1. Каждый автотест заливается в отдельную git branch
2. Для каждого автотеста создается свой Page Object (если автотесты используют разные страницы, например главная страница может быть использована в куче тестов и по этому создается только 1 Page Object, а вот страница логина уже будет другой Page)
3. Селекторы (локаторы) выносятся в отдельный файл

**Code Style: базовые принципы**

**Имена переменных и функций**

Одним из самых важных аспектов читаемого кода является именование: будь то объявление переменных, описание функций, названия классов и так далее. Очень важно, чтобы все имена, которые вы присваивали сущностям, были осмысленными и отражали реальную суть этого объекта. Избегайте однобуквенных и бессмысленных названий типа var1, x, y, my\_function, class2 и так далее. Идеальный код — самодокументируемый, к которому не нужны дополнительные пояснения. Если вы чувствуете, что вам хочется написать поясняющий комментарий, это повод переписать код так, чтобы комментарий не понадобился.

Обычно внутри каждой компании есть дополнительные внутренние соглашения о том, как именовать переменные, но общие правила в индустрии примерно одинаковые.

Функции пишутся через\_нижнее\_подчеркивание:

def test\_guest\_can\_see\_lesson\_name\_in\_lesson\_without\_course(self, driver):

Классы пишут с помощью CamelCase:

class TestLessonNameWithoutCourseForGuest():

Константы пишут в стиле UPPERCASE:

MAIN\_PAGE = "/catalog"

**Максимальная простота кода**

Здесь нам на помощь приходят известные принципы написания кода [DRY](https://en.wikipedia.org/wiki/Don't_repeat_yourself) (Don't repeat yourself) и [KISS](https://en.wikipedia.org/wiki/KISS_principle) (Keep it simple, stupid).

* Пишите максимально простой код везде, где это возможно.
* Не используйте переусложненных конструкций без большой необходимости (поменьше лямбда-выражений, map и разной другой магии). Если кусок кода можно заменить конструкцией более простой для понимания — замените.
* Пишите максимально линейный код, где это возможно, это проще для восприятия.
* Избегайте большой вложенности блоков кода, такие конструкции тяжело читать.
* Если можно вынести повторяющуюся логику куда-то, выносите, не повторяйтесь.
* По возможности пишите явный код вместо неявного. Чем меньше магии "под капотом", тем лучше.

**Code Style в автотестах**

Здесь мы попытались собрать важные принципы написания автотестов:

* Стремитесь к максимальной линейности кода тестов: избегайте ветвления и циклов в тест-кейсах. Если хочется добавить в тесте if, значит, нужно разбить этот тест на два теста или подготовить тестовое окружение так, чтобы не было необходимости использовать ветвление.
* Избегайте зависимых тестов, которые нужно запускать строго в определенном порядке. При росте количества автотестов вы захотите запускать их в несколько потоков параллельно, что будет невозможно при наличии зависимых тестов. А еще зависимые тесты очень не надежны. Подробнее: <http://barancev.github.io/test-deps-are-evil/>
* Стремитесь к тому, чтобы тест не полагался на контент, а готовил данные самостоятельно (и удалял после завершения). Используйте чистые браузеры и новых пользователей для лучшей воспроизводимости.
* Абсолютная линейность проверок. Когда вы пишете assert-утверждения в функциях, не следует использовать ветвления и циклы. Логика проверок должна быть линейна, иначе разбор багов и починка автотестов будут стоить очень дорого.
* Именуйте проверки в одинаковом стиле, чтобы по первому взгляду можно было понять, что это именно проверка. Например, можно именовать функции по шаблону should\_be\_smth:

def should\_be\_reply\_comment()

* Тесты именуются в одинаковом стиле. Имена тестов должны хорошо отражать различия в похожих сценариях. Можно использовать те же подходы, что и при добавлении имен к тест-кейсам в тестовой документации. Например, test\_guest\_can\_see\_teach\_button() — обратите внимание на явное указание на роль пользователя.
* Одинаковые тесты, которые отличаются только каким-то контентом (например, языком интерфейса), следует не копировать, а параметризовать.
* Пишите максимально атомарные и неделимые тесты. Не нужно писать один мега-тест, который проверяет вообще всё, напишите лучше десяток маленьких — проще будет локализовать проблему, когда она возникнет.

**Что такое Page Object Model?**

**Page Object Model** или кратко Page Object — это паттерн программирования, который очень популярен в автоматизации тестирования и является одним из стандартов при автоматизации тестирования веб-продуктов. Это также один из удобных способов структурировать свой код таким образом, чтобы его было удобно поддерживать, менять и работать с ним.

Основная идея состоит в том, что каждую страницу веб-приложения можно описать в виде объекта класса. Способы взаимодействия пользователя со страницей можно описать с помощью методов класса. В идеале тест, который будет использовать Page Object, должен описывать бизнес-логику тестового сценария и скрывать Selenium-методы взаимодействия с браузером и страницей. При изменениях в верстке страницы не придется исправлять тесты, связанные с этой страницей. Вместо этого нужно будет поправить только класс, описывающий страницу.

То есть здесь применяются те же принципы, что и в разработке: мы хотим повысить читаемость кода и вынести в абстрактные методы все детали. Тесты должны быть просто и понятно написаны, а повторяющиеся куски кода выделены в отдельные функции. В Page Object мы отделяем логику действий, например, авторизовать пользователя, от конкретной реализации (найти поле почты, ввести туда данные, найти поле пароля, ввести туда данные, найти кнопку и т.д.).

Рассмотрим такой простой тест-кейс:

1. Открыть главную страницу
2. Перейти на страницу логина

Ожидаемый результат:

Открыта страница логина

Давайте посмотрим на кусочек кода теста из предыдущего модуля, который реализует первую часть этого теста:

*test\_main\_page.py:*

link = "http://selenium1py.pythonanywhere.com/"

def test\_guest\_can\_go\_to\_login\_page(browser):

browser.get(link)

login\_link = browser.find\_element(By.CSS\_SELECTOR, "#login\_link")

login\_link.click()

Что здесь происходит?

Мы открываем ссылку, находим элемент с определенным селектором и нажимаем на этот элемент.

Что мы на самом деле имеем в виду?

Мы хотим открыть страницу логина. Давайте выделим это действие в отдельную функцию с понятным названием, пока все в том же файле *test\_main\_page*.py :

def go\_to\_login\_page(browser):

login\_link = browser.find\_element(By.CSS\_SELECTOR, "#login\_link")

login\_link.click()

и наш тест упрощается:

def test\_guest\_can\_go\_to\_login\_page(browser):

browser.get(link)

go\_to\_login\_page(browser)

При написании следующих тестов, когда нам понадобится перейти к странице логина с главной страницы, нам не нужно будет копировать этот кусочек кода или писать заново — мы сможем переиспользовать уже написанный метод.

**Почему именно Page Object?**

Можно, конечно, хранить всю логику наших тестов в одном файле под каким-нибудь условным названием steps.py, и для начала это уже неплохо. Но если мы тестируем большой веб-продукт с множеством разных состояний и переходов, этот файл может разрастись до огромных размеров, и найти в нем нужный метод будет непросто. Еще бывают ситуации, когда на разных страницах логически один и тот же метод имеет разную реализацию. Например, у нашего интернет-магазина может быть метод "добавить в корзину". Но пользователь может добавлять товар в корзину как со страницы каталога, так и со страницы самого товара.

Было бы удобно выделить все методы, которые логически относятся к одной веб-странице в нашем продукте, в отдельный класс в нашем коде. Отсюда и название Page Object — это абстрактный объект, который содержит в себе методы для работы с конкретной веб-страницей.

**Важно!**Обычно методы у Page Object бывают двух типов: *сделать что-то* и *проверить что-то.*

Рассмотрим страницу товара в интернет магазине <http://selenium1py.pythonanywhere.com/catalogue/the-shellcoders-handbook_209/>.

Какие могут быть методы у Page Object, ассоциированного с такой страницей? Запишем основные сценарии:

* добавить в корзину;
* проверить, что есть сообщение об успешном добавлении в корзину;
* перейти к написанию отзыва;
* проверить, что есть название, цена, описание товара;
* вернуться на главную.

Обратите внимание, что все проверки у нас тоже становятся отдельными методами. В самом тест-кейсе не остается никаких вспомогательных слов типа assert, только описание шагов. Прямо как в нашей тестовой документации.

Тесты будут выглядеть примерно так:

def test\_add\_to\_cart(browser):

page = ProductPage(url="", browser) # инициализируем объект Page Object

page.open() # открываем страницу в браузере

page.should\_be\_add\_to\_cart\_button() # проверяем что есть кнопка добавления в корзину

page.add\_product\_to\_cart() # жмем кнопку добавить в корзину

page.should\_be\_success\_message() # проверяем что есть сообщение с нужным текстом