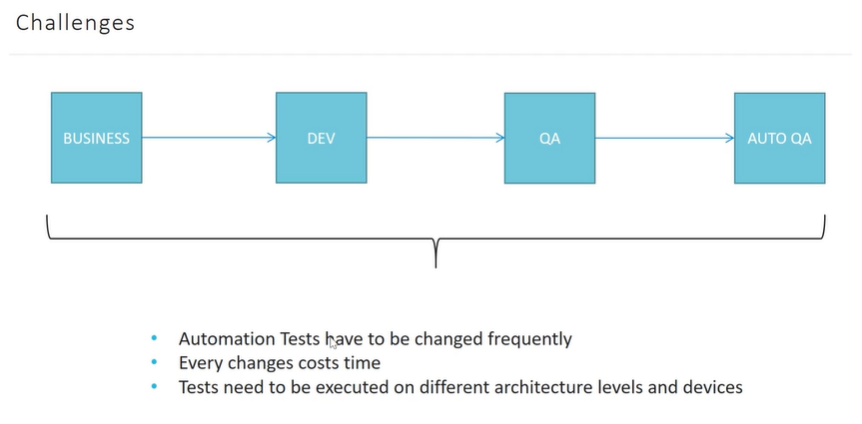
**Automation Framework**

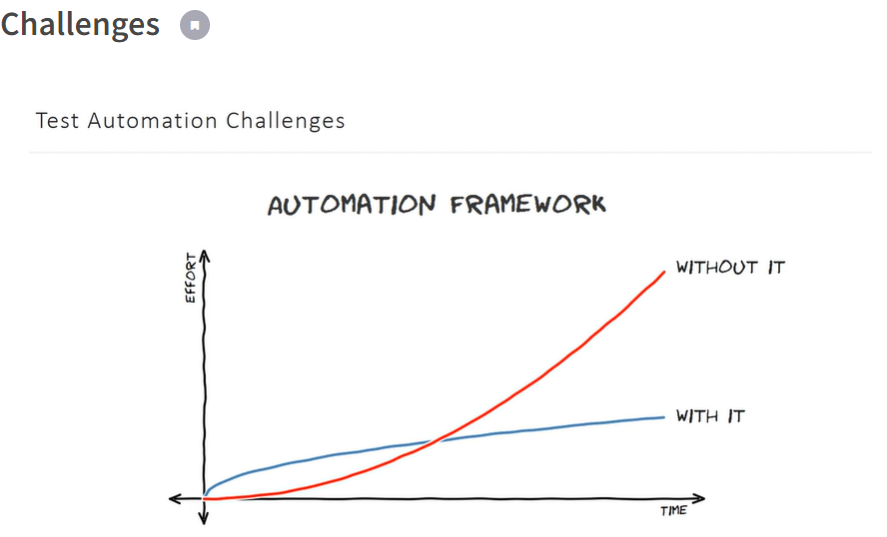
**Challenges**

1. Автотесты нужно менять достаточно часто – приходят новые изменения, фичи, которые нужно покрывать, появляются дефекты. Бизнес-логика меняется. Изменение на каждом уровне – изменение QA (Quality Assurance, обеспечение качества), локаторов
2. Любое изменение требует времени – время на правки в коде
3. Тесты должны выполняться на различных архитектурных уровнях и различных устройствах – веб, мобильное устройство



Если объединить тесты в общее решение, в общую систему, в один проект, это требует больше времени для написания сценариев – всегда быстрее написать в main, чем красиво разнести по классам, вынести переменные в константы

Видим на графике: линейное расположение тестов в коде на старте дешево, но с каждым новым тестом стоимость будет расти. С фреймворком начать тяжелее, но инвестиция в него позволяет быть гибче – позволяет решать задачи быстрого изменения автотестов



**Approaches. Типы фрейморков**

Подходы к построению Test Automation Framework

1. Record & Playback

Основан на записи действий пользователя и их воспроизведений

- Простые и дешевые

- Selenium IDE – плагин для браузеров – окошечко, в котором можно запускать сценарии: Selenium записывает действия браузера, которые можно воспроизвести

- Не получится промышленная автоматизация. Подходит для быстрых задач – или если человек не является носителем знаний программирования

- Нет интеграции с файловой системы, есть условия, но данные решения не проходят для enterprise

1. Reusable scripts

Переиспользованные скрипты (запись действий), их запуск в разном порядке

1. Keyword-driven
2. Data-driven

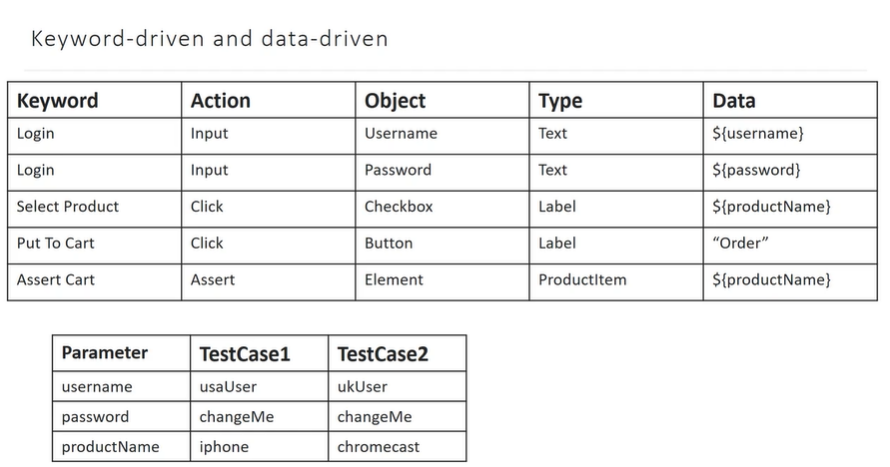
Как правило идут в паре. Основной элемент – таблица с ключевыми словами. Это тест кейс. Таблица читается построчно и выполняется построчно программой по автоматизации

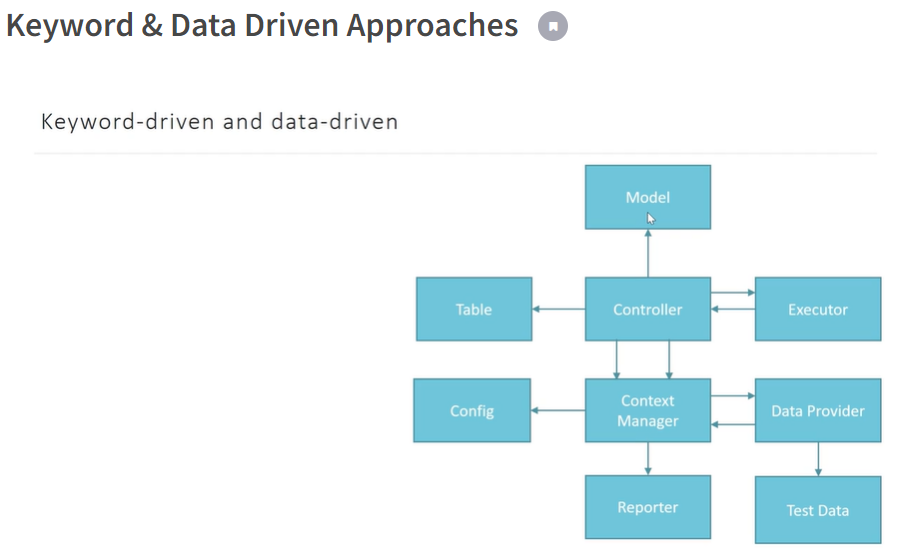
Keyword – атомарное бизнес-действие (поместить товар в корзину, login)

Особенность: на keyword можно строить новые и новые сценарии, чем больше keyword, тем быстрее строится новый тест-кейс. В коде будут часто повторяться одни и те же методы – эти вещи инкапсулируются одним словом keyword. Когда их станет много (например, проверка есть ли в корзине, нет ли в корзине …), они будут рабочие, можно создать новый тест-кейс, не написав строчек кода, просто переставив keyword в другом порядке

Тест состоит из переиспользуемых кирпичиков – тесты легко параметризовать новыми данными, которые обычно хранятся в отдельной таблице

- Это дешево и просто





Ядро Controller – работает с таблицей тест-кейсов, взаимодействует с моделью приложения или системы тестирования. Используя данные модели, таблицы keyword и данные Context Manager (конфигурация, тестовые данные), отправляет данные на Executor – сущность, выполняющая всю работу. Это может быть WebDriver – делает действие тест-кейса

По результату выполнения Executor отправляет данные обратно. Выполнил – отправил обратно. Controller передает это Context Manager, а тот в свою очередь формирует отчет Reporter, на основании использованных данных из таблицы и результирующих данных от Executor

Это упрощенная схема

Плюсы:

- легко дизайнить новые тесты, можно привлекать людей, которые не пишут код

- быстрое и дешевое создание нового сценария

Минусы:

- количество keyword будет расти, их надо отслеживать (трекать)

- надо хранить список известных и рабочих keyword, выбирать именно их, храним карту списков легальных, легитимных keyword, не дублируя функционал

- тяжело проводить Debug – Controller, читает таблицу построчно, вызывает методы построчно, отлаживать сложно

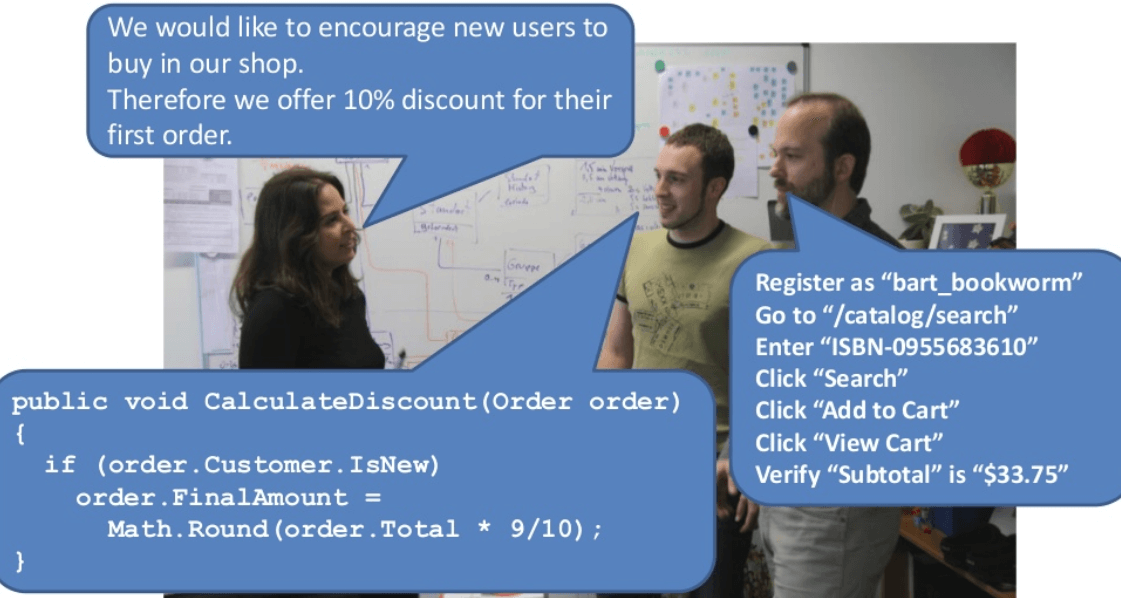
- составлять тесты не интересно, это рутинно выглядит, бюрократизированно

1. Behaviour-driven

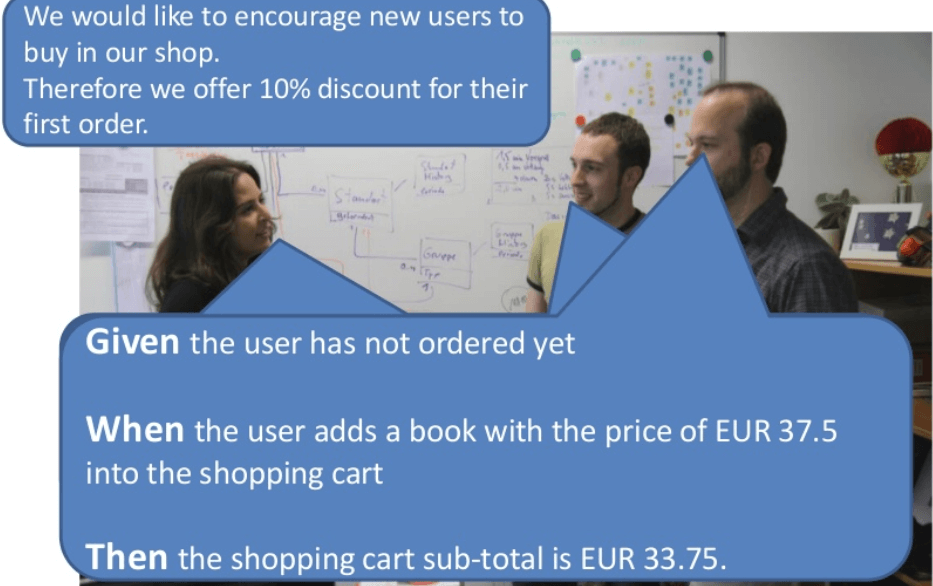
Дисклеймер: BDD подход разработки приложений и разработки автоматизированных тестов

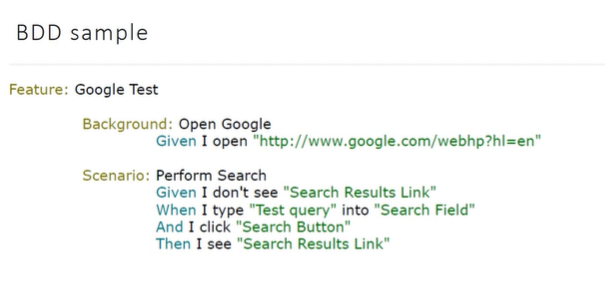
Основная идея выходит из Gherkin синтаксиса, описывающего определенные фичи. Сначала описываем как ведет себя система, затем имплементим это

До Gherkin



После





У нас есть фича, содержащая ряд сценариев – посвящена работе гугла

Background: предусловие, то, что выполняется перед каждым сценарием. Сценариев может быть несколько

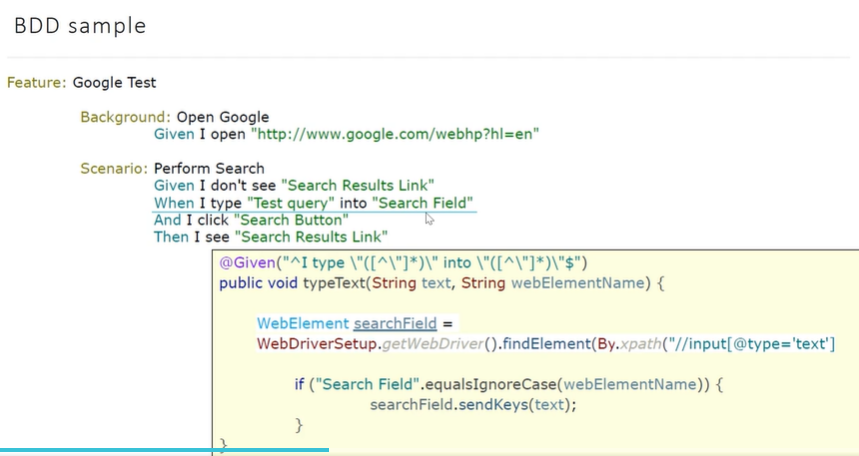
Given – то, что дано, что имеем

Given – When (когда) – And (и) – Then (тогда) описываем, что должно быть сделано

Фактически на английском языке



Каждая строка сценария парсится, в зависимости от результата парсинга выполняются определенные действия:



В рамке приведен пример метода, который соответствует подчеркнутой строке. Используется регулярное выражение, реагирует на слова «I type into» - реагирует на действие, без данных. Используем аннотацию @Given – она ищет нужный метод, соответствующий данному действию. **Тестовые данные попадают в качестве параметров – это поле, куда вводим, и текст, который будем писать**

**Далее работает классический код ЯП – разыскиваем веб-элемент, отправляем в него текст**. Текст передается параметром, и он изменяем

Так парсится каждая строка. Обычно вызывается с помощью фреймворков – Cucumber, **SpecFlow для .NET**, Behave

Плюсы:

- четко показывает, готовы ли мы к релизу, сразу можем сказать, какие фичи работают, а какие нет

**Согласно BDD, если у нас все сценарии проходят, значит фича сделана** – DEFINITION OF DONE

Это документация, которую можно читать, чтобы понять, что происходит. Это условие выполненной работы разработчика, это тест-кейс ручного тестирования и автоматизации – **все остаются С ОДНОЙ И ТОЙ ЖЕ ИНФОРМАЦИЕЙ: видим и делаем так, как написано. ЕДИНЫЙ ИСТОЧНИК ДАННЫХ – смотри фичу, там все есть**

Минусы: создавать фрейморвки по BDD сложнее, нужна инженерная зрелость, бизенс должен быть готов писать эти вещи, высокий риск опечаток. Нужна договоренность о нейминге на высоком уровне (type or set / when I type or when I set)

Ориентирован на бизнес, Data-Driver, чистые отчеты, отслеживаемость

Тяжело расширяется, требует консолидации, тяжело поддерживать. Это накладные расходы, если бизнес не поддерживает

1. Hybrid – большая часть фреймворков в промышленности, содержит элементы разных методологий.

Включает различные модули и элементы

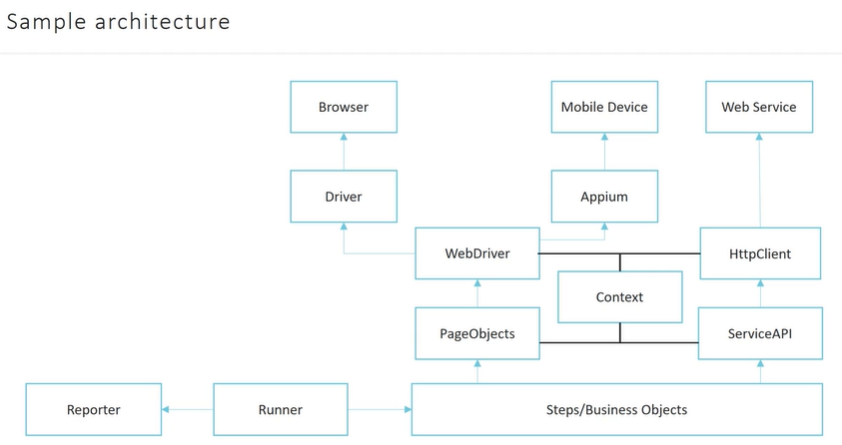
Пример 1 – у нас есть некая модель, в виде бизнес-объектов, и есть WebDriver с PageObject, не только веб-автоматизация, есть автоматизация на уровне сервисов и клиентов API, он вызывает определенные веб-сервисы

WebDriver через драйвер обращается к браузеру

Может быть мобильная автоматизация – Appium

Между ними – контекст: тесты на разных уровнях содержат состояния

Бизнес-объекты вызываются с помощью Runner, executor – выполняет непосредственно действия, это может быть XUnit, NUnit и так далее, выполняет действия при помощи драйвера, пишет отчет по результатам действий, он не знает про нашу «кухню», просто запускает сценарии

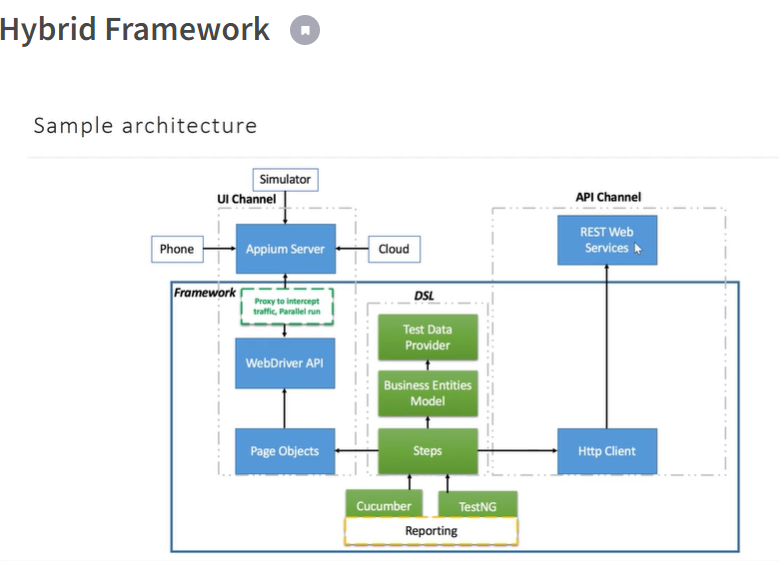


Пример 2

Есть HTTP-клиент, обращается к микросервисной архитектуре, есть WebDriver, PageObject, мобильная автоматизация Appium

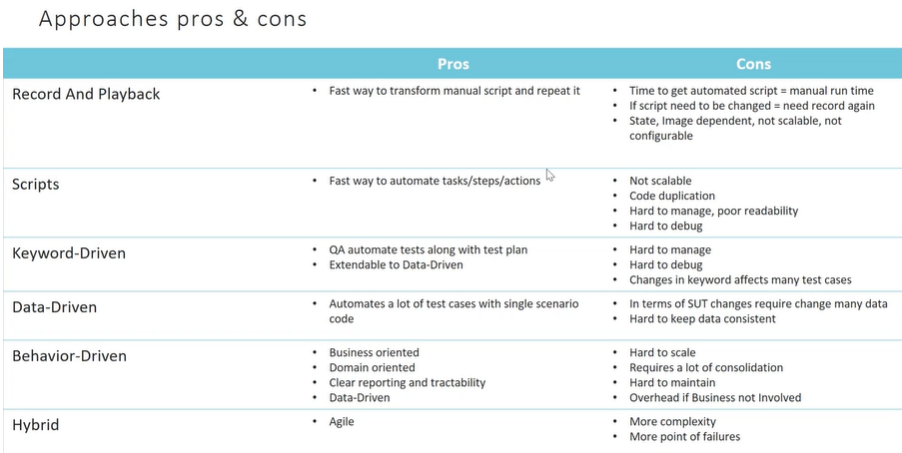
Runner на уровне TestNG XUnit или Cucumber, они вместе делают Report и определяются к Step – это атомы, кирпичики, из которых состоит тест

Step взаимодействует с бизнес-моделью и тестовыми данными



В этом варианте фрейморвка больше сложности, больше точек failures, но это ложится в концепцию Agile

Суммарно:



**Создание своего фреймворка**

Класс UserTests – содержит тесты, проверку Assert. Наследует класс CommonConditions

Класс CommonConditions содержит вещи, которые являются общими для всех тестов – здесь поля, которые мы будем переиспользовать, это константы путей, username, password, это инстанс драйвера. Метод SetUp(), поднимающий хром и разворачивающий его на полный экран, а еще TearDown(), закрывающий браузер в любом случае вне зависимости от результата тестов

Метод генерации рандомных строк находится прямо в тесте

В Page находятся абстракции веб-страниц. Их несколько Page-ей. Они все наследуются от супер-класса – абстрактного класса AbstractPage, в нем находится конструктор для хранения драйвера и метод открытия страницы и константа таймаута для wait

Четкое разделение по пакетам – страницы, тесты. Общие вещи – в суперклассах

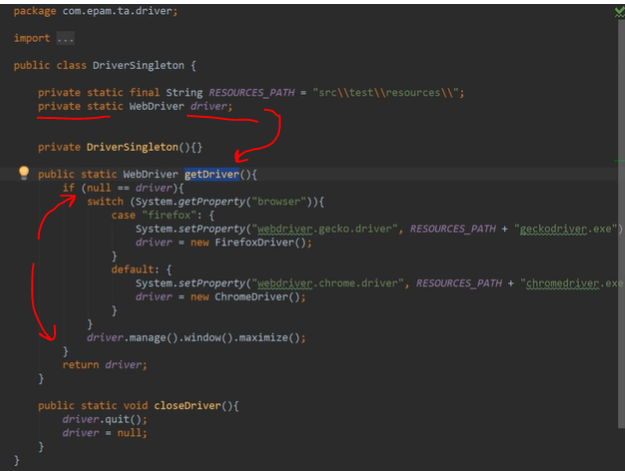
Все методы что-то возвращают в классах страниц, возвращают новые экземпляры страниц. Если остаемся на текущей странице (не переходим на новую, возвращаем this – текущий экземпляр)

Это нужно для цепочки вызова методов!!!!!!!!

**Библиотека hamcrest.Matchers для легких Assert**

Как было: Драйвер находится в суперклассе CommonConditions – там он инициализируется и создается, создаем new ChromeDriver()

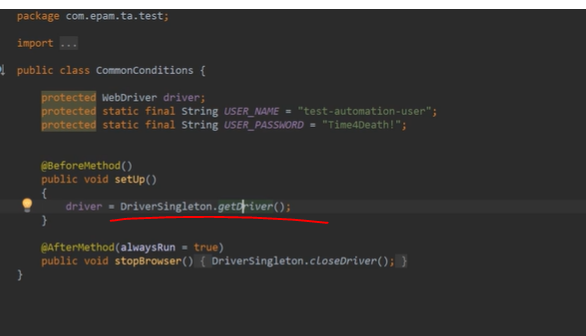
Как надо: На разных браузерах, перенесем создание и управление браузера в отдельный класс, класс DriverSingleton – это паттерн проектирования, нужен, чтобы мы имели 1 объект конкретного класса, не более, чем 1



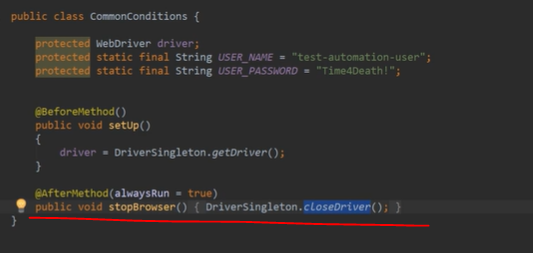
**Класс – набор статических переменных, статических методов. Переменная драйвера приватная и статическая. В методе getDriver() – если у нас он был, то есть не null, вернем переменную. Если не было – создаем, помещаем в переменную и возвращаем**

Здесь скрывается логика создания конкретного браузера

**Ссылаемся на DriverSingleton: дай драйвер**



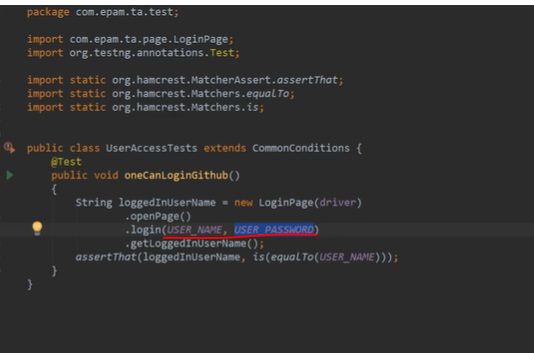
Когда не нужен – обращаемся к DriverSingleton и закрываем драйвер:

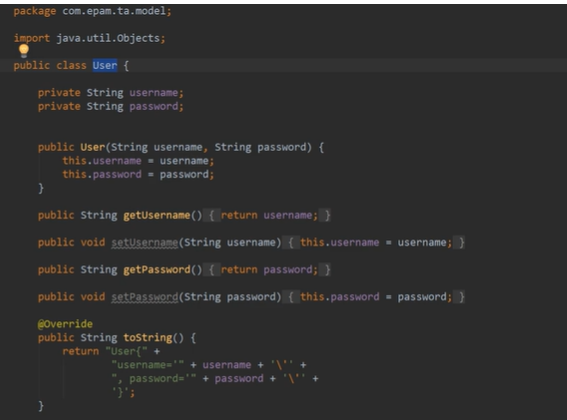


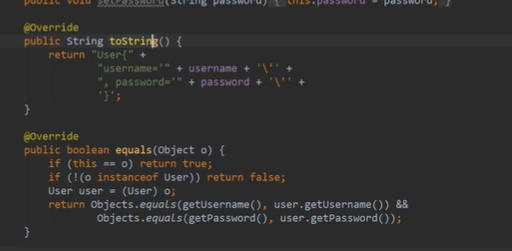
**WebDriver Manager – библиотека**

**Model**

Комбинация username + password = абстракция User

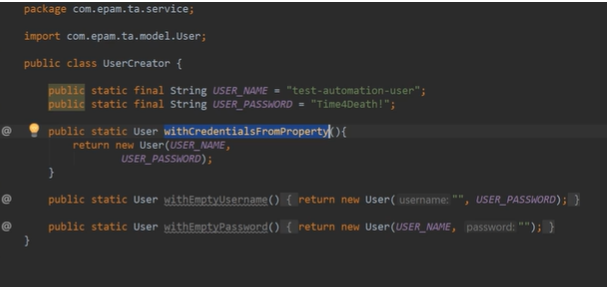


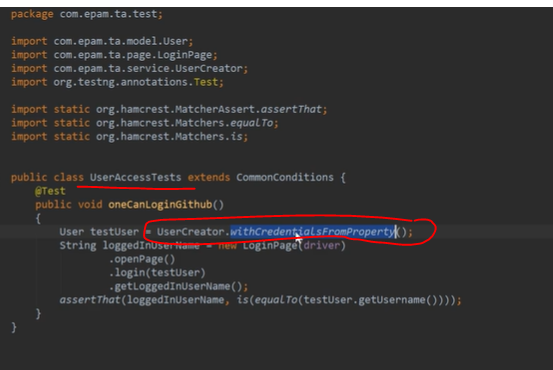




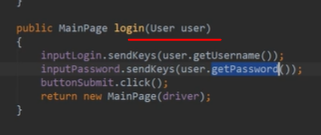


Класс User + класс UserCreator, статические методы в нем (создание пользователя с заданными значениями)





Передаем не пароль и логин, а пользователя:



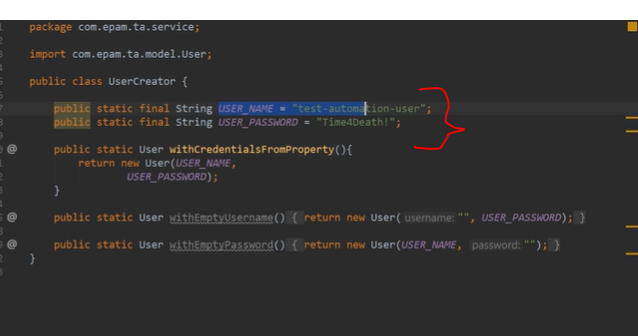
**Utils – утилиты**

Есть методы, которые нужны для теста, но они расположены в классе, где и тесты – нужно место, чтобы туда переехали вещи, которые мы не знаем, куда поместить

Класс \_\_\_\_Utils – статический, это какие-то операции по работе со строками/регулярными выражениями. Эти вещи вне бизнес-модели, это такой пакет. Но стоит быть осторожным и не превратить в свалку – подбирайте вещи для помещения тщательно!

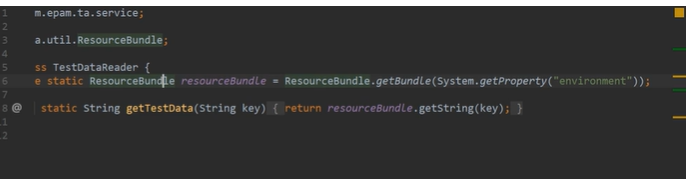
**Config & Test Data**

Пароль и имя пользователя жестко заданы:

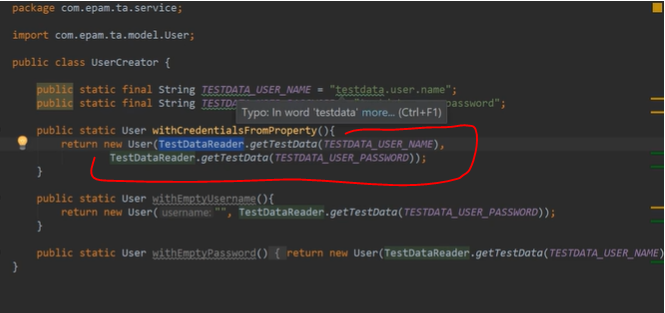


На реальных проектах – несколько окружений, несколько инфраструктурных решений для разных задач. На каждом инфраструктурном уровне – своя настройка тестовых данных, не всегда хочется пересобирать и переписывать константы, а, например, тесты надо запускать ежедневно во всех окружениях

Проще вынести в какой-то конфигурационный файл «ключ – значение» – и в конечном итоге управлять параметрами через командную строку. Для каждого окружения свой конфигурационный файл + метод доступа к этим данным



Вытягиваем из конфигурации:



**Reporting – модуль построения отчетов**

Репортинг - построение читаемого отчета о результатах выполнения тестов. В отчете прежде всего должна присутствовать информация о:

* Количестве запущенных тестов
* По каждому тесту:
* его название
* название сьюта
* результат
* время выполнения
* сообщение об ошибке, если тест упал
* Количество успешно выполненных тестов
* Количество упавших тестов
* Время выполнения всех тестов
* Дата и время запуска

TestListener

