носит название «хаб» или «коммутатор», а остальные сервера называются «ноды» или «узлы». Сеть может быть гетерогенной, то есть коммутатор и узлы могут работать под управлением разных операционных систем, на них могут быть установлены разные браузеры. Одна из задач Selenium Grid заключается в том, чтобы «подбирать» подходящий узел, когда во время старта браузера указываются требования к нему – тип браузера, версия, операционная система, архитектура процессора и ряд других атрибутов.  
  
Ранее Selenium Grid был самостоятельным продуктом. Сейчас физически продукт один – Selenium Server, но у него есть несколько режимов запуска: он может работать как самостоятельный сервер, как коммутатор кластера, либо как узел кластера, это определяется параметрами запуска.

## Selenium IDE

**Selenium IDE – плагин к браузеру Firefox, который может записывать действия пользователя**, воспроизводить их, а также генерировать код для WebDriver или Selenium RC, в котором выполняются те же самые действия. В общем, это «Selenium-рекордер».  
  
Тестировщики, которые не умеют (или не хотят) программировать, используют Selenium IDE как самостоятельный продукт, без преобразования записанных сценариев в программный код. Это, конечно, не позволяет разрабатывать достаточно сложные тестовые наборы, но некоторым хватает и простых линейных сценариев.

Эта статья является продолжением более общей статьи [«Что такое Selenium?»](http://habrahabr.ru/post/152653/), в которой объясняется, какое положение занимает Selenium WebDriver среди других инструментов автоматизации веб-приложений.  
  
Здесь я постараюсь рассказать более подробно о том, что такое Selenium WebDriver, и почему его бессмысленно сравнивать с TestComplete, QuickTest Pro и другими инструментами автоматизации тестирования. И дело не только в том, что Selenium WebDriver бесплатный и открытый – его столь же бессмысленно сравнивать с другими бесплатными инструментами, такими как Sahi или Robot Framework.  
  
Почему?  
  
Потому что **Selenium WebDriver – это не инструмент для автоматизации тестирования**.  
  
А что же это такое?  
  
На этот вопрос можно дать несколько разных ответов, сначала я дам короткие ответы, а потом – более подробные.  
  
Кроме того, я объясню, почему Selenium WebDriver имеет такой убогий и неудобный в использовании интерфейс (набор команд), почему он не генерирует красивые отчёты и почему несмотря на всё это он настолько популярен :)  
  
На всякий случай оговорюсь, что хотя в этой статье речь идёт про WebDriver, многие аргументы справедливы и в отношении Selenium RC, но я не буду ничего говорить специально про эту устаревшую версию, потому что её место – на свалке истории.

## Итак, что такое Selenium WebDriver?

**По назначению Selenium WebDriver представляет собой драйвер браузера**, то есть программную библиотеку, которая позволяет разрабатывать программы, управляющие поведением браузера.  
  
**По своей сущности Selenium WebDriver представляет собой**:

* **спецификацию программного интерфейса для управления браузером**,
* **референсные реализации этого интерфейса для нескольких браузеров**,
* **набор клиентских библиотек для этого интерфейса на нескольких языках программирования**.

Теперь понятно, почему бессмысленно сравнивать Selenium WebDriver с «другими инструментами тестирования»? Непонятно? Тогда добавим подробностей.

## Selenium WebDriver – это драйвер браузера

Наверняка каждый, кто сталкивался с компьютерами, даже не айтишник, знает слово «драйвер». Это такая маленькая программа, точнее программная библиотека, которая позволяет другим программам взаимодействовать с некоторым устройством. Драйвер принтера позволяет печатать что-нибудь на принтере. Драйвер диска позволяет читать и писать данные. Драйвер сетевой карты позволяет обмениваться данными с другими компьютерами по сети.  
  
С драйвером пользователи не работают непосредственно. Они работают с прикладными программами, которые, посредством драйверов, взаимодействуют с теми или иными устройствами. Драйвер не имеет пользовательского интерфейса. Постойте, но ведь иногда бывает пользовательский интерфейс для настройки драйвера? Бывает. Но это интерфейс программы для настройки драйвера, а не самого драйвера. Драйвер имеет только программный интерфейс, его назначение состоит в том, чтобы дать возможность прикладным пользовательским программам взаимодействовать с устройством.  
  
Так вот, **Selenium WebDriver, или просто WebDriver – это драйвер браузера, то есть не имеющая пользовательского интерфейса программная библиотека, которая позволяет различным другим программам взаимодействовать с браузером, управлять его поведением, получать от браузера какие-то данные и заставлять браузер выполнять какие-то команды.**  
  
Исходя из этого определения, ясно, что **WebDriver не имеет прямого отношения к тестированию**. Он всего лишь предоставляет автотестам доступ к браузеру. На этом его функции заканчиваются.  
  
Структурирование, группировку и запуск тестов, а также генерацию отчётов о тестировании, обеспечивает фреймворк тестирования, такой как [JUnit](http://www.junit.org/) или [TestNG](http://testng.org/) для Java, [NUnit](http://www.nunit.org/) или [Gallio](http://www.gallio.org/) для .Net, [RSpec](http://rspec.info/) или [Cucumber](http://cukes.info/) для Ruby и так далее. Разработка тестов ведётся в среде [Eclipse](http://www.eclipse.org/), [Intellij IDEA](http://www.jetbrains.com/idea/), [Visual Studio](http://www.microsoft.com/visualstudio/rus/products/visual-studio-overview), [RubyMine](http://www.jetbrains.com/ruby/) и так далее. Сборка осуществляется посредством [Maven](http://maven.apache.org/), [Gradle](http://www.gradle.org/), [Ant](http://ant.apache.org/), [NAnt](http://nant.sourceforge.net/),[Rake](http://rake.rubyforge.org/) и так далее. Запуск тестов по расписанию и публикацию отчётов выполняет сервер непрерывной интеграции – [Jenkins](http://jenkins-ci.org/), [CruiseControl](http://cruisecontrol.sourceforge.net/), [Bamboo](http://www.atlassian.com/software/bamboo/), [TeamCity](http://www.jetbrains.com/teamcity/) и так далее. И всё это – самостоятельные инструменты, не имеющие отношения к проекту Selenium.  
  
Впрочем, в рамках проекта Selenium разрабатывается не только драйвер, но ещё несколько сопутствующих продуктов – Selenium Server позволяет организовать удалённый запуск браузера, при помощи Selenium Grid можно построить кластер из Selenium-серверов. Они встают в один ряд с вышеперечисленными инструментами и фреймворками, потому что также участвуют в построении системы запуска тестов. Кроме того, имеется «рекордер», который называется Selenium IDE, он умеет записывать действия пользователя и генерировать код, в котором используется интерфейс WebDriver для выполнения записанных действий.  
  
Но главным в проекте Selenium является именно WebDriver, это ключевой элемент экосистемы Selenium.  
  
Существуют ли другие драйверы? Разумеется.  
  
Внутри каждого коммерческого «интегрированного» инструмента имеются драйверы браузеров, но они как правило не могут быть использованы отдельно вне этого инструмента. Есть и бесплатные открытые драйверы – [Watir](http://watir.com/) предоставляет доступ к основным браузерам, [WatiN](http://watin.org/) имеет неплохой драйвер для браузера Internet Explorer, [Sahi](http://sahi.co.in/) умеет работать с «большой пятёркой» браузеров.

## Как сравнить Selenium WebDriver с другими инструментами?

Из всего вышенаписанного можно сделать вывод, что сравнивать WebDriver с каким-нибудь инструментом тестирования типа TestComplete или Sahi бессмысленно. Они находятся в разных весовых категориях. Это всё равно, что сравнивать драйвер принтера с текстовым редактором.  
  
А что можно сравнивать?  
  
Можно сравнивать WebDriver с драйверами, которые включены в состав различных инструментов. Например, можно сравнить:

* какие браузеры и какие версии браузеров поддерживаются, в том числе мобильные,
* какие операционные системы поддерживаются, в том числе мобильные,
* можно ли управлять одновременно несколькими браузерами на одной машине, не возникает ли конфликтов,
* можно ли управлять браузером на удалённой машине,
* какие действия в браузере можно выполнять,
* какие данные из браузера можно получать,
* насколько точно драйвер эмулирует действия пользователя, то есть генерирует ли он все те же самые события в браузере, которые возникают при работе настоящего пользователя,
* можно ли работать с диалоговыми окнами (alert, prompt),
* можно ли работать с «нативными» окнами (диалог загрузки файлов),
* можно ли работать с HTTPS-протоколами и сертификатами,
* и так далее.

И вот тут WebDriver оказывается бесспорным лидером. Впрочем, само сравнение WebDriver с чем бы то ни было выходит за рамки этой статьи.  
  
Что касается сравнения с «комплексным» инструментами типа TestComplete или Sahi, для этого нужно брать не WebDriver, а полный стек.  
  
Например, стек для технологии Java может быть таким: Jenkins + Maven + Thucydices + JUnit+ WebDriver. К этому добавляются ещё все возможности языка программирования Java, плюс масса плагинов для Maven и Jenkins, а чтобы совсем всё было круто – можно запускать тесты в облаках, используя какой-нибудь сервис типа [SauceLabs](https://saucelabs.com/).  
  
Вот тогда сравнение будет интересным. Но это уже заслуга не только WebDriver, важен весь стек, а не только драйвер браузера. Что касается WebDriver, стоит отметить лишь то, что он прекрасно встраивается практически в любой стек, это одно из его достоинств как «независимого» драйвера.  
  
Разумеется, WebDriver может использоваться не только при тестировании. Ему вообще безразлично, кто и зачем хочет управлять браузером. Вы можете автоматизировать какие-то рутинные задачи. Можете сделать ботов, которые будут флудить в форумах. Можете сделать скрипт, который автоматически снимает скриншоты для документации. Всё что угодно. Драйверу всё равно. Он всего лишь предоставляет доступ к браузеру.  
  
Кроме того, какой бы инструмент вы ни использовали – вполне возможно, что к нему удастся подключить WebDriver, который имеет реализации на самых разных языках – Java, C#, Ruby, Python. И тогда вы в дополнение ко всем возможностям вашего любимого инструмента добавите все достоинства WebDriver. Это стоит потраченных усилий, потому что среди драйверов на данный момент он лучший.  
  
Ну да, я уже несколько раз повторил, что «он лучший», но при этом не привёл сравнения с другими драйверами. И не буду. Потому что есть аргумент, который в перспективе важнее любых сравнений.

## Selenium WebDriver – это спецификация интерфейса для управления браузером

Самое главное отличие WebDriver от всех остальных драйверов заключается в том, что это «стандартный» драйвер, а все остальные – «нестандартные».  
  
И это не простая фигура речи.  
  
Организация W3C действительно приняла WebDriver за основу при разработке [стандарта интерфейса для управления браузером](http://www.w3.org/TR/2012/WD-webdriver-20120710/). Сейчас он находится в состоянии публичного рассмотрения.  
  
Через год-полтора этот стандарт будет утверждён. И тогда реализация интерфейса WebDriver будет возложена на производителей браузеров, а WebDriver как независимый драйвер, возможно, в будущем исчезнет совсем, потому что он будет встроен непосредственно в браузеры.  
  
Таким образом, можно сказать, что Selenium WebDriver это вообще не инструмент, а спецификация, документ, стандарт, описывающий, какой интерфейс браузеры должны предоставлять наружу, чтобы через этот интерфейс можно было браузером управлять.  
  
Пока стандарт обсуждается, производители браузеров уже действуют. В рамках проекта Selenium было разработано несколько референсных реализаций для различных браузеров, но постепенно эта деятельность переходит в ведение производителей браузеров. Драйвер для браузера Chrome [разрабатывается в рамках проекта Chromium](http://code.google.com/p/chromedriver/), его делает та же команда, которая занимается разработкой самого браузера. Драйвер для браузера Opera [разрабатывается в компании Opera Software](http://www.opera.com/developer/tools/operadriver/). Драйвер для браузера Firefox пока разрабатывается участниками проекта Selenium, но в недрах компании Mozilla уже готовится ему замена, которая носит кодовое название [Marionette](https://developer.mozilla.org/en-US/docs/Marionette). Этот новый драйвер для Firefox уже доступен в девелоперских сборках браузера. На очереди Internet Explorer и Safari, к их разработке сотрудники соответствующих компаний пока не подключились, но кое-какие сдвиги в этом направлении есть, потому что стандарт (даже будущий) обязывает.  
  
В общем, можно сказать, что Selenium это единственный проект по созданию средств автоматизации управления браузерами, в котором участвуют непосредственно компании, разрабатывающие браузеры. Это одна из ключевых причин его успеха.  
  
А что случится после того, как во всех браузерах будет реализован этот стандарт?  
  
Было бы логично ожидать, что производители инструментов тестирования не станут изобретать велосипеды, а будут управлять браузером через стандартный интерфейс. Можно сказать, что все инструменты станут использовать WebDriver для взаимодействия с браузером. Но это будет уже не Selenium WebDriver как независимый драйвер, а Selenium WebDriver как спецификация интерфейса.

## Так почему же у него такой примитивный интерфейс?

Именно потому, что WebDriver – это:

* драйвер браузера, то есть библиотека достаточно низкого уровня абстракции,
* стандарт на интерфейс управления браузером, то есть минимальный набор команд, который должен быть реализован в каждом браузере.

При разработке Selenium WebDriver изначально была поставлена цель – не включать в него ничего лишнего. Стандартный интерфейс управления браузером должен быть простым и стабильным.  
  
Набор команд последовательно сокращался, были выброшены такие «повышающие удобство использования» команды как check, uncheck (для чекбоксов), select (для выпадающих списков). Все они сводятся к более простой команде click и поэтому они лишние. Сейчас в интерфейсе WebDriver осталась только одна избыточная команда – это submit, но может быть когда-нибудь и она будет устранена.  
  
Кроме того, структура интерфейса проектировалась таким образом, чтобы можно было описать его на языке IDL (именно это сделано в стандарте W3C) и сделать реализации на различных языках программирования. Поэтому использовался минимум языковых идиом, минимум «скрытых» переменных, интерфейс «тупой и прямолинейный».  
  
Но зато благодаря этой примитивности интерфейса сейчас для интерфейса WebDriver имеются реализации клиентских библиотек на Java, C#, Ruby, Python, JavaScript, PHP, Perl и даже Haskell!  
  
И благодаря той же самой простоте WebDriver прекрасно интегрируется с любыми другими инструментами, встраивается в любой стек. В этом секрет его популярности и быстрого распространения – он не пытается «победить» другие инструменты, вместо этого он интегрируется с ними.

## А как же удобство использования?

Эту задачу должны решать расширения, построенные на базе Selenium WebDriver. Именно они должны предоставлять расширенный набор команд, реализуя эти команды через примитивный интерфейс WebDriver. В дистрибутиве Selenium имеется класс Select, предназначенный для работы с выпадающими списками, который является наглядной демонстрацией того, как должны строиться расширения.  
  
Постепенно появляются библиотеки, которые строятся на базе Selenium WebDriver и предоставляют более высокий уровень абстракции: [Selenide](https://github.com/codeborne/selenide), [fluent-selenium](https://github.com/SeleniumHQ/fluent-selenium), [watir-webdriver](http://watirwebdriver.com/), [Thucidides](http://www.thucydides.info/). Популярные фреймворки для проектирования тестов позволяют наряду с другими драйверами использовать WebDriver. Среди таких фреймворков можно упомянуть [Robot Framework](http://code.google.com/p/robotframework/), [Capybara](http://jnicklas.github.com/capybara/) и тот же [Thucidides](http://www.thucydides.info/).  
  
Рано или поздно должны появиться вспомогательные библиотеки, облегчающие работу с теми или иными наборами виджетов – jQuery, Prototype, ExtJS, GWT и прочими.  
  
Число таких расширений и инструментов будет расти, сложность тоже. Так что вскоре может так случиться, что вы, используя какой-то инструмент, будете выполнять тесты, даже не подозревая о том, что взаимодействие с браузером осуществляется через драйвер Selenium WebDriver.

## Стоит ли тогда вообще изучать Selenium?

Может быть лучше изучать эти библиотеки и инструменты более высокого уровня?  
  
Чтобы ответить на этот вопрос, я сформулирую его иначе: кому и зачем стоит изучать Selenium, а кому лучше использовать более высокоуровневые библиотеки и инструменты?

1. Какой бы инструмент вы ни использовали, вам нужно выбрать драйвер, управляющий браузером. Чтобы его выбрать, вы должны знать возможности драйвера – что он может, а чего не может. На этом уровне Selenium необходимо освоить каждому специалисту по автоматизации. При этом конкретно интерфейс WebDriver, если вы с ним работаете, изучать нет необходимости.
2. Простой набор команд выучить проще, чем «расширенный», то есть Selenium освоить проще, чем его расширение. У этого явления есть и обратная сторона – если вы изучили расширенный набор команд, то внезапно оказывается, что набор команд WebDriver вы при этом тоже освоили.
3. Расширения, как правило, языково-зависимые, потому что добавление удобства предполагает использование языковых идиом, типичных приёмов организации кода на том или ином языке программирования. Базовый интерфейс WebDriver простой, поэтому освоив его, вы сможете использовать его на любом языке, он будет выглядеть практически одинаково.
4. Большинство библиотек, нацеленных на повышение удобства интерфейса, улучшают средства поиска элементов – дополнительные типы локаторов, более удобный способ описания локаторов и так далее. Примитивы, соответствующие действиям пользователя, в WebDriver уже и так достаточно хороши. Хотя, конечно, библиотеки будут реализовывать типовые «связки», то есть последовательности этих действий, аналогично тому, как это сделано в классе Select для выпадающих списков.
5. Если вы используете «таблички» для описания тестов (как в Robot Framework) или специальный язык для описания на уровне предметной области (DSL, Domain Specific Language) – вам нет необходимости знать о примитивах WebDriver. Но если вы реализуете «фикстуры» для тестов, описываете действия, которыми можно будет оперировать в табличках, реализуете DSL – вам придётся работать непосредственно с WebDriver, либо с каким-то его расширением, но не слишком высокоуровневым.
6. И самый последний аргумент, который, я надеюсь, со временем будет становиться всё менее актуальным – увы, пока хороших расширений катастрофически не хватает. Они обязательно появятся. Может быть, именно вы реализуете одно из таких расширений. Для этого вам понадобиться изучить интерфейс WebDriver. И те, кто будут пользоваться плодами вашего труда, смогут работать с более высокоуровневой библиотекой. А пока приходится использовать непосредственно WebDriver с небольшими надстройками над ним.

**11. Cucumber**К сожалению, нет магической формулы для разработки высококачественного программного обеспечения, но очевидно, что тестирование улучшает его качество, а автоматизация тестирования улучшает качество самого тестирования.   
  
В данной статье мы рассмотрим один из самых популярных фреймворков для автоматизации тестирования с использованием BDD-подхода – Cucumber. Также посмотрим, как он работает и какие средства предоставляет.   
  
Первоначально Cucumber был разработан Ruby-сообществом, но со временем был адаптирован и для других популярных языков программирования. В данной статье рассмотрим работу Cucumber на языке Java.

## Gherkin

BDD тесты – это простой текст, на человеческом языке, написанный в форме истории (сценария), описывающей некоторое поведение.  
  
В Cucumber для написания тестов используется Gherkin-нотация, которая определяет структуру теста и набор ключевых слов. Тест записывается в файл с расширением \*.feature и может содержать как один, так и более сценариев.  
  
Рассмотрим пример теста на русском языке с использованием Gherkin:

# language: ru

@all

Функция: Аутентификация банковской карты

Банкомат должен спросить у пользователя PIN-код банковской карты

Банкомат должен выдать предупреждение, если пользователь ввел неправильный PIN-код

Аутентификация успешна, если пользователь ввел правильный PIN-код

Предыстория:

Допустим пользователь вставляет в банкомат банковскую карту

И банкомат выдает сообщение о необходимости ввода PIN-кода

@correct

Сценарий: Успешная аутентификация

Если пользователь вводит корректный PIN-код

То банкомат отображает меню и количество доступных денег на счету

@fail

Сценарий: Некорректная аутентификация

Если пользователь вводит некорректный PIN-код

То банкомат выдает сообщение, что введённый PIN-код неверный

Как видно из примера, сценарии описаны на простом нетехническом языке, благодаря чему, понимать и писать их может любой участник проекта.   
  
Обратите внимание на структуру сценария:  
  
1. Получить начальное состояние системы;  
2. Что-то сделать;  
3. Получить новое состояние системы.  
  
В примере жирным выделены ключевые слова. Ниже представлен полный список ключевых слов на русском языке:

1. **Дано**, **Допустим**, **Пусть** – используются для описания предварительного, ранее известного состояния;
2. **Когда**, **Если**– используются для описания ключевых действий;
3. **И**, **К тому же**, **Также**– используются для описания дополнительных предусловий или действий;
4. **Тогда**, **То** – используются для описания ожидаемого результата выполненного действия;
5. **Но**, **А**– используются для описания дополнительного ожидаемого результата;
6. **Функция**, **Функционал**, **Свойство**– используется для именования и описания тестируемого функционала. Описание может быть многострочным;
7. **Сценарий**– используется для обозначения сценария;
8. **Предыстория**, **Контекст**– используется для описания действий, выполняемых перед каждым сценарием в файле;
9. **Структура сценария**, **Примеры**– используется для создания шаблона сценария и таблицы параметров, передаваемых в него.

Ключевые слова, перечисленные в пунктах 1-5, используются для описания шагов сценария, Cucumber их технически не различает. Вместо них можно использовать символ **\***, но делать так не рекомендуется. У этих слов есть определенная цель, и они были выбраны именно для неё.  
  
Список зарезервированных символов:  
  
**#** – обозначает комментарии;  
**@** – тэгирует сценарии или функционал;  
**|** – разделяет данные в табличном формате;  
**"""** – обрамляет многострочные данные.  
  
Сценарий начинается со строки # language: ru. Эта строчка указывает Cucumber, что в сценарии используется русский язык. Если её не указать, фреймворк, встретив в сценарии русский текст, выбросит исключение LexingError и тест не запустится. По умолчанию используется английский язык.

## Простой проект

Cucumber-проект состоит из двух частей – это текстовые файлы с описанием сценариев (\*.feature) и файлы с реализацией шагов на языке программирования (в нашем случае — файлы \*.java).   
  
Для создания проекта будем использовать систему автоматизации сборки проектов Apache Maven.  
Первым делом добавим cucumber в зависимости Maven:

<dependency>

<groupId>info.cukes</groupId>

<artifactId>cucumber-java</artifactId>

<version>1.2.4</version>

</dependency>

Для запуска тестов будем использовать JUnit (возможен запуск через TestNG), для этого добавим еще две зависимости:

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

<dependency>

<groupId>info.cukes</groupId>

<artifactId>cucumber-junit</artifactId>

<version>1.2.4</version>

</dependency>

Библиотека cucumber-junit содержит класс cucumber.api.junit.Cucumber, который позволяет запускать тесты, используя JUnit аннотацию RunWith. Класс, указанный в этой аннотации, определяет каким образом запускать тесты.  
  
Создадим класс, который будет являться точкой входа для наших тестов.

import cucumber.api.CucumberOptions;

import cucumber.api.SnippetType;

import cucumber.api.junit.Cucumber;

import org.junit.runner.RunWith;

@RunWith(Cucumber.class)

@CucumberOptions(

features = "src/test/features",

glue = "ru.savkk.test",

tags = "@all",

dryRun = false,

strict = false,

snippets = SnippetType.UNDERSCORE,

*// name = "^Успешное|Успешная.\*"*

)

public class RunnerTest {

}

Обратите внимание, название класса должно иметь окончание Test, иначе тесты не будут запускаться.  
  
Рассмотрим опции Cucumber:

1. features – путь к папке с .feature файлами. Фреймворк будет искать файлы в этой и во всех дочерних папках. Можно указать несколько папок, например: features = {«src/test/features», «src/test/feat»};
2. glue – пакет, в котором находятся классы с реализацией шагов и «хуков». Можно указать несколько пакетов, например, так: glue = {«ru.savkk.test», «ru.savkk.hooks»};
3. tags – фильтр запускаемых тестов по тэгам. Список тэгов можно перечислить через запятую. Символ ~ исключает тест из списка запускаемых тестов, например ~@fail;
4. dryRun – если true, то сразу после запуска теста фреймворк проверяет, все ли шаги теста разработаны, если нет, то выдает предупреждение. При false предупреждение будет выдаваться по достижении неразработанного шага. По умолчанию false.
5. strict – если true, то при встрече неразработанного шага тест остановится с ошибкой. False — неразработанные шаги пропускаются. По умолчанию false.
6. snippets – указывает в каком формате фреймворк будет предлагать шаблон для нереализованных шагов. Доступны значения: SnippetType.CAMELCASE, SnippetType.UNDERSCORE.
7. name – фильтрует запускаемые тесты по названиям удовлетворяющим регулярному выражению.

Для фильтрации запускаемых тестов нельзя одновременно использовать опции tags и name.  
  
**Создание «фичи»**  
  
В папке src/test/features создадим файл с описание тестируемого функционала. Опишем два простых сценария снятия денег со счета — успешный и провальный.

# language: ru

@withdrawal

Функция: Снятие денег со счета

@success

Сценарий: Успешное снятие денег со счета

Дано на счете пользователя имеется 120000 рублей

Когда пользователь снимает со счета 20000 рублей

Тогда на счете пользователя имеется 100000 рублей

@fail

Сценарий: Снятие денег со счета - недостаточно денег

Дано на счете пользователя имеется 100 рублей

Когда пользователь снимает со счета 120 рублей

Тогда появляется предупреждение "На счете недостаточно денег"

**Запускаем**  
  
Попробуем запустить RunnerTest со следующими настройками:

@RunWith(Cucumber.class)

@CucumberOptions(

features = "src/test/features",

glue = "ru.savkk.test",

tags = "@withdrawal",

snippets = SnippetType.CAMELCASE

)

public class RunnerTest {

}

В консоль появился результат прохождения теста:

Undefined scenarios:

test.feature:6 # Сценарий: Успешное снятие денег со счета

test.feature:12 # Сценарий: Снятие денег со счета - недостаточно денег

2 Scenarios (2 undefined)

6 Steps (6 undefined)

0m0,000s

You can implement missing steps with the snippets below:

@Дано("^на счете пользователя имеется (\\d+) рублей$")

public void наСчетеПользователяИмеетсяРублей(int arg1) throws Throwable {

*// Write code here that turns the phrase above into concrete actions*

throw new PendingException();

}

@Когда("^пользователь снимает со счета (\\d+) рублей$")

public void пользовательСнимаетСоСчетаРублей(int arg1) throws Throwable {

*// Write code here that turns the phrase above into concrete actions*

throw new PendingException();

}

@Тогда("^появляется предупреждение \"([^\"]\*)\"$")

public void появляетсяПредупреждение(String arg1) throws Throwable {

*// Write code here that turns the phrase above into concrete actions*

throw new PendingException();

}

Cucumber не нашел реализацию шагов и предложил свои шаблоны для разработки.  
Создадим класс MyStepdefs в пакете ru.savkk.test и перенесем в него методы, предложенные фреймворком:

import cucumber.api.PendingException;

import cucumber.api.java.ru.\*;

public class MyStepdefs {

@Дано("^на счете пользователя имеется (\\d+) рублей$")

public void наСчетеПользователяИмеетсяРублей(int arg1) throws Throwable {

*// Write code here that turns the phrase above into concrete actions*

throw new PendingException();

}

@Когда("^пользователь снимает со счета (\\d+) рублей$")

public void пользовательСнимаетСоСчетаРублей(int arg1) throws Throwable {

*// Write code here that turns the phrase above into concrete actions*

throw new PendingException();

}

@Тогда("^появляется предупреждение \"([^\"]\*)\"$")

public void появляетсяПредупреждение(String arg1) throws Throwable {

*// Write code here that turns the phrase above into concrete actions*

throw new PendingException();

}

}

При запуске теста Cucumber проходит по сценарию шаг за шагом. Взяв шаг, он отделяет ключевое слово от описания шага и пытается найти в Java-классах пакета указанного в опции glue аннотацию с регулярным выражением, подходящим описанию. Найдя совпадение, фреймворк вызывает метод с найденной аннотацией. Если несколько регулярных выражений удовлетворяют описанию шага, фреймворк выбрасывает ошибку.  
  
Как было сказано выше, для Cucumber технически нет отличия в ключевых словах, описывающих шаги, это верно и для аннотации, например:

@Когда("^пользователь снимает со счета (\\d+) рублей$")

и

@Тогда("^пользователь снимает со счета (\\d+) рублей$")

для фреймворка являются одинаковыми.  
  
То, что в регулярных выражениях записано в скобках передается в метод в виде аргумента. Фреймворк самостоятельно определяет, что необходимо передавать из сценария в метод в виде аргумента. Это числа — (\\d+). И текст, экранированный в кавычки — \"([^\"]\*)\". Это самые распространённые из передаваемых аргументов.  
  
Ниже в таблице представлены элементы, используемые в регулярных выражениях:  
регулярных выражениях:

|  |  |  |
| --- | --- | --- |
| Выражение | Описание | Соответствие |
| . | Один любой символ (за исключение переноса строки) | Ф 2 j |
| .\* | 0 или больше любых символов (за исключением переноса строки) | Abracadabra 789-160-87   , |
| .+ | Один или больше любых символов (за исключением переноса строки) | Все, что относилось к предыдущему, за исключением пустой строки. |
| .{2} | Любые два символа (за исключением переноса строки) | Фф 22 $х JJ |
| .{1,3} | От одного до трех любых символов (за исключением переноса строки) | Жжж Уу ! |
| ^ | Якорь начала строки | ^aaa соответствует aaa ^aaa соответствует aaabbb ^aaa не соответствует bbbaaa |
| $ | Якорь конца строки | aaa$ соответствует aaa aaa$ не соответствует aaabbb aaa$ соответствует bbbaaa |
| \d\* [0-9]\* | Любое число (или ничего) | 12321  5323 |
| \d+ [0-9]+ | Любое число | Все, что относилось к предыдущему, за исключением пустой строки. |
| \w\* | Любая буква, цифра или нижнее подчеркивание (или ничего) | \_we \_1ee Gfd4 |
| \s | Пробел, табуляция или перенос строки | \t, \r или \n |
| "[^\"]\*" | Любой символ (или ничего) в кавычках | "aaa" "" "3213dsa" |
| ? | Делает символ или группу символов необязательными | abc? соответствует ab или abc, но не b или bc |
| | | Логическое ИЛИ | aaa|bbb соответствует aaa или bbb, но не aaabbb |
| () | Группа. В Cucumber группа передается в определение шага в виде аргумента. | (\d+) рублей соответствует 10 рублей, при этом 10 передается в метод шага в виде аргумента |
| (?: ) | Не передаваемая группа. Cucumber не воспринимает группу как аргумент. | (\d+) (?:рублей|рубля) соответствует 3 рубля, при этом 3 передается в метод, а «рубля» - нет. |

## Передача коллекций в аргументы

Часто возникает ситуация, когда из сценария в метод необходимо передать набор однотипных данных – коллекций. Для подобной задачи в Cucumber есть несколько решений:

1. Фреймворк по умолчанию оборачивает данные, перечисленные через запятую, в ArrayList:

Дано в меню доступны пункты Файл, Редактировать, О программе

@Дано("^в меню доступны пункты (.\*)$")

public void вМенюДоступныПункты(List<String> arg) {

*// что-то сделать*

}

Для замены разделителя, можно воспользоваться аннотацией Delimiter:

Дано в меню доступны пункты Файл и Редактировать и О программе

@Дано("^в меню доступны пункты (.+)$")

public void вМенюДоступныПункты(@Delimiter(" и ") List<String> arg) {

*// что-то сделать*

}

1. Данные, записанные в виде таблицы с одной колонкой, Cucumber также может обернуть в ArrayList:
2. Дано в меню доступны пункты
3. | Файл |
4. | Редактировать |

| О программе |

@Дано("^в меню доступны пункты$")

public void вМенюДоступныПункты(List<String> arg) {

*// что-то сделать*

}

1. Данные, записанные в таблицу с двумя колонками, Cucumber может обернуть в ассоциативный массив, где данные из первой колонки – это ключ, а из второй – данные:
2. Дано в меню доступны пункты
3. | Файл | true |
4. | Редактировать | false |

| О программе | true |

public void вМенюДоступныПункты(Map<String, Boolean> arg) {

*// что-то сделать*

}

1. Передача данных в виде таблицы с большим количеством колонок возможна двумя способами:  
     
   * DataTable
   * Дано в меню доступны пункты
   * | Файл | true | 5 |
   * | Редактировать | false | 8 |

| О программе | true | 2 |

@Дано("^в меню доступны пункты$")

public void вМенюДоступныПункты(DataTable arg) {

*// что-то сделать*

}

DataTable – это класс, который эмулирует табличное представление данных. Для доступа к данным в нем имеется большое количество методов. Рассмотрим некоторые из них:

public <K,V> List<Map<K,V>> asMaps(Class<K> keyType,Class<V> valueType)

Конвертирует таблицу в список ассоциативных массивов. Первая строка таблицы используется для именования ключей, остальные как значения:

Дано в меню доступны пункты

| Название | Доступен | Количество подменю |

| Файл | true | 5 |

| Редактировать | false | 8 |

| О программе | true | 2 |

@Дано("^в меню доступны пункты$")

public void вМенюДоступныПункты(DataTable arg) {

List<Map<String, String>> table = arg.asMaps(String.class, String.class);

System.out.println(table.get(0).get("Название"));

System.out.println(table.get(1).get("Название"));

System.out.println(table.get(2).get("Название"));

}

Данный пример выведет на консоль:  
  
Файл  
Редактировать  
О программе

public <T> List<List<T>> asLists(Class<T> itemType)

Метод преобразует таблицу в список списков:

Дано в меню доступны пункты

| Файл | true | 5 |

| Редактировать | false | 8 |

| О программе | true | 2 |

@Дано("^в меню доступны пункты$")

public void вМенюДоступныПункты(DataTable arg) {

List<List<String>> table = arg.asLists(String.class);

System.out.print(table.get(0).get(0) + " ");

System.out.print(table.get(0).get(1) + " ");

System.out.println(table.get(0).get(2) + " ");

System.out.print(table.get(1).get(0) + " ");

System.out.print(table.get(1).get(1) + " ");

System.out.println(table.get(1).get(2) + " ");

}

На консоль будет выведено:  
  
Файл true 5   
Редактировать false 8

public List<List<String>> cells(int firstRow)

Этот метод делает то же, что и предыдущий метод, за исключением того, что нельзя определить какого типа данные находятся в таблице, всегда возвращает список строк – List. В качестве аргумента метод принимает номер первой строки:

Дано в меню доступны пункты

| Файл | true | 5 |

| Редактировать | false | 8 |

| О программе | true | 2 |

@Дано("^в меню доступны пункты$")

public void вМенюДоступныПункты(DataTable arg) {

List<List<String>> table = arg.cells(1);

System.out.print(table.get(0).get(0) + " ");

System.out.print(table.get(0).get(1) + " ");

System.out.println(table.get(0).get(2) + " ");

System.out.print(table.get(1).get(0) + " ");

System.out.print(table.get(1).get(1) + " ");

System.out.println(table.get(1).get(2) + " ");

}

Метод выведет на консоль:  
  
Редактировать false 8   
О программе true 2

* + Class   
    Cucumber может создать объекты из табличных данных, переданных из сценария. Существует два способа это сделать.   
      
    Создадим для примера класс Menu:
  + public class Menu {
  + private String title;
  + private boolean isAvailable;
  + private int subMenuCount;
  + public String getTitle() {
  + return title;
  + }
  + public boolean getAvailable() {
  + return isAvailable;
  + }
  + public int getSubMenuCount() {
  + return subMenuCount;
  + }

}

Для первого способа шаг в сценарии запишем в следующем виде:

Дано в меню доступны пункты

| title | isAvailable | subMenuCount |

| Файл | true | 5 |

| Редактировать | false | 8 |

| О программе | true | 2 |

Реализация:

@Дано("^в меню доступны пункты$")

public void вМенюДоступныПункты(List<Menu> arg) {

for (int i = 0; i < arg.size(); i++) {

System.out.print(arg.get(i).getTitle() + " ");

System.out.print(Boolean.toString(arg.get(i).getAvailable()) + " ");

System.out.println(Integer.toString(arg.get(i).getSubMenuCount()));

}

}

Вывод в консоль:  
  
Файл true 5  
Редактировать false 8  
О программе true 2  
  
Фреймворк создает связанный список объектов из таблицы с тремя колонками. В первой строке таблицы должны быть указаны наименования полей класса, создаваемого объекта. Если какое-то поле не указать, оно не будет инициализировано.  
  
Для второго способа приведем шаг сценария к следующему виду:

Дано в меню доступны пункты

| title | Файл | Редактировать | О программе |

| isAvailable | true | false | true |

| subMenuCount | 5 | 8 | 2 |

А в аргументе описания шага используем аннотацию @Transpose.

@Дано("^в меню доступны пункты$")

public void вМенюДоступныПункты(@Transpose List<Menu> arg) {

*// что-то сделать*

}

Cucumber, как и в предыдущем примере, создаст связанный список объектов, но, в данном случае, наименования полей записывается в первой колонке таблицы.

1. Многострочные аргументы  
     
   Для передачи многострочных данных в аргумент метода, их необходимо экранировать тремя двойными кавычками:
2. Тогда отображается форма с текстом
3. """
4. На ваш номер телефона был выслан одноразовый пароль.
5. Для подтверждения платежа необходимо ввести полученный
6. одноразовый пароль.

"""

Данные в метод приходят в виде объекта класса String:

@Тогда("^отображается форма с текстом$")

public void отображаетсяФормаСТекстом(String expectedText) {

*// что-то сделать*

}

## Date

Фреймворк самостоятельно приводит данные из сценария к типу данных, указанному в аргументе метода. Если это невозможно, то выбрасывает исключение ConversionException. Это справедливо и для классов Date и Calendar. Рассмотрим пример:

Дано дата создания документа 04.05.2017

@Дано("^дата создания документа (.+)$")

public void датаСозданияДокумента (Date arg) {

*// что-то сделать*

}

Все прекрасно сработало, Cucumber преобразовал 04.05.2017 в объект класса Date со значением «Thu May 04 00:00:00 EET 2017».  
  
Рассмотрим еще один пример:

Дано дата создания документа 04-05-2017

@Дано("^дата создания документа (.+)$")

public void датаСозданияДокумента (Date arg) {

*// что-то сделать*

}

Дойдя до этого шага, Cucumber выбросил исключение:

cucumber.deps.com.thoughtworks.xstream.converters.ConversionException: Couldn't convert "04-05-2017" to an instance of: [class java.util.Date]

Почему первый пример сработал, а второй нет?  
  
Дело в том, что в Cucumber встроена поддержка форматов дат чувствительных к текущей локали. Если необходимо записать дату в формате, отличающемся от формата текущей локали, нужно использовать аннотацию Format:

Дано дата создания документа 04-05-2017

@Дано("^дата создания документа (.+)$")

public void датаСозданияДокумента (@Format("dd-MM-yyyy") Date arg) {

*// что-то сделать*

}

## Структура сценария

Бывают случаи, когда необходимо запустить тест несколько раз с различным набором данных, в таких случая на помощь приходит конструкция «Структура сценария»:

*# language: ru*

@withdrawal

Функция: Снятие денег со счета

@success

Структура сценария: Успешное снятие денег со счета

Дано на счете пользователя имеется <изначально> рублей

Когда пользователь снимает со счета <снято> рублей

Тогда на счете пользователя имеется <осталось> рублей

Примеры:

| изначально | снято | осталось |

| 10000 | 1 | 9999 |

| 9999 | 9999 | 0 |

Суть данной конструкции заключается в том, что в места, обозначенные символами <>, вставляются данные из таблицы Примеры. Тест будет запускаться поочередно для каждой строки из данной таблицы. Названия колонок должно совпадать с названием мест вставки данных.

## Использование хуков

Cucumber поддерживает хуки (hooks) – методы, запускаемые до или после сценария. Для их обозначения используется аннотация Before и After. Класс с хуками должен находиться в пакете, указанном в опциях фреймворка. Пример класса с хуками:

import cucumber.api.java.After;

import cucumber.api.java.Before;

public class Hooks {

@Before

public void prepareData() {

*//подготовить данные*

}

@After

public void clearData() {

*//очистить данные*

}

}

Метод c аннотацией Before будет запускаться перед каждым сценарием, After – после.

### Порядок выполнения

Хукам можно задать порядок, в котором они будут выполняться. Для этого необходимо в аннотации указать параметр order. По умолчанию значение order равно 10000.   
  
Для Before чем меньше это значение, тем раньше выполнится метод:

@Before(order = 10)

public void connectToServer() {

*//подключиться к серверу*

}

@Before(order = 20)

public void prepareData() {

*//подготовить данные*

}

В данном примере первым выполнится метод connectToServer(), затем prepareData().  
  
After отрабатывает в обратном порядке.

### Тэгирование

В параметре value можно указать тэги сценариев, для которых будут отрабатывать хуки. Символ ~ означает «за исключением». Пример:

@Before(value = "@correct", order = 30)

public void connectToServer() {

*//сделай что-нибудь*

}

@Before(value = "~@fail", order = 20)

public void prepareData() {

*//сделай что-нибудь*

}

Метод connectToServer будет выполнен для всех сценариев с тэгом [correct](https://habrahabr.ru/users/correct/), метод prepareData для всех сценариев за исключением сценариев с тэгом [fail](https://habrahabr.ru/users/fail/).

### Scenario class

Если в определении метода-хука в аргументе указать объект класса Scenario, то в данном методе можно будет узнать много полезной информации о запущенном сценарии, например:

@After

public void getScenarioInfo(Scenario scenario) {

System.out.println(scenario.getId());

System.out.println(scenario.getName());

System.out.println(scenario.getStatus());

System.out.println(scenario.isFailed());

System.out.println(scenario.getSourceTagNames());

}

Для сценария:

# language: ru

@all

Функция: Аутентификация банковской карты

Банкомат должен спросить у пользователя PIN-код банковской карты

Банкомат должен выдать предупреждение если пользователь ввел неправильный PIN-код

Аутентификация успешна если пользователь ввел правильный PIN-код

Предыстория:

Допустим пользователь вставляет в банкомат банковскую карту

И банкомат выдает сообщение о необходимости ввода PIN-кода

@correct

Сценарий: Успешная аутентификация

Если пользователь вводит корректный PIN-код

То банкомат отображает меню и количество доступных денег на счету

Выведет в консоль:  
  
аутентификация-банковской-карты;успешная-аутентификация  
Успешная аутентификация  
passed  
false  
[@correct, @all]

## В заключение

Cucumber — очень мощный и гибкий фреймворк, который можно использовать в связке со многими другими популярными инструментами. Например, с Selenium – фреймворком для автоматизации веб-приложений, или Yandex.Allure – библиотекой, позволяющей создавать удобные отчеты.  
Всем удачи в автоматизации.

**12. Page Object  
  
Page Object** – это шаблон проектирования, который широко используется в автоматизированном тестировании и позволяет разделять логику выполнения тестов от их реализации. Page Object как бы моделирует страницы тестируемого приложения в качестве объектов в коде. В результате его использования у вас получатся отдельные классы, отвечающие за работу с HTML каждой конкретной веб-страницы. Такой подход значительно уменьшает объем повторяющегося кода, потому что одни и те же объекты страниц можно использовать в различных тестах. Основное преимущество Page Object заключается в том, что в случае изменения пользовательского интерфейса, можно выполнить исправление только в одном месте, а не исправлять каждый тест, в котором этот интерфейс используется.  
Класс PageObject не обязательно должен представлять собой всю страницу. Он может быть частью страницы, которая часто используется на сайте (или даже на одной странице). Это может быть, например, пагинатор или меню. Основной принцип состоит в том, что есть только один класс в Вашем проекте, который знает о структуре HTML конкретной страницы или ее части.

Разделение логики и реализации  
Существует большая разница между логикой тестирования (что проверить) и его реализацией (как проверить). Пример тестового сценария: «Пользователь вводит неверный логин или пароль, нажимает кнопку входа, получает сообщение об ошибке». Этот сценарий описывает логику теста, в то время как реализация содержит в себе такие действия как поиск полей ввода на странице, их заполнение, проверку полученной ошибки и т.д. И если, например, измениться способ вывода сообщения об ошибке, то это никак не повлияет на сценарий теста, все также нужно будет ввести неверные данные, нажать кнопку входа и проверить ошибку. Но это напрямую затронет реализацию теста — необходимо будет изменить метод получающий и обрабатывающий сообщение об ошибке. При разделении логики теста от его реализации автотесты становятся более гибкими и их, как правило, легче поддерживать.

Page Object в Selenium  
Паттерн Page Object в Selenium реализован с помощью библиотеки PageFactory и класса страницы. Page Object представляет собой отдельный класс, содержащий локаторы элементов, методы для работы с ними и конструктор принимающий в качестве параметра объект WebDriver. Методы класса Page Object могут возвращать объекты других Page Object классов. С помощью этого можно воссоздать копию переходов и поведения веб-приложения. Например, метод успешной регистрации в классе RegistrationPage должен возвращать экземпляр HomePage, потому что после регистрации на сайте пользователя перенаправляет на домашнюю страницу. Одним из следствий такого подхода является то, что необходимо моделировать как успешные, так и неуспешные методы. Или, например, в случае если нажатие на элемент может открывать различные страницы в зависимости от условий, то также необходимо создавать разные методы для каждого необходимого случая:

|  |
| --- |
| public class RegistrationPage {        private WebDriver driver;        public RegistrationPage(WebDriver driver) {          this.driver = driver;      }        public HomePage registerUserSuccess(User user) {          // успешная регистрация и переход на домашнюю страницу      }        public RegistrationPage registerUserError(User user) {          // регистрация пользователя с неверно заполненными полями          // вывод ошибки, остаемся на той же странице      }  } |

Поиск элементов на странице можно осуществлять в методах используя driver.findElement, а можно объявить элементы в классе декларативно, используя аннотацию @FindBy.

|  |
| --- |
| import java.util.List;    import org.openqa.selenium.WebDriver;  import org.openqa.selenium.WebElement;  import org.openqa.selenium.support.FindBy;  import org.openqa.selenium.support.PageFactory;    public class HomePage {        private WebDriver driver;        /\*\*       \* Имя пользователя       \*/      @FindBy(id = "profilename")      private WebElement userName;        /\*\*       \* Выход       \*/      @FindBy(linkText = "Выход")      private WebElement exit;        /\*\*       \* Элементы меню       \*/      @FindBy(className = "menu\_item")      private List<WebElement> menuItems;        public HomePage(WebDriver driver) {          PageFactory.initElements(driver, this);          this.driver = driver;      }  } |

@FindBy в качестве параметров принимает те же механизмы поиска, что и методы класса By.  
Аннотация @FindBy работает только с использованием PageFactory. Без вызова PageFactory.initElements(driver, this); при обращении к элементам Вы получите NullPointerException. PageFactory инициализирует элементы при помощи вызова поиска driver.findElement. Инициализация веб-элементов происходит на странице не вовремя вызова методаinitElements, PageFactory использует, так называемую, LAZY инициализацию. То есть, поиск элемента будет осуществляться только при обращении к нему в ходе выполнения теста. Если вы никогда не используете элемент в PageObject, то findElement для него никогда не будет вызван. Так объявленный в классе несуществующий на странице элемент вызовет исключение только при попытке его использовать.

PageFactory поддерживает инициализацию элементов по умолчанию. Это позволяет опускать аннотацию @FindBy, и в этом случае имя поля в классе выступает как ID или name элемента на HTML-странице. Сперва драйвер ищет элемент по соответствию id, и если элемент не найден, то затем по имени класса. Инициализация без @FindBy, естественно, не работает для List<WebElement>, поскольку не принято иметь несколько элементов с одинаковым ID или name на странице.

Каждый раз при обращении к элементу драйвер будет снова и снова осуществлять его поиск на странице. Для сложных AJAX-приложений это как раз то, что нужно. Но для ускорения выполнения существует возможность кешировать найденные элементы. Тогда поиск будет осуществляться только один раз при первом обращении к элементу.

|  |
| --- |
| @FindBy(id = "profilename")  @CacheLookup  private WebElement userName; |

Но делайте это только если вы уверены, что:

* элемент всегда будет на странице;
* элемент не будет меняться;
* Вы не покидаете HTML страницу и возвращаетесь обратно ([StaleElementReferenceException](http://internetka.in.ua/selenium-driver-exceptions/)).

Пример класса Page Object:

|  |
| --- |
| import org.junit.Assert;  import org.openqa.selenium.WebDriver;  import org.openqa.selenium.WebElement;  import org.openqa.selenium.support.FindBy;  import org.openqa.selenium.support.PageFactory;    public class RegistrationPage {        private static String URL\_MATCH = "registration";        private WebDriver driver;        /\*\*       \* Логин       \*/      @FindBy(id = "userLogin")      private WebElement login;        /\*\*       \* Пароль       \*/      @FindBy(id = "regPassword")      private WebElement password;        /\*\*       \* Пароль подтверждение       \*/      @FindBy(id = "passwordConfirmation")      private WebElement passwordConfirm;        /\*\*       \* E-mail       \*/      @FindBy(id = "UserEmail")      private WebElement email;        /\*\*       \* Кнопка зарегистрировать       \*/      @FindBy(id = "submitRegistration")      private WebElement bSubmitRegister;        /\*\*       \* Сообщение об ошибке       \*/      @FindBy(id = "error-message")      private WebElement registerError;        public RegistrationPage(WebDriver driver) {          // проверить, что вы находитесь на верной странице          if (!driver.getCurrentUrl().contains(URL\_MATCH)) {                      throw new IllegalStateException(                          "This is not the page you are expected"                          );          }            PageFactory.initElements(driver, this);          this.driver = driver;      }        /\*\*       \* Регистраци пользователя       \* @param user - {@link User}       \*/      private void registerUser(User user) {          System.out.println(driver.getTitle());          login.sendKeys(user.login);          password.sendKeys(user.password);          passwordConfirm.sendKeys(user.passwordConfirmation);          email.sendKeys(user.email);            bSubmitRegister.click();      }        /\*\*       \* Успешная регистрация пользователя       \* @param user - {@link User}       \* @return {@link HomePage}       \*/      public HomePage registerUserSuccess(User user) {          registerUser(user);          return new HomePage(driver);      }        /\*\*       \* Неуспешная регистрация       \* @param user - {@link User}       \* @return {@link RegistrationPage}       \*/      public RegistrationPage registerUserError(User user) {          registerUser(user);          return new RegistrationPage(driver);      }        /\*\*       \* Проверить сообщение об ошибке       \* @param user - {@link User}       \* @return {@link RegistrationPage}       \*/      public RegistrationPage checkErrorMessage(String errorMessage) {          Assert.assertTrue("Error message should be present",                          registerError.isDisplayed());          Assert.assertTrue("Error message should contains " + errorMessage,                          registerError.getText().contains(errorMessage));          return this;      }  } |

Вызов методов страницы регистрации в тесте:

|  |
| --- |
| @Test  public void registerUserTest() {      driver.get("http://HOST\_NAME/registration");      User user = User.createValidUser();      user.email = "not\_valid\_email";        RegistrationPage registrationPage =  new RegistrationPage(driver);      registrationPage          .registerUserError(user)          .checkErrorMessage(errorMessage)          ;        user = User.createValidUser();        registrationPage          .registerUserSuccess(user)          // ...          // вызов методов HomePage          ;  } |

**13. TestNG**

Итак, что же здесь есть? Как и в JUnit 4 тесты описываются с помощью аннотаций, также поддерживаются тесты, написанные на JUnit 3. Есть возможность вместо аннотаций использовать доклет.  
  
Для начала рассмотрим иерархию тестов. Все тесты принадлежат к какой-либо последовательности тестов(сюите), включают в себя некоторое количество классов, каждый из которых может состоять из нескольких тестовых методов. При этом классы и тестовые методы могут принадлежать к определенной группе. Наглядно это выглядит так:

+- suite/

+- test0/

| +- class0/

| | +- method0(integration group)/

| | +- method1(functional group)/

| | +- method2/

| +- class1

| +- method3(optional group)/

+- test1/

+- class3(optional group, integration group)/

+- method4/

У каждого участника этой иерархии могут иметься *before* и *after* конфигураторы. Запускается это все в таком порядке:

+- before suite/

+- before group/

+- before test/

+- before class/

+- before method/

+- test/

+- after method/

...

+- after class/

...

+- after test/

...

+- after group/

...

+- after suite/

Теперь поподробнее о самих тестах. Рассмотрим пример. Утилита для работы с локалями, умеет парсить из строки, а также искать кандидаты(*en\_US -> en\_US, en, root*):

public abstract class LocaleUtils {

*/\*\**

*\* Root locale fix for java 1.5*

*\*/*

public static final Locale ROOT\_LOCALE = new Locale("");

private static final String LOCALE\_SEPARATOR = "\_";

public static Locale parseLocale(final String value) {

if (value != null) {

final StringTokenizer tokens = new StringTokenizer(value, LOCALE\_SEPARATOR);

final String language = tokens.hasMoreTokens() ? tokens.nextToken() : "";

final String country = tokens.hasMoreTokens() ? tokens.nextToken() : "";

String variant = "";

String sep = "";

while (tokens.hasMoreTokens()) {

variant += sep + tokens.nextToken();

sep = LOCALE\_SEPARATOR;

}

return new Locale(language, country, variant);

}

return null;

}

public static List<Locale> getCandidateLocales(final Locale locale) {

final List<Locale> locales = new ArrayList<Locale>();

if (locale != null) {

final String language = locale.getLanguage();

final String country = locale.getCountry();

final String variant = locale.getVariant();

if (variant.length() > 0) {

locales.add(locale);

}

if (country.length() > 0) {

locales.add((locales.size() == 0) ? locale : new Locale(language, country));

}

if (language.length() > 0) {

locales.add((locales.size() == 0) ? locale : new Locale(language));

}

}

locales.add(ROOT\_LOCALE);

return locales;

}

}

Напишем к ней тест в стиле JUnit-a(не стоит рассматривать данный пример как руководство к написанию тестов на TestNG):

public class LocaleUtilsOldStyleTest extends Assert {

private final Map<String, Locale> parseLocaleData = new HashMap<String, Locale>();

@BeforeClass

private void setUp() {

parseLocaleData.put(null, null);

parseLocaleData.put("", LocaleUtils.ROOT\_LOCALE);

parseLocaleData.put("en", Locale.ENGLISH);

parseLocaleData.put("en\_US", Locale.US);

parseLocaleData.put("en\_GB", Locale.UK);

parseLocaleData.put("ru", new Locale("ru"));

parseLocaleData.put("ru\_RU\_xxx", new Locale("ru", "RU", "xxx"));

}

@AfterTest

void tearDown() {

parseLocaleData.clear();

}

@Test

public void testParseLocale() {

for (Map.Entry<String, Locale> entry : parseLocaleData.entrySet()) {

final Locale actual = LocaleUtils.parseLocale(entry.getKey());

final Locale expected = entry.getValue();

assertEquals(actual, expected);

}

}

}

Что здесь есть?

* Как уже было сказано в предыдущей статье я предпочитаю наследовать тест-класс от *Assert*, это можно заменить статическим импортом, либо использованием класса напрямую(*Assert.assertEquals(...)*). В реальной системе удобнее всего наследовать тест от какого-либо базового класса, который в свою очередь наследовать от *Assert*, это дает возможность переопределять либо добавлять необходимые методы. Внимание: в отличие от такого же класса в JUnit здесь во все методы актуальное значение передается первым, ожидаемое вторым(в JUnit наоборот).
* Аннотации *@BeforeSuite*, *@AfterSuite* обозначают методы, которые исполняются единожды до/после исполнения всех тестов. Здесь удобно располагать какие-либо тяжелые настройки общие для всех тестов, например, здесь можно создать пул соединений с базой данных.
* Аннотации *@BeforeTest*, *@AfterTest* обозначают методы, которые исполняются единожды до/после исполнения теста(тот, который включает в себя тестовые классы, не путать с тестовыми методами). Здесь можно хранить настройки какой-либо группы взаимосвязанных сервисов, либо одного сервиса, если он тестируется несколькими тест-классами.
* Аннотации *@BeforeClass*, *@AfterClass* обозначают методы, которые исполняются единожды до/после исполнения всех тестов в классе, идентичны предыдущим, но применимы к тест-классам. Наиболее применим для тестирования какого-то определенного сервиса, который не меняет свое состояние в результате теста.
* Аннотации *@BeforeMethod*, *@AfterMethod* обозначают методы, которые исполняются каждый раз до/после исполнения тестового метода. Здесь удобно хранить настройки для определенного бина или сервиса, если он не меняет свое состояние в результате теста.
* Аннотации *@BeforeGroups*, *@AfterGroups* обозначает методы, которые исполняются до/после первого/последнего теста принадлежащего к заданным группам.
* Аннотация *@Test* обозначает сами тесты. Здесь размещаются проверки. Также применима к классам

У всех этих аннотаций есть следующие параметры:

* enabled — можно временно отключить, установив значение в false
* groups — обозначает, для каких групп будет исполнен
* inheritGroups — если true(а по умолчанию именно так), метод будет наследовать группы от тест-класса
* timeOut — время, после которого метод «свалится» и потянет за собой все зависимые от него тесты
* description — название, используемое в отчете
* dependsOnMethods — методы, от которых зависит, сначала будут выполнены они, а затем данный метод
* dependsOnGroups — группы, от которых зависит
* alwaysRun — если установить в true, будет вызываться всегда независимо от того, к каким группам принадлежит, не применим к *@BeforeGroups*, *@AfterGroups*

Как видно из примера тест практически ничем не отличается от такого же теста на JUnit. Если нет разницы, то зачем использовать TestNG?

Параметризированные тесты

Напишем этот же тест другим способом:

public class LocaleUtilsTest extends Assert {

@DataProvider

public Object[][] parseLocaleData() {

return new Object[][]{

{null, null},

{"", LocaleUtils.ROOT\_LOCALE},

{"en", Locale.ENGLISH},

{"en\_US", Locale.US},

{"en\_GB", Locale.UK},

{"ru", new Locale("ru")},

{"ru\_RU\_some\_variant", new Locale("ru", "RU", "some\_variant")},

};

}

@Test(dataProvider = "parseLocaleData")

public void testParseLocale(String locale, Locale expected) {

final Locale actual = LocaleUtils.parseLocale(locale);

assertEquals(actual, expected);

}

}

Проще? Конечно, данные хранятся отдельно от самого теста. Удобно? Конечно, можно добавлять тесты, добавляя всего лишь строчку в метод parseLocaleData.  
  
Итак, как это работает?

* Объявляем тестовый метод со всеми нужными ему параметрами, например входные и ожидаемые данные. В нашем случае это строка, которую нужно распарсить в локаль и ожидаемая в результате локаль.
* Объявляем дата провайдер, хранилище данных для теста. Обычно это метод, возвращающий *Object[][]* либо *Iterator<Object[]>*, содержащий список параметров для определенного теста, например *{«en\_US», Locale.US}*. Этот метод должен быть зааннотирован с помощью *@DataProvider*, в самом тесте он объявляется с помощью параметра *dataProvider* в аннотации *@Test*. Также можно указать имя(параметр *name*), если не указывать в качестве имени будет использоваться название метода.

Еще один пример, теперь разнесем данные и логику теста в разные классы:

public class LocaleUtilsTestData {

@DataProvider(name = "getCandidateLocalesData")

public static Object[][] getCandidateLocalesData() {

return new Object[][]{

{null, Arrays.asList(LocaleUtils.ROOT\_LOCALE)},

{LocaleUtils.ROOT\_LOCALE, Arrays.asList(LocaleUtils.ROOT\_LOCALE)},

{Locale.ENGLISH, Arrays.asList(Locale.ENGLISH, LocaleUtils.ROOT\_LOCALE)},

{Locale.US, Arrays.asList(Locale.US, Locale.ENGLISH, LocaleUtils.ROOT\_LOCALE)},

{new Locale("en", "US", "xxx"), Arrays.asList(

new Locale("en", "US", "xxx"), Locale.US, Locale.ENGLISH, LocaleUtils.ROOT\_LOCALE)

},

};

}

}

public class LocaleUtilsTest extends Assert {

*// other tests*

@Test(dataProvider = "getCandidateLocalesData", dataProviderClass = LocaleUtilsTestData.class)

public void testGetCandidateLocales(Locale locale, List<Locale> expected) {

final List<Locale> actual = LocaleUtils.getCandidateLocales(locale);

assertEquals(actual, expected);

}

}

В этом случае задаются параметры *dataProviderClass* и *dataProvider*. Метод, возвращающий тестовые данные должен быть *static*.  
  
Кроме описанного выше есть еще один способ параметризировать тесты. Нужный метод аннотируется с помощью *@Parameters*, где указываются имена всех необходимых параметров. Некоторые из параметров можно зааннотировать с помощью *@Optional* с указанием значения по умолчанию(если не указать, то будут использоваться значения по умолчанию для примитивов, либо null для всех остальных типов). Значения параметров хранятся в конфигурации TestNG(которая будет рассмотрена позже). Пример:

public class ParameterizedTest extends Assert {

private DataSource dataSource;

@Parameters({"driver", "url", "username", "password"})

@BeforeClass

public void setUpDataSource(String driver, String url, @Optional("sa") String username, @Optional String password) {

*// create datasource*

dataSource = ...

}

@Test

public void testOptionalData() throws SQLException {

dataSource.getConnection();

*// do some staff*

}

}

В данном случае метод *setUpDataSource* будет принимать в качестве параметров настройки соединения с БД, причем параметры *username* и *password* опциональны, с заданными значениями по умолчанию. Очень удобно использовать с данными, общими для всех тестов(ну или почти всех), например, как в примере настройки соединения с БД.  
  
Ну и в завершение следует сказать пару слов о фабриках, которые позволяют создавать тесты динамически. Также, как и сами тесты, могут быть параметризированы с помощью *@DataProvider* либо *@Parameters*:

public class FactoryTest {

@DataProvider

public Object[][] tablesData() {

return new Object[][] {

{"FIRST\_TABLE"},

{"SECOND\_TABLE"},

{"THIRD\_TABLE"},

};

}

@Factory(dataProvider = "tablesData")

public Object[] createTest(String table) {

return new Object[] { new GenericTableTest(table) };

}

}

public class GenericTableTest extends Assert {

private final String table;

public GenericTableTest(final String table) {

this.table = table;

}

@Test

public void testTable() {

System.out.println(table);

*// do some testing staff here*

}

}

Вариант с *@Parameters*:

public class FactoryTest {

@Parameters("table")

@Factory

public Object[] createParameterizedTest(@Optional("SOME\_TABLE") String table) {

return new Object[] { new GenericTableTest(table) };

}

}

Многопоточность

Нужно проверить, как поведет себя приложение во многопоточном окружении? Можно сделать так, чтобы тесты исполнялись одновременно из нескольких потоков:

public class ConcurrencyTest extends Assert {

private Map<String, String> data;

@BeforeClass

void setUp() throws Exception {

data = new HashMap<String, String>();

}

@AfterClass

void tearDown() throws Exception {

data = null;

}

@Test(threadPoolSize = 30, invocationCount = 100, invocationTimeOut = 10000)

public void testMapOperations() throws Exception {

data.put("1", "111");

data.put("2", "111");

data.put("3", "111");

data.put("4", "111");

data.put("5", "111");

data.put("6", "111");

data.put("7", "111");

for (Map.Entry<String, String> entry : data.entrySet()) {

System.out.println(entry);

}

data.clear();

}

@Test(singleThreaded = true, invocationCount = 100, invocationTimeOut = 10000)

public void testMapOperationsSafe() throws Exception {

data.put("1", "111");

data.put("2", "111");

data.put("3", "111");

data.put("4", "111");

data.put("5", "111");

data.put("6", "111");

data.put("7", "111");

for (Map.Entry<String, String> entry : data.entrySet()) {

System.out.println(entry);

}

data.clear();

}

}

* threadPoolSize определяет максимальное количество потоков используемое для тестов.
* singleThreaded если установлен в true все тесты будут запущены в одном потоке.
* invocationCount определяет количество запусков теста.
* invocationTimeOut определяет общее время всех запусков теста, после которого тест считается провалившемся.

Первый тест будет время от времени проваливаться с *ConcurrentModificationException*, так как будет запускаться из разных потоков, второй — нет, так как все тесты будут запущены последовательно из одного потока.  
  
Еще можно установить параметр *parallel* у дата провайдера в true, тогда тесты для каждого набора данных будут запущены паралельно, в отдельном потоке:

public class ConcurrencyTest extends Assert {

*// some staff here*

@DataProvider(parallel = true)

public Object[][] concurrencyData() {

return new Object[][] {

{"1", "2"},

{"3", "4"},

{"5", "6"},

{"7", "8"},

{"9", "10"},

{"11", "12"},

{"13", "14"},

{"15", "16"},

{"17", "18"},

{"19", "20"},

};

}

@Test(dataProvider = "concurrencyData")

public void testParallelData(String first, String second) {

final Thread thread = Thread.currentThread();

System.out.printf("#%d %s: %s : %s", thread.getId(), thread.getName(), first, second);

System.out.println();

}

}

Данный тест будет выводить нечто вроде:

#16 pool-1-thread-3: 5 : 6

#19 pool-1-thread-6: 11 : 12

#14 pool-1-thread-1: 1 : 2

#22 pool-1-thread-9: 17 : 18

#20 pool-1-thread-7: 13 : 14

#18 pool-1-thread-5: 9 : 10

#15 pool-1-thread-2: 3 : 4

#17 pool-1-thread-4: 7 : 8

#21 pool-1-thread-8: 15 : 16

#23 pool-1-thread-10: 19 : 20

Без этого параметра будет что-то вроде:

#1 main: 1 : 2

#1 main: 3 : 4

#1 main: 5 : 6

#1 main: 7 : 8

#1 main: 9 : 10

#1 main: 11 : 12

#1 main: 13 : 14

#1 main: 15 : 16

#1 main: 17 : 18

#1 main: 19 : 20

Дополнительные возможности

Кроме всего описанного есть и другие возможности, например для проверки выброса исключений(очень удобно использовать для тестов на неправильных данных):

public class ExceptionTest {

@DataProvider

public Object[][] wrongData() {

return new Object[][] {

{"Hello, World!!!"},

{"0x245"},

{"1798237199878129387197238"},

};

}

@Test(dataProvider = "wrongData", expectedExceptions = NumberFormatException.class,

expectedExceptionsMessageRegExp = "^For input string: \"(.\*)\"$")

public void testParse(String data) {

Integer.parseInt(data);

}

}

* expectedExceptions задает варианты ожидаемых исключений, если они не выбрасываются, тест считается провалившемся.
* expectedExceptionsMessageRegExp то же что и предыдущий параметр, но задает regexp для сообщения об ошибке.

Еще один пример:

public class PrioritiesTest extends Assert {

private boolean firstTestExecuted;

@BeforeClass

public void setUp() throws Exception {

firstTestExecuted = false;

}

@Test(priority = 0)

public void first() {

assertFalse(firstTestExecuted);

firstTestExecuted = true;

}

@Test(priority = 1)

public void second() {

assertTrue(firstTestExecuted);

}

}

* priority определяет приоритет теста внутри класса, чем меньше, тем раньше будет выполнен.

Данный пример пройдет успешно, так как сначала выполнится метод *first*, затем *second*. Если поменять приоритет у *first* на 2, тест провалится.  
  
Похожее поведение будет наблюдаться также если указать зависимости у теста, например, добавим в наш тест:

public class PrioritiesTest extends Assert {

*// some staff*

@Test(dependsOnMethods = {"first"})

public void third() {

assertTrue(firstTestExecuted);

}

*// some staff*

}

Обычно это удобно, когда один тест зависит от другого, например, *Утилита1* использует *Утилиту0*, если *Утилита0* работает неправильно, то нет смысла тестировать *Утилиту1*. С другой стороны зависимости также удобно использовать в *@Before*, *@After* методах, особенно для связи базового теста с наследующимся, причем иногда бывает необходимо сделать так, чтобы даже если метод *A*свалился, а метод *B* зависит от него, метод *B* все равно вызывался. В этом случае устанавливаем параметр *alwaysRun* в true.

Внедрение зависимостей

Хочу порадовать любителей фреймворка от «корпорации добра» [Guice](http://code.google.com/p/google-guice/). В TestNG есть встроенная поддержка последнего. Выглядит это так:

public class GuiceModule extends AbstractModule {

@Override

protected void configure() {

bind(String.class).annotatedWith(Names.named("guice-string-0")).toInstance("Hello, ");

}

@Named("guice-string-1")

@Inject

@Singleton

@Provides

public String provideGuiceString() {

return "World!!!";

}

}

@Guice(modules = {GuiceModule.class})

public class GuiceTest extends Assert {

@Inject

@Named("guice-string-0")

private String word0;

@Inject

@Named("guice-string-1")

private String word1;

@Test

public void testService() {

final String actual = word0 + word1;

assertEquals(actual, "Hello, World!!!");

}

}

Все, что надо — зааннотировать нужный класс с помощью *@Guice* и указать в параметре modules все необходимые guice-модули. Далее в тест классе можно уже использовать внедрение зависимостей, используя *@Inject*.  
  
Добавлю еще, что любителям других подобных фреймворков не стоит расстраиваться, так как у них обычно есть своя поддержка TestNG, например, у Spring-а.

Расширение функционала

Расширение функционала может быть реализовано с помощью механизма слушателей. Поддерживаются следующие типы слушателей:

* [IAnnotationTransformer, IAnnotationTransformer2](http://testng.org/doc/documentation-main.html#annotationtransformers) — позволяют переопределять настройки теста, например, количество потоков для запуска теста, таймаут, ожидаемое исключение:
* public class ExpectTransformer implements IAnnotationTransformer {
* public void transform(ITestAnnotation annotation, Class testClass, Constructor testConstructor, Method testMethod) {
* if (testMethod.getName().startsWith("expect")) {
* annotation.setExpectedExceptions(new Class[] {Exception.class});
* }
* }
* }

Данный пример будет ожидать выброс исключения от тест-методов, начинающихся с expect.

* [IHookable](http://testng.org/doc/documentation-main.html#ihookable) — позволяет переопределить тест-метод или по возможности пропустить, в туториале к TestNG приводится пример с JAAS.
* IInvokedMethodListener, IInvokedMethodListener2 — похож на предыдущий слушатель, но исполняет код до и после исполнения тест-метода
* [IMethodInterceptor](http://testng.org/doc/documentation-main.html#methodinterceptors) — позволяет изменять порядок запуска тестов(применим только к тестам, которые независимы от других тестов). В туториале есть хороший пример
* [IReporter](http://testng.org/doc/documentation-main.html#logging-reporters) — позволяет расширить функционал, выполняемый после выполнения всех тестов, обычно этот функционал связан с генерацией отчетов об ошибках и т.д. Таким образом можно реализовать свой механизм отчетов
* ITestListener — слушатель, который может обрабатывать большинство событий от тест-метода, например, start, finish, success, failure
* ISuiteListener — похож на предыдущий, но для сюит, получает только события start и finish

Об одном из примеров интересного использования механизма слушателей можно почитать [здесь](http://seleniumexamples.com/blog/guide/using-soft-assertions-in-testng/).

Конфигурация

Теперь перейдем к конфигурации тестов. Простейший способ запустить тесты выглядит примерно так:

final TestNG testNG = new TestNG(true);

testNG.setTestClasses(new Class[] {SuperMegaTest.class});

testNG.setExcludedGroups("optional");

testNG.run();

Но чаще всего для запуска тестов используется XML либо YAML конфигурация. XML конфигурация выглядит примерно так:

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Test suite" parallel="classes" thread-count="10">

<test name="Default tests" verbose="1" annotations="JDK" thread-count="10" parallel="classes">

<parameter name="driver" value="com.mysql.jdbc.Driver"/>

<parameter name="url" value="jdbc:mysql://localhost:3306/db"/>

<groups>

<run>

<exclude name="integration"/>

</run>

</groups>

<packages>

<package name="com.example.\*"/>

</packages>

</test>

<test name="Integration tests" annotations="JDK">

<groups>

<run>

<include name="integration"/>

</run>

</groups>

<packages>

<package name="com.example.\*"/>

</packages>

</test>

</suite>

Аналогичная YAML конфигурация:

name: YAML Test suite

parallel: classes

threadCount: 10

tests:

- name: Default tests

parameters:

driver: com.mysql.jdbc.Driver

url: jdbc:mysql:*//localhost:3306/db*

excludedGroups: [ integration ]

packages:

- com.example.\*

- name: Integration tests

parameters:

driver: com.mysql.jdbc.Driver

url: jdbc:mysql:*//localhost:3306/db*

includedGroups: [ integration ]

packages:

- com.example.\*

Тогда для запуска тестов нужно будет сделать следующее:

final TestNG testNG = new TestNG(true);

*//final Parser parser = new Parser("testing/testing-testng/src/test/resources/testng.xml");*

final Parser parser = new Parser("testing/testing-testng/src/test/resources/testng.yaml");

final List<XmlSuite> suites = parser.parseToList();

testNG.setXmlSuites(suites);

testNG.run();

Хотя, наверное, программный запуск тестов это излишне, т.к. для этого можно использовать возможности IDE.  
  
Вернемся к самой конфигурации. На самом верхнем уровне настраивается последовательность тестов(сюита). Может принимать следующие параметры:

* name — название используемое в отчете
* thread-count — количество потоков используемое для запуска тестов
* data-provider-thread-count — количество потоков используемое для передачи данных из дата-провайдеров в сами тесты для параллельных дата провайдеров(*@DataProvider(parallel = true)*)
* parallel — может принимать следующие значения:
* methods — тестовые методы будут запущены в разных потоках, нужно быть осторожным если есть зависимости между методами
* classes — все методы одного класса в одном потоке, но разные классы в разных потоках
* tests — все методы одного теста в одном потоке, разные тесты в разных потоках
* time-out — время, после которого тест будет считаться провалившимся, то же что и в аннотации, но распостраняется на все тестовые методы
* junit — JUnit 3 тесты
* annotations — если javadoc, то будет использован доклет для конфигурации

Также могут быть настроены:

* parameter — параметры, те, что используются в *@Parameters*
* packages — пакеты, где искать тест-классы
* listeners — слушатели, с их помощью можно расширить функционал TestNG, о них уже сказал пару слов
* method-selectors — селекторы для тестов, должны реализовывать интерфейс *IMethodSelector*
* suite-files — можно включать другие файлы конфигурации

Сюиты в свою очередь могут включать в себя тесты с практически такими же настройками, что и для сюит (аттрибуты *name, thread-count, parallel, time-out, junit, annotations*, тэги *parameter, packages, method-selectors*). Также у тестов имеются и своеобразные настройки, например, запускаемые группы:

<test name="Default tests" verbose="1" annotations="JDK" thread-count="10" parallel="classes">

*<!-- some staff here -->*

<groups>

<run>

<exclude name="integration"/>

</run>

</groups>

*<!-- some staff here -->*

</test>

В данном примере тест будет включать в себя только тесты не относящиеся к группе integration.  
  
Еще тесты могут включать в себя тест-классы, которые в свою очередь могут включать/исключать в себя тест-методы.

<test name="Integration tests">

<groups>

<run>

<include name="integration"/>

</run>

</groups>

<classes>

<class name="com.example.PrioritiesTest">

<methods>

<exclude name="third"/>

</methods>

</class>

</classes>

</test>

Вывод

Это не все, что можно сказать по этому замечательному фреймворку, все охватить в одной статье очень трудно. Но я постарался раскрыть основные моменты его использования. Тут, конечно, можно было рассказать немного больше про механизм слушателей, про интеграции с другими фреймворками, про конфигурацию тестов, но тогда статья превратилась бы в книгу, да и не знаю я всего. Но, надеюсь, эта статья была кому-то полезна и то, что было сказано, сподвигнет некоторых читателей на использование TestNG, или хотя бы на DDT технику написания тестов.

**14. JUnit**

Для создания теста нужно унаследовать тест-класс от *TestCase*, переопределить методы *setUp* и *tearDown* если надо, ну и самое главное — создать тестовые методы(должны начинаться с *test*). При запуске теста сначала создается экземляр тест-класса(для каждого теста в классе отдельный экземпляр класса), затем выполняется метод *setUp*, запускается сам тест, ну и в завершение выполняется метод *tearDown*. Если какой-либо из методов выбрасывает исключение, тест считается провалившимся.  
  
Примечание: тестовые методы должны быть *public void*, могут быть *static*.  
  
Сами тесты состоят из выполнения некоторого кода и проверок. Проверки чаще всего выполняются с помощью класса *Assert* хотя иногда используют ключевое слово *assert*.  
  
Рассмотрим пример. Есть утилита для работы со строками, есть методы для проверки пустой строки и представления последовательности байт в виде 16-ричной строки:

public abstract class StringUtils {

private static final int HI\_BYTE\_MASK = 0xf0;

private static final int LOW\_BYTE\_MASK = 0x0f;

private static final char[] HEX\_SYMBOLS = {

'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f',

};

public static boolean isEmpty(final CharSequence sequence) {

return sequence == null || sequence.length() <= 0;

}

public static String toHexString(final byte[] data) {

final StringBuffer builder = new StringBuffer(2 \* data.length);

for (byte item : data) {

builder.append(HEX\_SYMBOLS[(HI\_BYTE\_MASK & item) >>> 4]);

builder.append(HEX\_SYMBOLS[(LOW\_BYTE\_MASK & item)]);

}

return builder.toString();

}

}

Напишем для нее тесты, используя JUnit 3. Удобнее всего, на мой взгляд, писать тесты, рассматривая нейкий класс как черный ящик, писать отдельный тест на каждый значимый метод в этом классе, для каждого набора входных параметров какой-то ожидаемый результат. Например, тест для *isEmpty*метода:

public void testIsEmpty() {

boolean actual = StringUtils.isEmpty(null);

assertTrue(actual);

actual = StringUtils.isEmpty("");

assertTrue(actual);

actual = StringUtils.isEmpty(" ");

assertFalse(actual);

actual = StringUtils.isEmpty("some string");

assertFalse(actual);

}

Можно разделить данные и логику теста, перенеся создание данных в метод *setUp*:

public class StringUtilsJUnit3Test extends TestCase {

private final Map toHexStringData = new HashMap();

protected void setUp() throws Exception {

toHexStringData.put("", new byte[0]);

toHexStringData.put("01020d112d7f", new byte[] { 1, 2, 13, 17, 45, 127 });

toHexStringData.put("00fff21180", new byte[] { 0, -1, -14, 17, -128 });

*//...*

}

protected void tearDown() throws Exception {

toHexStringData.clear();

}

public void testToHexString() {

for (Iterator iterator = toHexStringData.keySet().iterator(); iterator.hasNext();) {

final String expected = (String) iterator.next();

final byte[] testData = (byte[]) toHexStringData.get(expected);

final String actual = StringUtils.toHexString(testData);

assertEquals(expected, actual);

}

}

*//...*

}

## Дополнительные возможности

Кроме того, что было описано, есть еще несколько дополнительных возможностей. Например, можно группировать тесты. Для этого нужно использовать класс *TestSuite*:

public class StringUtilsJUnit3TestSuite extends TestSuite {

public StringUtilsJUnit3TestSuite() {

addTestSuite(StringUtilsJUnit3Test.class);

addTestSuite(OtherTest1.class);

addTestSuite(OtherTest2.class);

}

}

Можно запустить один и тот же тест несколько раз. Для этого используем *RepeatedTest*:

public class StringUtilsJUnit3RepeatedTest extends RepeatedTest {

public StringUtilsJUnit3RepeatedTest() {

super(new StringUtilsJUnit3Test(), 100);

}

}

Наследуя тест-класс от *ExceptionTestCase*, можно проверить что-либо на выброс исключения:

public class StringUtilsJUnit3ExceptionTest extends ExceptionTestCase {

public StringUtilsJUnit3ExceptionTest(final String name) {

super(name, NullPointerException.class);

}

public void testToHexString() {

StringUtils.toHexString(null);

}

}

Как видно из примеров все довольно просто, ничего лишнего, минимум нужный для тестирования(хотя недостает и некоторых нужных вещей).

# JUnit 4

Здесь была добавлена поддержка новых возможностей из Java 5, тесты теперь могут быть объявлены с помощью аннотаций. При этом существует обратная совместимость с предыдущей версией фреймворка, практически все рассмотренные выше примеры будут работать и здесь(за исключением *RepeatedTest*, его нет в новой версии).  
  
Итак, что же поменялось?

## Основные аннотации

Рассмотрим тот же пример, но уже используя новые возможности:

public class StringUtilsJUnit4Test extends Assert {

private final Map<String, byte[]> toHexStringData = new HashMap<String, byte[]>();

@Before

public static void setUpToHexStringData() {

toHexStringData.put("", new byte[0]);

toHexStringData.put("01020d112d7f", new byte[] { 1, 2, 13, 17, 45, 127 });

toHexStringData.put("00fff21180", new byte[] { 0, -1, -14, 17, -128 });

*//...*

}

@After

public static void tearDownToHexStringData() {

toHexStringData.clear();

}

@Test

public void testToHexString() {

for (Map.Entry<String, byte[]> entry : toHexStringData.entrySet()) {

final byte[] testData = entry.getValue();

final String expected = entry.getKey();

final String actual = StringUtils.toHexString(testData);

assertEquals(expected, actual);

}

}

}

Что мы здесь видим?

* Для упрощения работы я предпочитаю наследоваться от класса *Assert*, хотя это необязательно.
* Аннотация [*Before*](https://habrahabr.ru/users/before/) обозначает методы, которые будут вызваны до исполнения теста, методы должны быть *public void*. Здесь обычно размещаются предустановки для теста, в нашем случае это генерация тестовых данных (метод *setUpToHexStringData*).
* Аннотация *@BeforeClass* обозначает методы, которые будут вызваны до создания экземпляра тест-класса, методы должны быть *public static void*. Имеет смысл размещать предустановки для теста в случае, когда класс содержит несколько тестов использующих различные предустановки, либо когда несколько тестов используют одни и те же данные, чтобы не тратить время на их создание для каждого теста.
* Аннотация [*After*](https://habrahabr.ru/users/after/) обозначает методы, которые будут вызваны после выполнения теста, методы должны быть *public void*. Здесь размещаются операции освобождения ресурсов после теста, в нашем случае — очистка тестовых данных (метод *tearDownToHexStringData*).
* Аннотация *@AfterClass* связана по смыслу с *@BeforeClass*, но выполняет методы после теста, как и в случае с *@BeforeClass*, методы должны быть *public static void*.
* Аннотация [*Test*](https://habrahabr.ru/users/test/) обозначает тестовые методы. Как и ранее, эти методы должны быть *public void*. Здесь размещаются сами проверки. Кроме того, у данной аннотации есть два параметра, *expected*— задает ожидаемое исключение и *timeout* — задает время, по истечению которого тест считается провалившимся.

@Test(expected = NullPointerException.class)

public void testToHexStringWrong() {

StringUtils.toHexString(null);

}

@Test(timeout = 1000)

public void infinity() {

while (true);

}

Если какой-либо тест по какой-либо серьезной причине нужно отключить(например, этот тест постоянно валится, но его исправление отложено до светлого будущего) его можно зааннотировать *@Ignore*. Также, если поместить эту аннотацию на класс, то все тесты в этом классе будут отключены.

@Ignore

@Test(timeout = 1000)

public void infinity() {

while (true);

}

## Правила

Кроме всего вышеперечисленного есть довольно интересная вещь — правила. Правила это некое подобие утилит для тестов, которые добавляют функционал до и после выполнения теста.  
  
Например, есть встроенные правила для задания таймаута для теста(*Timeout*), для задания ожидаемых исключений(*ExpectedException*), для работы с временными файлами(*TemporaryFolder*) и д.р. Для объявления правила необходимо создать *public* не *static* поле типа производного от *MethodRule* и зааннотировать его с помощью [*Rule*](https://habrahabr.ru/users/rule/).

public class OtherJUnit4Test {

@Rule

public final TemporaryFolder folder = new TemporaryFolder();

@Rule

public final Timeout timeout = new Timeout(1000);

@Rule

public final ExpectedException thrown = ExpectedException.none();

@Ignore

@Test

public void anotherInfinity() {

while (true);

}

@Test

public void testFileWriting() throws IOException {

final File log = folder.newFile("debug.log");

final FileWriter logWriter = new FileWriter(log);

logWriter.append("Hello, ");

logWriter.append("World!!!");

logWriter.flush();

logWriter.close();

}

@Test

public void testExpectedException() throws IOException {

thrown.expect(NullPointerException.class);

StringUtils.toHexString(null);

}

}

Также в сети можно найти и другие варианты использования. Например, [здесь](http://blog.mycila.com/2009/11/writing-your-own-junit-extensions-using.html) рассмотрена возможность параллельного запуска теста.

## Запускалки

Но и на этом возможности фреймворка не заканчиваются. То, как запускается тест, тоже может быть сконфигурировано с помощью *@RunWith*. При этом класс, указанный в аннотации должен наследоваться от *Runner*. Рассмотрим запускалки, идущие в комплекте с самим фреймворком.  
  
*JUnit4* — запускалка по умолчанию, как понятно из названия, предназначена для запуска JUnit 4 тестов.  
  
*JUnit38ClassRunner* предназначен для запуска тестов, написанных с использованием JUnit 3.  
  
*SuiteMethod* либо *AllTests* тоже предназначены для запуска JUnit 3 тестов. В отличие от предыдущей запускалки, в эту передается класс со статическим методом suite возвращающим тест(последовательность всех тестов).  
  
*Suite* — эквивалент предыдущего, только для JUnit 4 тестов. Для настройки запускаемых тестов используется аннотация *@SuiteClasses*.

@Suite.SuiteClasses( { OtherJUnit4Test.class, StringUtilsJUnit4Test.class })

@RunWith(Suite.class)

public class JUnit4TestSuite {

}

*Enclosed* — то же, что и предыдущий вариант, но вместо настройки с помощью аннотации используются все внутренние классы.  
  
*Categories* — попытка организовать тесты в категории(группы). Для этого тестам задается категория с помощью *@Category*, затем настраиваются запускаемые категории тестов в сюите. Это может выглядеть так:

public class StringUtilsJUnit4CategoriesTest extends Assert {

*//...*

@Category(Unit.class)

@Test

public void testIsEmpty() {

*//...*

}

*//...*

}

@RunWith(Categories.class)

@Categories.IncludeCategory(Unit.class)

@Suite.SuiteClasses( { OtherJUnit4Test.class, StringUtilsJUnit4CategoriesTest.class })

public class JUnit4TestSuite {

}

*Parameterized* — довольно интересная запускалка, позволяет писать параметризированные тесты. Для этого в тест-классе объявляется статический метод возвращающий список данных, которые затем будут использованы в качестве аргументов конструктора класса.

@RunWith(Parameterized.class)

public class StringUtilsJUnit4ParameterizedTest extends Assert {

private final CharSequence testData;

private final boolean expected;

public StringUtilsJUnit4ParameterizedTest(final CharSequence testData, final boolean expected) {

this.testData = testData;

this.expected = expected;

}

@Test

public void testIsEmpty() {

final boolean actual = StringUtils.isEmpty(testData);

assertEquals(expected, actual);

}

@Parameterized.Parameters

public static List<Object[]> isEmptyData() {

return Arrays.asList(new Object[][] {

{ null, true },

{ "", true },

{ " ", false },

{ "some string", false },

});

}

}

*Theories* — чем-то схожа с предыдущей, но параметризирует тестовый метод, а не конструктор. Данные помечаются с помощью *@DataPoints* и *@DataPoint*, тестовый метод — с помощью [*Theory*](https://habrahabr.ru/users/theory/). Тест использующий этот функционал будет выглядеть примерно так:

@RunWith(Theories.class)

public class StringUtilsJUnit4TheoryTest extends Assert {

@DataPoints

public static Object[][] isEmptyData = new Object[][] {

{ "", true },

{ " ", false },

{ "some string", false },

};

@DataPoint

public static Object[] nullData = new Object[] { null, true };

@Theory

public void testEmpty(final Object... testData) {

final boolean actual = StringUtils.isEmpty((CharSequence) testData[0]);

assertEquals(testData[1], actual);

}

}

Как и в случае с правилами, в сети можно найти и другие варианты использования. Например, [здесь](http://blog.mycila.com/2009/11/writing-your-own-junit-extensions-using.html)рассмотрена та же возможность паралельного запуска теста, но с использованием запускалок.

# Вывод

Это, конечно же, не все, что можно было сказать по JUnit-у, но я старался вкратце и по делу. Как видно, фреймворк достаточно прост в использовании, дополнительных возможностей немного, но есть возможность расширения с помощью правил и запускалок.

**15. Selenium Grid**

## Введение

Кластер Grid обеспечивает:

* Масштабируемость, за счет распределения тестов по нескольким машинам (за счет параллельного выполнения).
* Управление различными средами из одного места, упрощая проведение тестов на многочисленных комбинациях ОС и браузеров.
* Сокращение времени, необходимого для обслуживания кластера, благодаря возможности создания пользовательских хуков для использования виртуальной инфраструктуры узла кластера.

## Краткое руководство

*На этом примере мы покажем, как запустить хаб Selenium 2, зарегистрировать узел WebDriver или узел старой версии Selenium 1 RC. Мы также покажем, как вызывать кластер из Java. В приведенном примере и хаб, и все узлы запускаются на одной машине, но вы, конечно же, можете скопировать selenium-server-standalone на множество машин. Примечание: Пакет selenium-server-standalone включает в себя хаб, Selnium WebDriver и старый Selenium RC, необходимые для запуска кластера. Ant больше не требуется. Вы можете загрузить selenium-server-standalone-\*.jar с*[*http://code.google.com/p/selenium/downloads/list*](http://code.google.com/p/selenium/downloads/list)*. В этом руководстве предполагается, что Java уже установлена.*

### Шаг 1: Запуск хаба

Хаб - это центральный узел, который получает все запросы тестов и распределяет их по надлежащим узлам.

Откройте командную строку и перейдите в ту директорию, куда вы скопировали файл selenium-server-standalone . Введите следующую команду:

Хаб автоматически запустится по умолчанию на 4444 порте. Для изменения порта по умолчанию можно добавить к команде необязательный параметр -port. Вы можете наблюдать состояние хаба, открыв браузер и перейдя по адресу <http://localhost:4444/grid/console>

### Шаг 2: Запуск узлов

Независимо от того, хотите ли вы запустить кластер с функционалом нового WebDriver, или функционалом Selenium 1 RC, или обоих одновременно, вы можете использовать один и тот же selenium-server-standalone jar-файл для запуска узлов:

Примечание: в случае, когда присутствует параметр -role не в значении “hub” и порт явно не указан, по умолчанию будет использован 5555.

Для обратной совместимости роли “wd” и “rc” все еще являются допустимыми именами ролей узлов. Но эти роли ограничивают виды удаленных соединений до соответствующего API, в то время как “node” позволяет устанавливать соединения как RC, так и WebDriver.

### Шаг 3: Использование кластера для запуска тестов

(Как пример используется Java) Теперь, когда запущен кластер, мы должны получить к нему доступ из наших тестов. Для узлов Selenium RC можно продолжать использовать объект DefaultSelenium и передавать в него настройки хаба:

Selenium selenium = new DefaultSelenium("localhost", 4444, "\*firefox", "http://www.google.com");

Для узлов WebDriver нужно использовать объекты RemoteWebDriver и [``](https://selenium2.ru/docs/selenium-grid.html#id6)DesiredCapabilities``для определения желаемого браузера, версии и платформы. Укажите характеристики целевого браузера для проведения тестов:

DesiredCapabilities capability = DesiredCapabilities.firefox();

Передайте их в объект RemoteWebDriver:

WebDriver driver = new RemoteWebDriver(new URL("http://localhost:4444/wd/hub"), capability);

И хаб назначит тест на соответствующий узел.

Узел будет выбран, если все запрошенные характеристики будут совпадать. Чтобы запросить конкретные характеристики у кластера, укажите их перед передачей в объекте WebDriver:

capability.setBrowserName();

capability.setPlatform();

capability.setVersion()

capability.setCapability(,);

Пример: узел, зарегистрированный с настройками

соответствует характеристикам:

capability.setBrowserName("firefox" );

capability.setPlatform("LINUX");

capability.setVersion("3.6");

или также соотвествует характеристикам:

capability.setBrowserName("firefox" );

capability.setVersion("3.6");

Не указанные характеристики будут проигнорированы. Если указать характеристики, не существующие в вашем кластере (например, ваш тест указывает Firefox версии 4.0, но нигде не запущен экземпляр Firefox 4.0), то соответствие не будет установлено и тест не запустится.

## Настройка узлов

По умолчанию запускается 11 браузеров: 5 Firefox, 5 Chrome, 1 Internet Explorer. Максимальное количество параллельных тестов по умолчанию равно 5. Для изменения этих и прочих настроек браузеров можно передавать параметры в каждом ключе -browser(каждый ключ представляет узел с заданными вами параметрами). Если используется ключ -browser, браузеры по умолчанию игнорируются и будут использоваться только те браузеры, которые заданы этим ключом.

Такие параметры приведут к запуску 5 узлов с Firefox 3.6 на Linux-машине.

Если на вашей удаленной машине есть несколько версий Firefox, которые вы бы хотели использовать, вы можете указать расположение исполняемых файлов соответствующих версий на одной и той же машине:

Совет: если в значении какого-либо параметра присутствуют пробелы, необходимо заключить этот параметр в двойные кавычки:

## Необязательные параметры

* -port 4444 (4444 - значение по умолчанию)
* -nodeTimeout 30 (30 - значение по умолчанию) Время в секундах перед тем, как хаб автоматически освободит узел, не получающий никаких запросов более определенного количества секунд. По истечении этого времени узел будет освобожден для следующего в очереди теста. Это позволяет обходить клиентские сбои без ручного вмешательства. Чтобы отключить эту возможность, укажите -nodeTimeout 0. Тогда хаб не будет освобождать узел.

Примечание: Это НЕ таймаут WebDriver для команд типа “wait for WebElement”.

* -maxSession 5 (5 - значение по умолчанию) Максимальное количество браузеров, которые могут быть запущены параллельно на одном узле. Этот параметр отличается от параметра maxInstance, задающего макс. количество экземпляров одного поддерживаемого браузера (например maxSession=1 на узле, поддерживающим Firefox 3.6, Firefox 4.0 и Internet Explorer 8, обеспечит, что никогда не будет запущено больше одного браузера. С maxSession=2 может быть одновременно запущено 2 Firefox теста, или 1 Internet Explorer и 1 Firefox тест).
* -browser < params > Если параметр -browser не задан, узел запустится с пятью экземплярами Firefox, пятью Chrome и одним Internet Explorer (предполагается, что это Windows-машина). Для задания различных типов браузеров можно несколько раз указать этот параметр в одной строке.

Параметры, поддерживаемые -browser: .. code-block:: bash browserName={android, chrome, firefox, htmlunit, internet explorer, iphone, opera}

*version={версия браузера} firefox\_binary={путь к исполняемому файлу} maxInstances={максимальное количество браузеров указанного типа} platform={WINDOWS, LINUX, MAC}*

* -registerCycle насколько часто в миллисекундах узел будет пытаться перерегистрировать себя. Позволяет перезапускать хаб без необходимости перезапуска узлов.

## Сообщения для диагностики хаба

При обнаружении аномальных моделей использования, хаб может выдать следующее сообщение:

*Client requested session XYZ that was terminated due to REASON| (Клиент запросил сессию XYZ, которая была завершена по причине REASON)*

| **REASON** | **Причинакак исправить** |
| --- | --- |
| TIMEOUT | Эта сессия истекла, поскольку клиент не обращался к ней в течение указанного времени. Если клиент был каким-либо образом приостановлен, это может случиться при его пробуждении. |
| ORPHAN | Ожидающий в очереди клиент отказался, как только ему была выделена новая сессия. |
| CLIENT\_STOPPED\_SESSION | Сессия была остановлена клиентом с использованием обычного запроса для остановки/выхода. Почему вы снова используете ее? |
| CLIENT\_GONE | Клиентский процесс (ваш код) либо умер, либо по иной причине не отвечает на запросы. Может быть связано с проблемами сети. |
| FORWARDING\_TO\_NODE\_FAILED | Хаб не смог связаться с узлом. Может быть вызвано нехваткой памяти, нестабильностью узла или проблемами сети. |
| CREATIONFAILED | Узел не смог запустить браузер. Это обычно случается, когда есть проблемы с средой/конфигурацией самого узла. Попробуйте непосредственно зайти на узел для выяснения проблемы. |
| PROXY\_REREGISTRATION | Сессия была отменена, поскольку узел перерегистрировался в кластере (во время выполнения теста). |

**16. Реляциоцинные бд, основные сущности**

**GitLab**

## GitLab

GitWeb довольно-таки прост. Если вам нужен более современный, полнофункциональный Git-сервер, есть несколько решений с открытым исходным кодом, которые можно использовать. Так как GitLab это один из самых популярных, мы рассмотрим его установку и использование в качестве примера. Это немного сложнее, чем GitWeb, и скорее всего потребует больше обслуживания, но и функциональность гораздо богаче.

### Установка

GitLab — это веб-приложение на основе базы данных, так что его установка немного сложней, чем у некоторых других серверов git. К счастью, этот процесс хорошо задокументирован и поддерживается.

Есть несколько подходов к установке GitLab. Для экономии времени, вы можете загрузить образ виртуальной машины или инсталлятор из <https://bitnami.com/stack/gitlab>, и поправить конфигурацию под своё окружение. Приятная возможность, включенная Bitnami, это экран входа (доступен по нажатию alt-→); он указывает IP-адрес и логин с паролем по умолчанию для установленного GitLab.

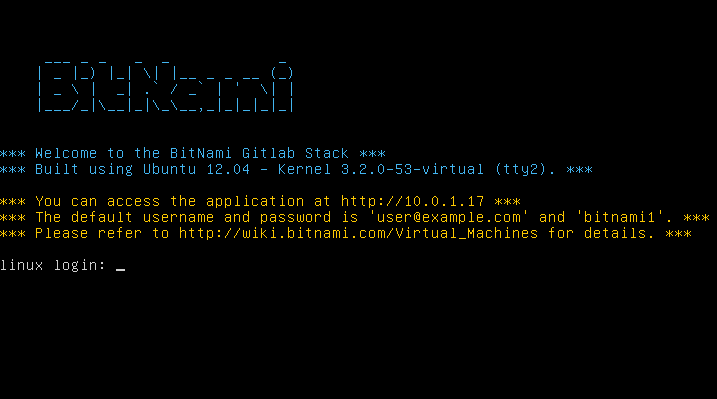


Рисунок 50. Экран входа виртуальной машины Bitnami GitLab.

Для всего остального, следуйте инструкциям GitLab Community Edition, которые можно найти на <https://gitlab.com/gitlab-org/gitlab-ce/tree/master>. Там вы найдёте помощь по установке GitLab посредством рецептов Chef, на виртуальную машину Digital Ocean, или из RPM или DEB пакетов (которые на момент написания всё находились в бета-версии). Есть также “неофициальная” инструкция по запуску GitLab на нестандартных операционных системах и базах данных, полностью ручной сценарий установки и множество других тем.

### Администрирование

Административный интерфейс GitLab доступен через веб. Просто направьте ваш браузер на имя или IP-адрес хоста, где установлен GitLab, и войдите как администратор. Имя пользователя по умолчанию admin@local.host, пароль по умолчанию 5iveL!fe (вас попросят изменить их при входе). Войдя, нажмите иконку “Административная зона” в меню справа и сверху.



Рисунок 51. Пункт “Административная зона” в меню GitLab.

#### Пользователи

Пользователи в GitLab — это учётные записи, соответствующие людям. Пользовательские учётные записи не очень сложны; в основном это набор персональной информации, прикреплённый к имени. У каждого пользователя есть **пространство имён**, логически группирующее проекты данного пользователя. Если у пользователя jane есть проект project, адрес этого проекта будет <http://server/jane/project>.



Рисунок 52. Экран управления пользователями GitLab.

Удаление пользователя может быть выполнено двумя способами. “Блокирование” (“Blocking”) пользователя запрещает ему вход в GitLab, но все данные в его пространстве имен сохраняются, и коммиты, подписанные этим пользователем, будут указывать на его профиль.

“Разрушение” (“Destroying”) пользователя, с другой стороны, полностью удаляет его из базы данных и файловой системы. Все проекты и данные в его пространстве имен удаляются, как и все принадлежащие ему группы. Конечно, этим более постоянным и разрушительным действием пользуются реже.

#### Группы

Группы GitLab — это коллекция проектов с указанием того, как пользователи получают к ним доступ. Каждая группа имеет пространство имён проектов (так же как и пользователи), так что если в группе training есть проект materials, его адрес будет <http://server/training/materials>.



Рисунок 53. Экран управления группами GitLab.

Каждая группа связана с пользователями, каждый из которых имеет уровень доступа к проектам группы и к самой группе. Он разнится от “Гостя” (“Guest”, только проблемы и чат) до “Владельца” (“Owner”, полный контроль над группой, её членами и проектами). Типы разрешений слишком обширны, чтобы перечислять их здесь, но на экране управления GitLab есть полезная ссылка с описанием.

#### Проекты

Проект GitLab примерно соответствует одному git-репозиторию. Каждый проект принадлежит одному пространству имён, групповому или пользовательскому. Если проект принадлежит пользователю, владелец контролирует, кто имеет доступ к проекту; если проект принадлежит группе, действуют групповые уровни доступа для пользователей.

Каждый проект также имеет уровень видимости, который контролирует, кто имеет доступ на чтение страниц проекта или репозитория. Если проект Приватный (Private), владелец должен явно дать доступ на чтение отдельным пользователям. Внутренний (Internal) проект виден любому вошедшему пользователю GitLab, а Публичный (Public) проект видим всем. Это относится как к доступу git “fetch”, так и к доступу к проекту через веб-интерфейс.

#### Хуки

GitLab включает поддержку хуков (перехватчиков, hooks) на уровне проектов и всей системы. В обоих случаях, когда происходит некоторое событие, сервер GitLab выполняет запрос HTTP POST с осмысленным JSON-содержанием. Это отличный способ соединить ваши git-репозитории и инсталляцию GitLab с автоматикой инфраструктуры разработки, такой как сервера непрерывной интеграции, комнаты чатов или инструменты деплоя.

### Базовое использование

Первое, чего вы захотите от GitLab, это создать новый проект. Это достигается нажатием иконки “+” на панели инструментов. Будут запрошены имя проекта, пространство имён, которому он должен принадлежать, и уровень видимости. Большинство из этих настроек можно потом изменить через интерфейс настроек. Нажмите “Создать проект” (“Create Project”), чтобы закончить.

Когда проект создан, вы, наверное, захотите соединить его с локальным git-репозиторием. Каждый проект может быть доступен через HTTPS или SSH, каждый из которых может быть использован для указания удалённого репозитория. Адреса (URL) видимы наверху домашней страницы проекта. Для существующего локального репозитория, следующая команда создаст удалённый репозиторий с именем gitlab и размещением на сервере:

$ git remote add gitlab https://server/namespace/project.git

Если у вас нет локального репозитория, можно просто сделать его:

$ git clone https://server/namespace/project.git

Веб-интерфейс даёт доступ к нескольким полезным видам самого репозитория. Домашняя страница каждого проекта показывает недавнюю активность, а ссылки наверху ведут на список файлов проекта и журнала коммитов.

### Совместная работа

Самый простой метод совместной работы над проектом GitLab — это выдача другому пользователю прямого доступа на запись (push) в git-репозитории. Вы можете добавить пользователя в проект в разделе “Участники” (“Members”) настроек проекта, указав уровень доступа (уровни доступа кратко обсуждались в [Группы](https://git-scm.com/book/ru/v2/ch00/_gitlab_groups_section)). Получая уровень доступа “Разработчик” (“Developer”) или выше, пользователь может невозбранно отсылать свои коммиты и ветки непосредственно в репозиторий.

Другой, более разобщённый способ совместной работы — использование запросов на слияние (merge requests). Эта возможность позволяет любому пользователю, который видит проект, вносить свой вклад подконтрольным способом. Пользователи с прямым доступом могут просто создать ветку, отослать в неё коммиты и открыть запрос на слияние из их ветки обратно в master или любую другую ветку. Пользователи без доступа на запись могут “форкнуть” репозиторий (“fork”, создать собственную копию), отправить коммиты в эту копию и открыть запрос на слияние из их форка обратно в основной проект. Эта модель позволяет владельцу полностью контролировать, что попадает в репозиторий и когда, принимая помощь от недоверенных пользователей.

Запросы на слияние и проблемы (issues) это основные единицы долгоживущих дискуссий в GitLab. Каждый запрос на слияние допускает построчное обсуждение предлагаемого изменения (поддерживая облегчённое рецензирование кода), равно как и общее обсуждение. И те и другие могут присваиваться пользователям или организовываться в вехи (milestones).

Мы в основном сосредоточились на частях GitLab, связанных с git, но это — довольно зрелая система, и она предоставляет много других возможностей, помогающих вашей команде работать совместно, например вики-страницы для проектов и инструменты поддержки системы. Одно из преимуществ GitLab в том, что, однажды запустив и настроив сервер, вам редко придётся изменять конфигурацию или заходить на него по SSH; большинство административных и пользовательских действий можно выполнять через веб-браузер.

**TestLink**

Как и во многих других стартапах в наших проектах, в первый год существования родились горы кода, которые хоть и были неплохо покрыты unit-тестами, но всё же оставались практически не охвачены с точки зрения функционального тестирования. В первое время мы обходились небольшим чек-листом. Проблем не возникало, пока на проекте было несколько тестировщиков, хорошо знающих все тонкости проекта. Но, как это всегда бывает, если один человек из небольшой команды уходит, то необходимо найти ему адекватную замену. И вот тут возникли большие трудности.  
  
Наш проект имеет очень сложную структуру: один только раздел может иметь документацию более чем на 100 страниц. Плюсом, конечно, идет паровоз из недокументированных фич и индивидуальных “хотелок” заказчика, которые начинающие члены команды могут принять за баги.  
  
К счастью, в нашей компании существует и активно развивается направление по работе со студентами и начинающими специалистами. Недостаток с кадрами легко можно устранить, но вот как быстро подключить новичка к работе? Было решено внедрить подробные тест-кейсы, описывающие весь набор функций проекта. Для решения задачи на первом этапе мы решили не гнаться за платными решениями, а использовать одну из бесплатных систем управления тестами. Выбор пал на TestLink.

#### Внедрение

Сам TestLink написан на PHP, и особых проблем с установкой не возникло. Мы создали базу данных, дали права на папочки, и установка прошла как по маслу.  
  
Далее мы завели в системе проект, создали структуру проекта и начали писать тесты. Сама цель внедрения данной системы заключалась в том, чтобы мы смогли выявить полезность или бесполезность подобных систем для наших проектов.

#### Как мы используем ресурсы команды и TestLink

После того, как мы написали и проверили свои тесты, пришло время опробовать систему на людях. Мы распределили готовые тесты среди начинающих сотрудников и стажеров. Первый же прогон показал, что использование подробных тестов позволило найти множество багов в потаенных местах проекта, куда обычно не дотягивались руки в силу приоритетности работы других фич.  
  
TestLink в настоящее время используется не в полную силу. Он включает возможность использования не только тестов, но и условий. Использование условий видится очень перспективной возможностью данной системы. Условия относятся к тестам как многие ко многим, что позволяет реализовать несколько следующих моделей использования:

* Связать тесты, затрагивающие один и тот же набор функций, одним условием. Это позволит эффективно и быстро создавать инкрементные прогоны, например, для горячих исправлений.
* Вынести в условия некоторые ограничения, которые должны быть достигнуты для удачного прохождения теста.

Ввиду того, что наши тесты пока не успели обзавестись условиями, ниже мы не будем о них упоминать.

#### Влияние TestLink на качество тестирования

Качество тестирования после внедрения TestLink, однозначно, возросло. Мы очень благодарны этой системе управления тестами за подробные отчеты после прогона тестов и за то, что она познакомила нас с тем, как необходимо подходить к тестированию. Сейчас мы имеем неплохую отлаженную систему тестирования своего проекта, благодаря которой новый сотрудник спустя буквально полдня будет полностью включен в работу с проектом и, самостоятельно сделав прогон всех тестов, сможет понять до 80% функций проекта.

#### Плюсы и минусы TestLink

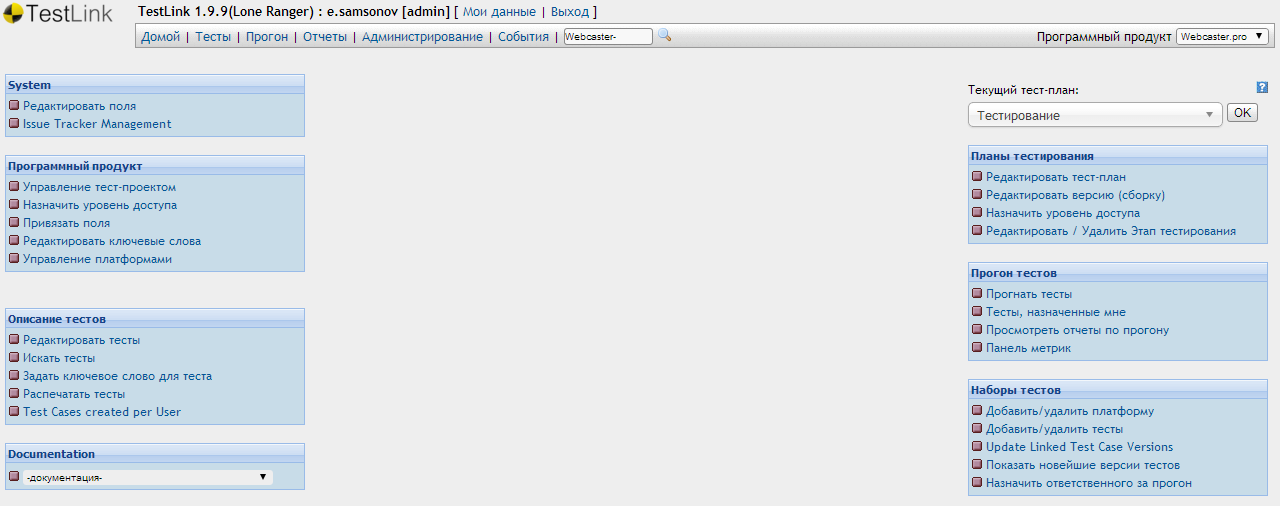
Несмотря на все достоинства, столкнулись мы и с рядом недостатков системы. Естественно, всё очень субъективно, и для других компаний наши претензии будут казаться необоснованными. Мы не исключаем, что возможно, некоторые из недостатков можно устранить, изучив документацию или покопавшись немного в коде, но цель стояла немного иная. Нам было необходимо быстро оценить полезность подобной системы внутри нашей компании.  
Достоинства:

* Развитая система ролей. Очень гибко можно настроить доступы для разных сотрудников и так же оперативно отнять эти самые права.
* Простая и понятная система структурирования проекта. Проект — Группа тестов — Тест — Шаги (Рис. 2).
* Легкое распределение тестов между тестировщиками. Можно выбирать группы тестов или отдельные тесты, а также просто скопировать распределение тестов из прошлого релиза (Рис. 3).
* Красивые отчеты по прогонам в разных форматах, в том числе с возможностью отправки по почте всем ответственным и заинтересованным результатами сотрудникам (Рис. 4).

Недостатки:

* Нет ссылок на отдельные страницы внутри системы. Вместо того, чтобы послать коллеге ссылку, приходится называть проект, группу тестов и номер теста.
* Проблема с версионностью тестов. В нашем случае с настройками по умолчанию невозможно просмотреть разные версии тестов из окна прогона тестов или в разделе редактирования, выбрав Compare versions.
* Интеграция с Redmine. Она есть, но не удалось ее полностью настроить на работу с нашим Redmine с ходу в обе стороны (необходимы или дополнительные плагины, или разрешения).
* Юзабилити. Без подготовки не очевидно, какие кнопочки нажимать, чтобы всё работало (Рис. 1).

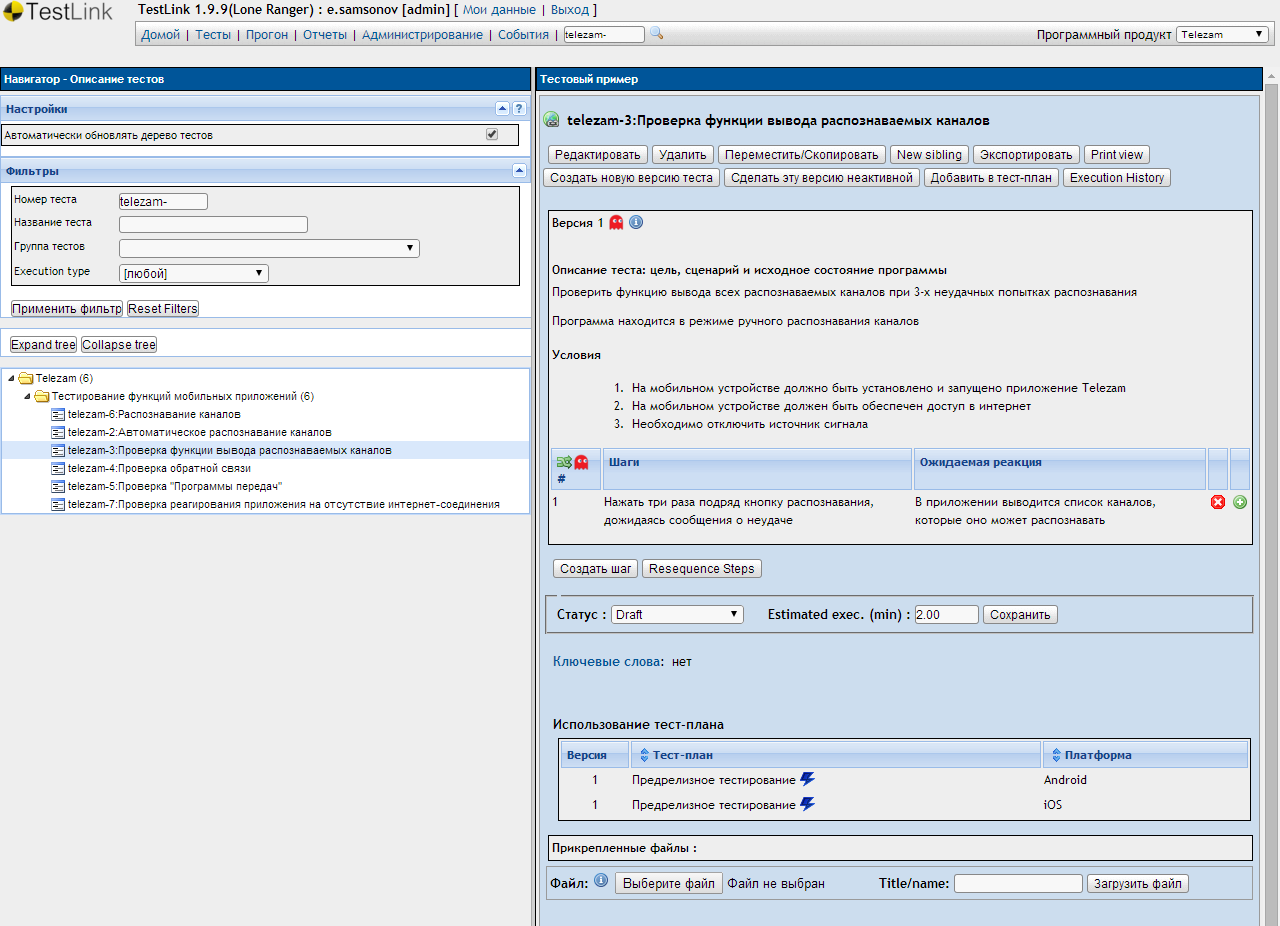
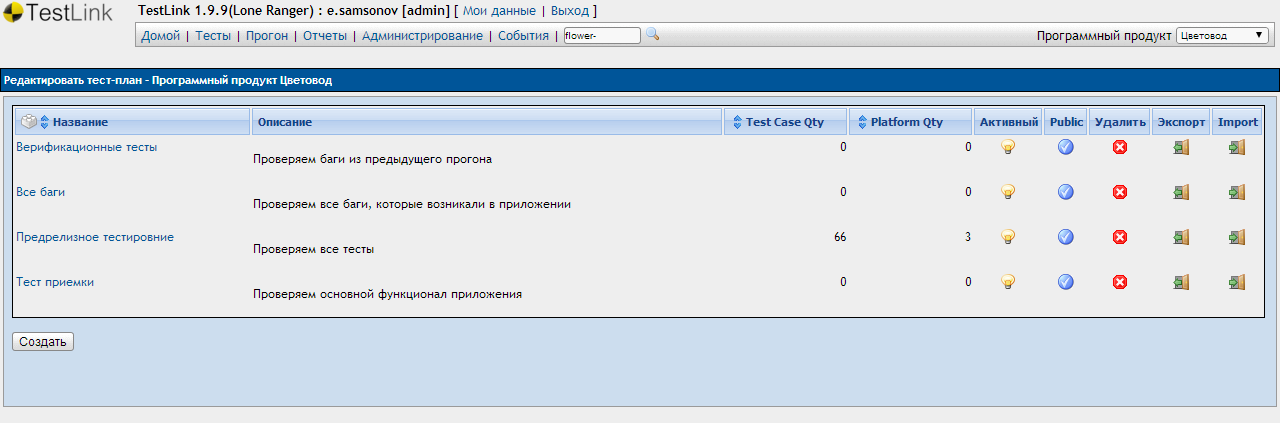
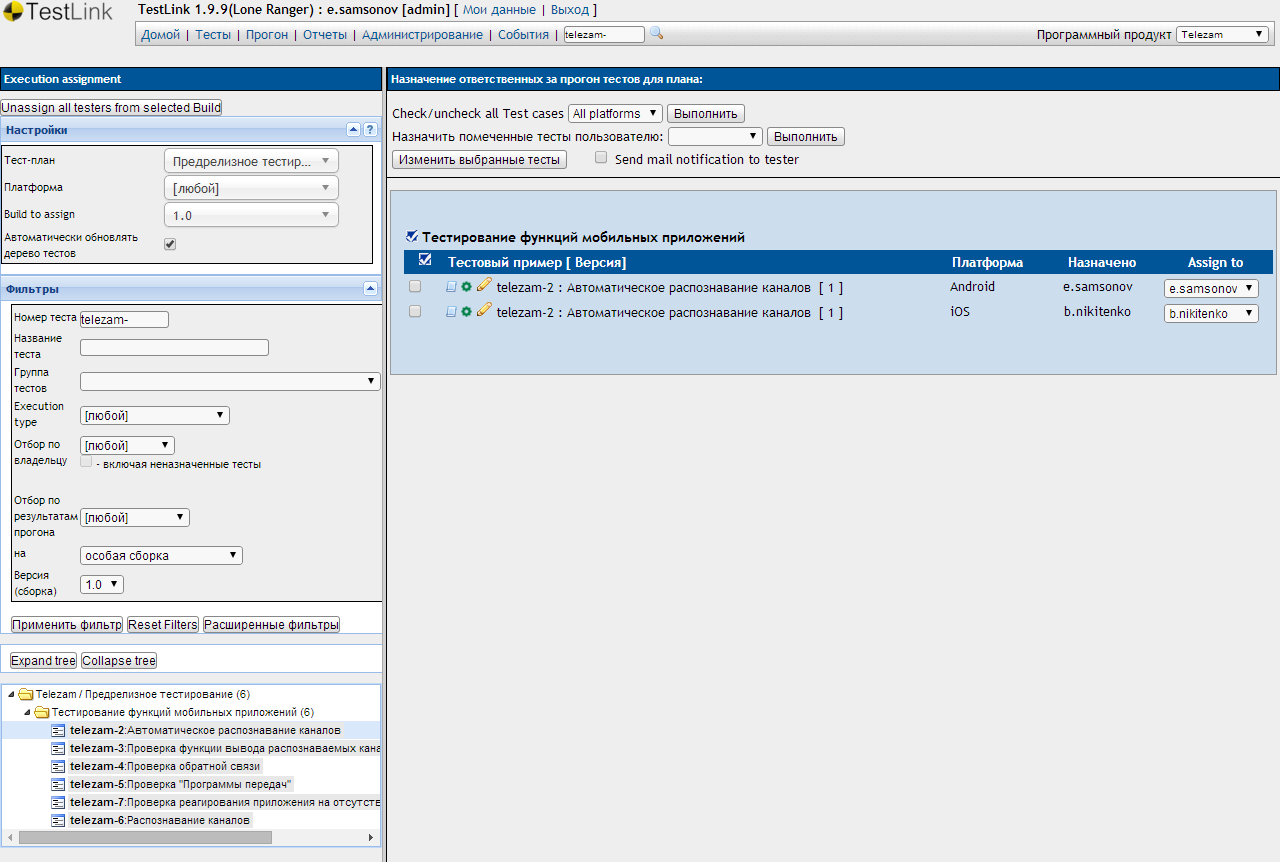
#### Мини-обзор возможностей TestLink

Продемонстрируем работу системы на примерах из наших проектов. Полный обзор системы делать не имеет смысла. Основной посыл, в данном случае, направлен на то, чтобы показать, как выглядит система внутри, и сможете ли вы с ней ужиться или не стоит даже устанавливать. Поэтому мы опустим этапы установки системы на сервер, а также этапы заведения проекта.   
  
Первое, что видит тестировщик, открыв TestLink ‒ это стартовый экран (Рис. 1).  
  
  
*Рис. 1 ‒ Стартовый экран работы с проектом в TestLink*  
  
Отсюда начинается любая активность в проекте. Здесь выполняется:

* редактирование/создание тестов;
* назначение пользователей, ответственных за прогон;
* создание сборок;
* редактирование настроек самого проекта.

Страница редактирования тестов включает в себя такие возможности, как:

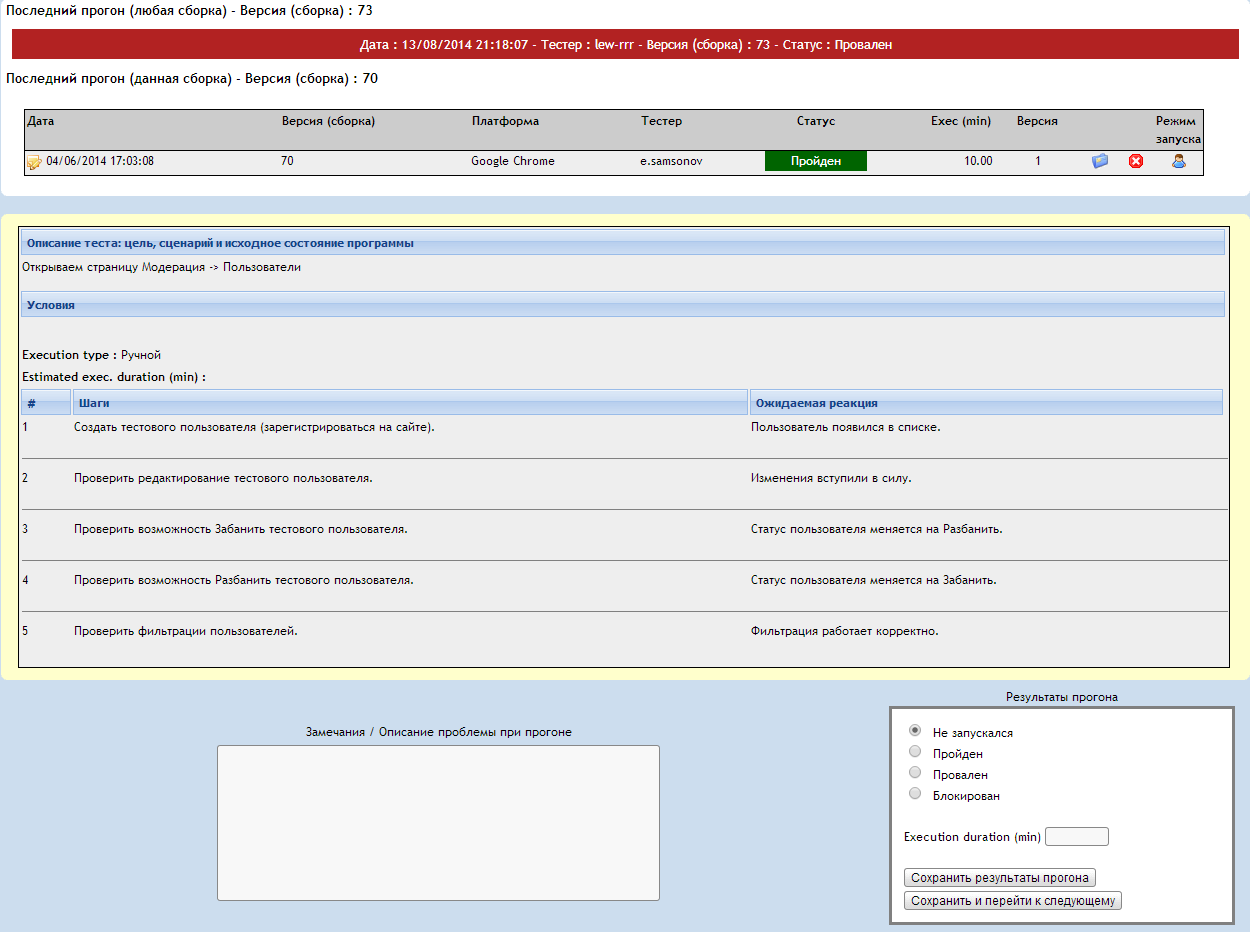
* создание/редактирование нового теста;
* создание/редактирование группы тестов;
* работа с версиями тестов;
* создание/редактирование шагов в тестах.

  
*Рис. 2 ‒ Редактирование тестов в TestLink*  
  
Далее рассмотрим распределение тестов по тест-планам.   
  
Для этого следует разбить тесты на различные группы, которые затем могут использоваться для прогонов, подходящих под заданные условия. На рисунке 3 видно, что в данном случае тесты распределены по видам тестирования, которые мы проводим для выбранного проекта. Для других проектов у нас есть различные варианты разбиения на тест-планы, например, на проектах, где нет разделения тестов по группам и нужно каждый раз прогонять все тесты без исключения, мы завели тест-планы для Production и Staging окружений.  
  
  
*Рис. 3 ‒ Редактирование тест-плана*  
  
Назначать тесты членам вашей команды очень просто. В левой части окна (рис. 4) можно выбрать как один тест, так и группу тестов, а также использовать фильтры для поиска нужных тестов. Затем можно назначить тестровщика, ответственного за прогон конкретного теста на конкретной платформе, с помощью чек-боксов и выпадающих списков в правой части раздела.  
  
  
*Рис. 4 ‒ Распределение задач по тестировщикам в TestLink*  
  
После распределения тестов прогон выглядит, как на рисунке 5. На нем доступно следующее: 

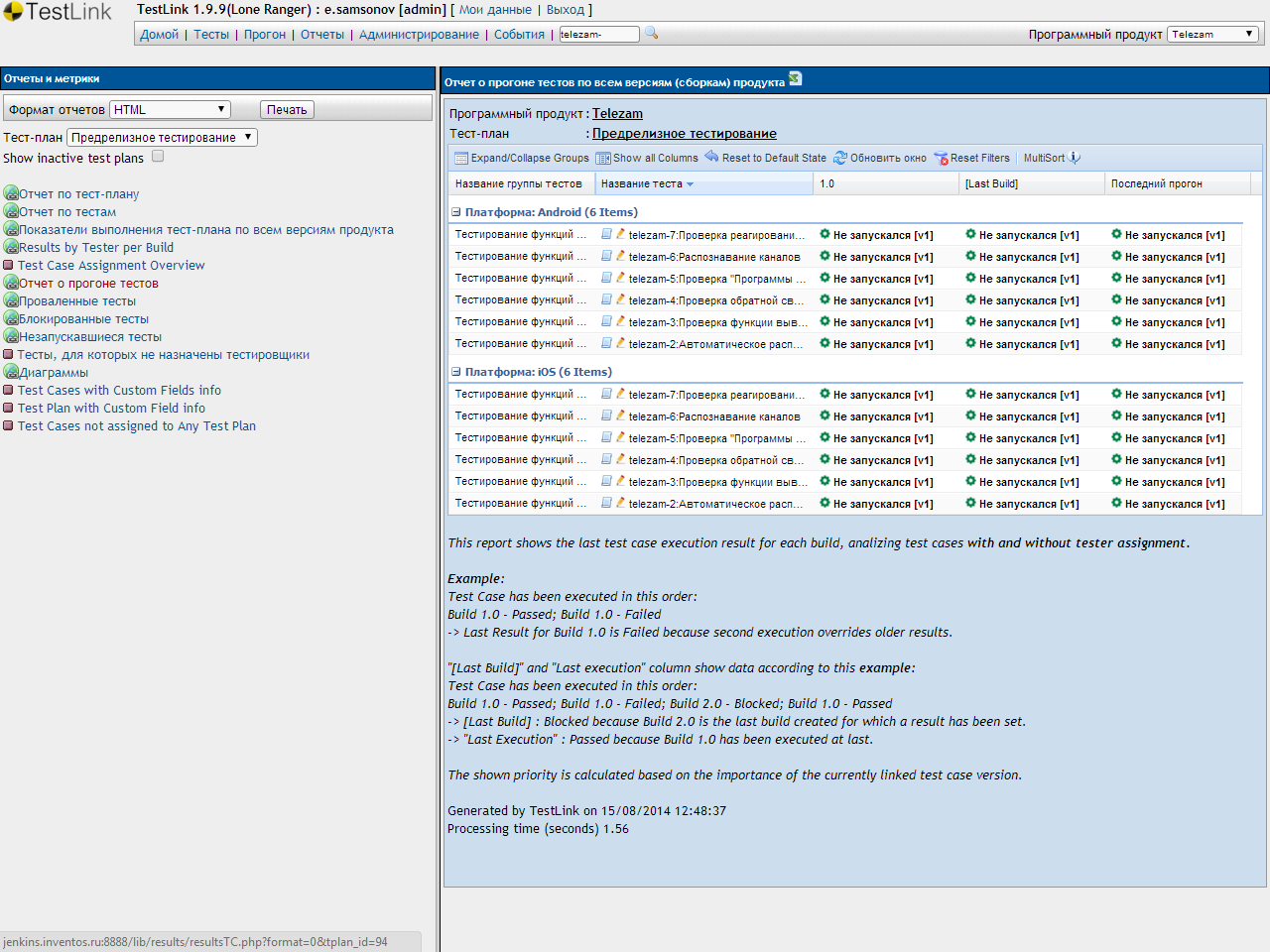
* Описание всех шагов, которые необходимо выполнить;
* Поле для ввода замечаний;
* Отметка о пройденности теста.

Кроме перечисленных элементов, можно увидеть и дополнительную информацию:

* Предыдущие прогоны;
* Платформа, на которой необходимо осуществить проверку;
* Описание сборки продукта;
* Описание тест-плана (не отображено на рисунке 5).

  
*Рис. 5 ‒ Прогон тестов в TestLink*  
  
Сразу после распределения тестов мы уже можем наблюдать за появлением данных по отчетам тестировщиков в разделе “Отчеты” TestLink. В разделе представлено множество опций:

* Слежение за результатами прогона;
* Экспорт в популярные форматы;
* Отправка отчетов по почте заинтересованным лицам;
* Группировка тестов по различным признакам, представленным в левой части рисунка 6.

  
*Рис. 6 ‒ Система формирования отчетов о прогонах в TestLink*

#### Заключение

Надеюсь, наш опыт внедрения TestLink откроет для вас новые горизонты. Возможно, кто-то наконец откажется от ведения документации в таблицах. Нам бы хотелось услышать в комментариях ваше мнение о TestLink или о подобных разработках, как вы их используете, какова степень интеграции ваших инструментов, что удобно и неудобно.

# Объяснение SQL объединений JOIN: LEFT/RIGHT/INNER/OUTER

Категория: / [DEV Блог](http://www.skillz.ru/dev/) / [PHP (LAMP)](http://www.skillz.ru/dev/php/)

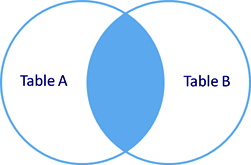
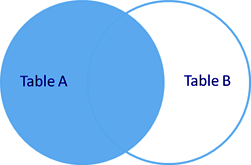
Разберем пример. Имеем две таблицы: пользователи и отделы.

U) users               D) departments  
id name       d\_id     id  name  
-- ----       ----     --  ----  
1  Владимир    1       1  Сейлз  
2  Антон       2       2  Поддержка  
3  Александр   6       3  Финансы  
4  Борис       2       4  Логистика  
5  Юрий        4

**SELECT** u.id, u.name, d.name **AS** d\_name  
**FROM** users u  
**INNER** **JOIN** departments d **ON** u.d\_id = d.id

Запрос вернет объединенные данные, которые пересекаются по условию, указанному в INNER JOIN ON <..>.   
В нашем случае условие <таблица\_пользователей>.<идентификатор\_отдела> должен совпадать с <таблица\_отделов>.<идентификатор>  
  
В результате *отсутствуют*:  
  
- пользователь Александр (отдел 6 - не существует)  
- отдел Финансы (нет пользователей) 

id name       d\_name  
-- --------   ---------  
1  Владимир    Сейлз  
2  Антон       Поддержка  
4  Борис       Поддержка  
3  Юрий        Логистика

  
  
*рис. Inner join*  
  
Внутреннее объединение INNER JOIN (синоним JOIN, ключевое слово INNER можно опустить).   
  
Выбираются только совпадающие данные из объединяемых таблиц.   
  
  
Чтобы получить данные, которые подходят по условию частично, необходимо использовать   
  
внешнее объединение - OUTER JOIN.   
  
Такое объединение вернет данные из обеих таблиц (совпадающие по условию объединения) ПЛЮС дополнит выборку оставшимися данными из внешней таблицы, которые по условию не подходят, заполнив недостающие данные значением NULL.  
  
  
  
*рис. Left join*  
  
Существует два типа внешнего объединения OUTER JOIN - LEFT OUTER JOIN и RIGHT OUTER JOIN.   
  
Работают они одинаково, разница заключается в том что LEFT - указывает что "внешней" таблицей будет находящаяся слева (в нашем примере это таблица users).   
Ключевое слово OUTER можно опустить. Запись LEFT JOIN идентична LEFT OUTER JOIN.

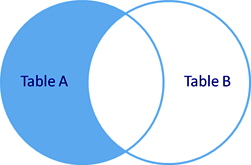
**SELECT** u.id, u.name, d.name **AS** d\_name  
**FROM** users u  
LEFT **OUTER** **JOIN** departments d **ON** u.d\_id = d.id

Получаем полный список пользователей и сопоставленные департаменты.

id      name          d\_name  
--      --------      ---------  
1       Владимир      Сейлз  
2       Антон         Поддержка  
3       Александр     NULL  
4       Борис         Поддержка  
5       Юрий          Логистика

Добавив условие 

**WHERE** d.id **IS** **NULL**

в выборке останется только 3#Александр, так как у него не назначен департамент.  
  
  
  
*рис. Left outer join с фильтрацией по полю*  
  
  
RIGHT OUTER JOIN вернет полный список департаментов (правая таблица) и сопоставленных пользователей.

**SELECT** u.id, u.name, d.name **AS** d\_name  
**FROM** users u  
RIGHT **OUTER** **JOIN** departments d **ON** u.d\_id = d.id

id      name        d\_name  
--      --------    ---------  
1       Владимир    Сейлз  
2       Антон       Поддержка  
4       Борис       Поддержка  
NULL    NULL        Финансы  
5       Юрий        Логистика

Дополнительно можно отфильтровать данные, проверяя их на NULL.

**SELECT** d.id, d.name  
**FROM** users u  
RIGHT **OUTER** **JOIN** departments d **ON** u.d\_id = d.id   
**WHERE** u.id **IS** **null**

В нашем примере указав WHERE u.id IS null, мы выберем департаменты, в которых не числятся пользователи. (3#Финансы)  
  
  
**Все примеры вы можете протестировать здесь:**  
  
[SQLFiddle](http://www.skillz.ru/r.php?url=http://sqlfiddle.com/#!2/659f69/1)  
  
  
**Cross/Full Join**  
  
FULL JOIN возвращает `объединение` объединений LEFT и RIGHT таблиц, комбинируя результат двух запросов.  
  
CROSS JOIN возвращает перекрестное (декартово) объединение двух таблиц. Результатом будет выборка всех записей первой таблицы объединенная с каждой строкой второй таблицы. Важным моментом является то, что для кросса не нужно указывать условие объединения.  
  
**Дублирование строк при использовании JOIN**  
  
При использовании объединения новички часто забывают что результирующая выборка может содержать дублирующиеся данные!  
Если вам нужна одна запись, делайте объединение с подзапросом

**SELECT** t1.\*, t2.\* **from** left\_table t1 left **join** (**select** \* **from** right\_table **where** some\_column = 1 **limit** 1) t2**ON** t1.id = t2.join\_id

**Self Join**  
  
Выборка из одной и той же таблицы для нескольких условий.  
  
Рассмотрим задачку от яндекса:   
  
Есть таблица товаров.

**CREATE** **TABLE** `ya**\_**goods` (  
  `id` **int**(11) **unsigned** **NOT** **NULL** **AUTO\_INCREMENT**,  
  `name` **varchar**(64) **NOT** **NULL**,  
  **PRIMARY KEY** (`id`)  
) **ENGINE**=**InnoDB** **DEFAULT** **CHARSET**=utf8;  
**insert** **into** ya\_goods **values** (1, 'яблоки'), (2, 'яблоки') ,(3, 'груши'), (4,'яблоки'), (5, 'апельсины'), (6,'груши');

Она содержит следующие значения.

`id`    `name`  
1       Яблоки  
2       Яблоки  
3       Груши  
4       Яблоки  
5       Апельсины  
6       Груши

Напишите запрос, выбирающий уникальные пары `id` товаров с одинаковыми `name`, например:  
  
(1,2), (4,1), (2,4), (6,3)...  
  
При решении задачи необходимо учесть, что пары (x,y) и (y,x) — одинаковы.  
  
Решение:

**SELECT** g1.id id1, g2.id id2  
*-- CONCAT('(', LEAST(g1.id, g2.id), ',', GREATEST(g1.id, g2.id), ')') row*  
**FROM** ya\_goods g1   
**INNER** **JOIN** ya\_goods g2 **ON** g1.name = g2.name   
**WHERE** g1.id <> g2.id  
**GROUP BY** LEAST(g1.id, g2.id), GREATEST(g1.id, g2.id)  
**ORDER BY** g1.id;  
   
*-- или без группировки (быстрее)*  
   
**SELECT** **DISTINCT** CONCAT('(', LEAST(g1.id, g2.id), ',', GREATEST(g1.id, g2.id), ')') row  
**FROM** ya\_goods g1   
**INNER** **JOIN** ya\_goods g2 **ON** g1.name = g2.name   
**WHERE** g1.id <> g2.id

Объединяем таблицы ya\_goods по одинаковому полю `name`, группируем по уникальным idентификаторам и получаем результат.  
  
(1,2)(1,4)(2,4)(3,6)  
  
  
**Множественное объединение multi join**  
  
Пригодится нам, если необходимо выбрать более одного значения из таблиц для нескольких условий.  
  
Пример: набор вариантов (вес, объем) товаров.   
Продукты в таблице products, Варианты - таблица product\_options, Значения вариантов - таблица product2options  
Необходимо: фильтровать продукты по дате, и имеющимся вариантам

**CREATE** **TABLE**  `products` (  
  `id` **int**(11),  
  `title` **varchar**(255),  
  `created**\_**at` **datetime**  
)  
   
**CREATE** **TABLE** `product**\_**options` (  
  `id` **int**(11),  
  `name` **varchar**(255)  
)  
   
**CREATE** **TABLE** `product2options` (  
  `product**\_**id` **int**(11),  
  `option**\_**id` **int**(11),  
  `value` **int**(11)  
)

Тестовые данные

**INSERT** **INTO** `products` (`id`, `title`, `created**\_**at`) **VALUES**  
        (1, 'Кружка', '2009-01-17 20:00:00'),  
        (2, 'Ложка', '2009-01-18 20:00:00'),  
        (3, 'Тарелка', '2009-01-19 20:00:00');  
   
**INSERT** **INTO** `product**\_**options` (`id`, `name`) **VALUES**  
        (11, 'Вес'),  
        (12, 'Объем');  
   
**INSERT** **INTO** `product2options` (`product**\_**id`, `option**\_**id`, `value`) **VALUES**  
        (1, 11, 200),  
        (1, 12, 250),  
        (2, 11, 35),  
        (2, 12, 15),  
        (3, 11, 310),  
        (3, 12, 300),  
        (2, 11, 45),  
        (2, 12, 25);

Пример: выбрать товары,   
добавленные после 17/01/2009 в следующих вариантах:  
- вес=310, объем=300  
- вес=35, объем=15  
- вес=45, объем=25  
- вес=200, объем=250  
  
Просто перечислить условия вариантов в подзапросе/джоине через OR/AND не сработает,   
необходимо осуществить объединение таблиц вариантов равное количеству этих самых вариантов (у нас - 2: объем и вес)

**SELECT** p.\*, po1.name 'P1', p2o1.**value**, po2.name 'P2', p2o2.**value**       
   
**FROM** products p         
   
**INNER** **JOIN** product2options p2o1 **ON** p.id = p2o1.product\_id  
**INNER** **JOIN** product\_options po1  **ON** po1.id = p2o1.option\_id  
   
**INNER** **JOIN** product2options p2o2 **ON** p.id = p2o2.product\_id  
**INNER** **JOIN** product\_options po2  **ON** po2.id = p2o2.option\_id  
   
**WHERE** p.created\_at > '2009-01-17 21:00'   
  **AND** ( *-- тарелка#3*  
  p2o1.option\_id = 11 **AND** p2o1.**value** = 310   
  **AND** p2o2.option\_id = 12 **AND** p2o2.**value** = 300  
  **OR**  *-- ложка#2*  
  p2o1.option\_id = 11 **AND** p2o1.**value** = 35   
  **AND** p2o2.option\_id = 12 **AND** p2o2.**value** = 15             
  **OR**  *-- ложка#2*  
  p2o1.option\_id = 11 **AND** p2o1.**value** = 45   
  **AND** p2o2.option\_id = 12 **AND** p2o2.**value** = 25     
  **OR**  *-- кружка#1 не попадает по дате*  
  p2o1.option\_id = 12 **AND** p2o1.**value** = 250   
  **AND** p2o2.option\_id = 11 **AND** p2o2.**value** = 200  
  )    
;

Результ выборки:

id      title        created\_at                P1        value    P2            value  
2       Ложка        2009-01-18 20:00:00       Вес        35      Объем         15  
3       Тарелка      2009-01-19 20:00:00       Вес        310     Объем         300  
2       Ложка        2009-01-18 20:00:00       Вес        45      Объем         25  
   
-- не попадает по дате  
1       Кружка          2009-01-17 20:00:00     Объем   250     Вес             200

[Этот пример на SQLFiddle](http://www.skillz.ru/r.php?url=http://sqlfiddle.com/#!9/49866/2)  
  
**UPDATE и JOIN**  
  
Объединение можно использовать совместно с UPDATE.   
Например, имеем таблицу houses (id, title, area). Нужно выбрать title, если в нем встречается `число м2`, заменить поле area, если оно меньше. Т.к. в mysql отстутсутствует поддержка регулярных выражений, нужно немного поколдовать с locate и substr.  
В подзапросе выбираем интересующие нас данные, и в финальной стадии осуществляем обновление данных подходящий по критерию (p5 > area).

**UPDATE** houses base  
**INNER** **JOIN** (  
*-- Антарис аренда офиса 1594 м2, по ставке 12700 руб. м2/год -> 1594*  
**SELECT**  
        id,  
        @baseString := title title,  
        @areaTitleEnd := LOCATE(' м2', @baseString) **as** p2,  
        @tmpString := LTRIM(REVERSE(SUBSTR(@baseString, 1, @areaTitleEnd))) **as** p3,  
        @areaTitleBegin := LEFT(@tmpString, -1 + LOCATE(' ', @tmpString)) **as** p4,  
        @**value** := CAST(REVERSE(@areaTitleBegin) **as** **UNSIGNED**) **as** p5  
   
        **FROM** ga\_pageviews  
        **WHERE** title **like** '**%**м2**%**'  
) calc **USING** (`id`)  
**SET** base.area = calc.p5  
**WHERE** base.area < calc.p5

**DELETE и JOIN**  
  
Рассмотрим пример с удалением дубликатов. Есть таблица tableWithDups (id, email). Нужно удалить строки с одинаковыми email:

**DELETE** tableWithDups  
**FROM** tableWithDups  
**INNER** **JOIN** (  
        **SELECT** MAX(id) **AS** lastId, email  
        **FROM** tableWithDups  
        **GROUP BY** email  
        **HAVING** COUNT(\*) > 1  
        ) dups **ON** dups.email = tableWithDups.email  
**WHERE** tableWithDups.id < dups.lastId;

Последние два примера не совместимы с ANSI SQL, но работают в mySQL.  
  
За бортом статьи остались смежные объединениям (а также специфичные для определенных базданных темы):  
SELF JOIN, FULL OUTER JOIN, CROSS JOIN (CROSS [OUTER] APPLY), операции над множествами UNION [ALL], INTERSECT, EXCEPT и т.д.