ARTICLE HISTORY: July: 19, 2021 Accepted: September 12, 2021 Published: September 19, 2021

УДК 004.05

ТЕХНОЛОГИЯ АВТОМАТИЗАЦИИ ТЕСТИРОВАНИЯ ANDROID-ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ ПРОТОКОЛА WEBDRIVER НА ПЛАТФОРМЕ .NET

Ананьев Владимир Юрьевич
магистрант
Санкт-Петербургский государственный университет
телекоммуникаций им. проф. М.А. Бонч-Бруевича,

TEST AUTOMATION TECHNOLOGY OF ANDROID APPLICATIONS USING THE WEBDRIVER PROTOCOL ON THE .NET PLATFORM

Ananev Vladimir Iurievich
Master's student of
Federal State Budget-Financed Educational Institution of Higher Education
The Bonch-Bruevich Saint Petersburg State University of Telecommunications

Аннотация. В статье рассматриваются вопросы разработки автоматических тестов программного обеспечения с использованием протокола WebDriver. Проанализированы технологии WebDriver и Appium, которые позволяют реализовывать автоматизированные тесты Android-приложений на уровне графического интерфейса на платформе .NET. Исследованы шаблоны проектирования, наиболее полезные при разработке тестов графического интерфейса. Предложена архитектура проекта, позволяющая просто разрабатывать надёжные и поддерживаемые автотесты мобильных приложений на платформе .NET.

Abstract*:* The article deals with the development of automated tests utilizing the WebDriver Protocol. The WebDriver and Appium technologies are analyzed, that allow implementation of automated tests of Android applications at the level of the graphical interface on the .NET platform. The most useful patterns in the development of GUI tests are investigated. The project architecture is proposed, which allows you to simply develop reliable and supported automated tests of mobile applications on the .NET platform.

Ключевые слова: автоматизация тестирования, тестирование программного обеспечения, Android, Appium, .NET, NUnit, WebDriver.

Key words: test automation, software testing, Android, Appium, .NET, NUnit, WebDriver.

Введение. Верификация (или тестирование) программного обеспечения является важным, одним из ключевых этапов оценки качества программного продукта. Верификация представляет собой процесс анализа программного объекта для обнаружения различий между имеющимся и необходимым состоянием, и оценивания особенностей программного объекта. При этом тестирование программного обеспечения является деятельностью, которая сегодня должна производиться в течение всего процесса разработки.

Обзор предыдущих исследований. Тестирование программного продукта получило развитие, начиная с середины прошлого века. Процесс тестирования был предельно формализован, отделен от процесса непосредственной разработки ПО. Существовала концепция так называемого «исчерпывающего тестирования» – проверки всевозможных путей выполнения кода со всеми возможными входными данными. Однако выяснилось, что исчерпывающее тестирование на практике зачастую нереализуемо, так как количество возможных путей и входных данных очень велико. Стали исследоваться вопросы оптимизации тестирования. Процессинг тестирования обрел свою теорию. Говоря о родоначальниках тестирования, нужно сказать о книге «Искусство тестирования программ» Гленфорда Майерса [1]. Достойным источником информации о процессах тестирования являются разработки Рекса Блэка в книге «Ключевые процессы тестирования» [2]. Информативным представляется исследование о практике тестирования С. Куликова [3].

В начале 21 века развитие тестирования продолжалось в контексте поиска все новых и новых путей, методологий, техник и подходов к обеспечению качества [4, с. 5–10]. Серьезное влияние на понимание тестирования оказало появление гибких методологий разработки и таких подходов, как «разработка под управлением тестирования».

В настоящее время для упрощения процесса тестирования и сокращения временных затрат на этапе контроля качества в процессе разработки программного обеспечения стали часто применять автоматизацию. Автоматизация тестирования сегодня воспринимается как неотъемлемая часть большинства проектов с целью сокращения затрат путем использования программных средств для выполнения тестов и проверки результатов их выполнения.

Целью исследования является анализ технологий и подходов к разработке проекта автоматизации тестирования Android-приложения на уровне графического интерфейса с использованием протокола WebDriver на платформе .NET.

Основная часть.

Основными целями тестирования можно назвать следующие:

- проверка соответствия требований к продукту и текущего состояния продукта;
- выявление ситуаций, в которых поведение программы является неправильным, нежелательным или не соответствующим спецификации.

Для упрощения процесса тестирования и сокращения временных затрат на этапе контроля качества в процессе разработки программного обеспечения часто применяют автоматизированное тестирование.

Средства автоматизации позволяют:

- увеличить скорость выполнения тест-кейсов;
- снизить влияние человеческого фактора в процессе их выполнения;
- минимизировать затраты при многократном выполнении тест-кейсов;
- выполнять тест-кейсы, непосильные человеку ввиду сложности, скорости и других причин [4, с. 5–10].

Протокол WebDriver

Selenium был первоначально разработан Джейсоном Хаггинсом в 2004 году в качестве внутреннего инструмента в ThoughtWorks. Позже к Хаггинсу присоединились другие программисты и тестировщики в ThoughtWorks. В том же году Selenium был опубликован как ПО с открытым исходным кодом.

В 2007 году Хаггинс присоединился к Google. Вместе с другими разработчиками, в частности, с Дженнифер Беван, продолжил разработку Selenium. В то же время Саймон Стюарт из ThoughtWorks разработал инструмент автоматизации браузера под названием WebDriver. В 2009 году, после встречи разработчиков на конференции Google по автоматизации тестирования, было решено объединить два проекта и назвать новый проект Selenium WebDriver, или Selenium 2.0. С течением времени Selenium стал основным инструментом управления веб-браузерами.

В 2012 года Саймон Стюарт (изобретатель WebDriver), который тогда работал в Google, и Дэвид Бернс из Mozilla вели переговоры с W3C о том, чтобы принять WebDriver в качестве стандарта. В июле 2012 года был выпущена рабочая версия документа, а стандарт был принят июне 2018 года.

WebDriver – это стандарт W3C, описывающий API для автоматизации браузеров, и инструмент, который реализует данный стандарт. Для эмуляции действий пользователя при работе с мобильными приложениями существует много инструментов автоматизации, однако есть только один, реализующий стандарт W3C WebDriver [5, с. 36–38].

Appium как реализация стандарта WebDriver для управления мобильными устройствами

Appium – это инструмент для автоматизации тестирования нативных, мобильных, веб-приложений для iOS, Android и Windows платформ. Важным его преимуществом является кросс-платформенность: он позволяет писать тесты на разные платформы, используя один и тот же API, что дает возможность повторно использовать один и тот же код.

Appium представляет собой многокомпонентный инструмент, реализующий протокол WebDriver, принятый консорциумом W3C в качестве стандарта удалённого управления интерфейсами. Таким образом, Appium, наследуя большое сообщество разработчиков, предоставляет инструмент управления мобильными устройствами, принципы работы которого известны большому числу разработчиков.

Идеологи Appium исходили из следующих принципов:

1) разработчики не должны пересобирать или модифицировать приложения для автоматизации его тестирования;
2) разработчики не должны быть скованы одним языком или платформой для развертывания тестов.

Appium имеет клиент-серверную архитектуру. Сервер получает команды, которые необходимо исполнить и возвращает результат выполнения.

Клиент-серверная архитектура позволяет разрабатывать сценарии для тестов на любом языке программирования, который поддерживает протокол HTTP. При выполнении команд Appium-сервер переводит стандартные команды в вызовы методов ПО управления устройством. Для Android используется UI Automator, для iOS – XCUITest. Таким образом Appium инкапсулирует различные технологии управления различными устройствами в единый интерфейс, предусмотренный стандартом.

Данный инструмент позволяет настраивать сценарии, шагом в котором является нажатие на экран или ввод текста в поле так, как будто это сделал человек. Это максимально близкий к реальному пользователю способ. Однако из этого следует главный недостаток данного способа автоматизации тестирования – большая продолжительность каждого шага и потенциальная нестабильность тестов.

Выполнение команд всегда происходит в контексте сеанса. Клиенты инициируют сеанс с сервером, указывая набор так называемых «желаемых свойств» (Desired capabilities). Набор «желаемых свойств» сервера содержит в себе указания, управление, каким устройством требуется клиенту и каким приложением необходимо управлять. В случае, если сервер не может обеспечить сессию с требуемыми свойствами, запрос создания сессии завершится с ошибкой.

Appium поддерживает работу как с эмуляторами, так и с реальными устройствами. Для упрощения работы и минимизации затрат мы будем использовать эмулятор Android устройств от Google, который поставляется вместе с пакетом разработки Android-приложений Android Studio.

После установки Android Studio и требуемого Android SDK необходимо создать эмулятор требуемой версии и запустить его при помощи команды "emulator -avd <MyAvdName>".

Убедиться в том, что эмулятор или устройство доступно для Appium можно, запустив команду "adb devices".

Паттерны проектирования, специфичные для автоматизированного тестирования пользовательских интерфейсов

Page Object

Так как одно действие пользователя может требовать нескольких операций с WebDriver, предлагается отделить действия пользователя от деталей реализации данных действий. Таким образом минимизируются изменения в логике самих тестов, необходимые для их адаптации при изменении интерфейса. Для реализации этой абстракции использован шаблон проектирования, известный, как Page Object. Он состоит в том, что каждой странице веб-сайта или приложения в соответствие ставится класс, а доступная на странице функциональность описывается методами этого класса. Это позволяет исключить дублирование кода и описывать в тестовых сценариях логику взаимодействия в терминах высокоуровневого функционального описания системы. Код, использующий библиотеки для управления браузером, скрыт внутри классов – наследников описания страницы. Исследования показывают, что применение данного шаблона упрощает поддержку тестовых скриптов [5, с. 36–38].

При создании экземпляра класса, инкапсулирующего какую-либо страницу, может возникнуть несоответствие реального состояния приложения и его модели. Для того, чтобы этого избежать, создадим абстрактный класс ScreenBase, от которого наследуются остальные классы Page Object.

Код класса ScreenBase:

```
public abstract class ScreenBase
{
 public abstract AndroidElement pageLoadedVerifierElement { get; }

 protected DriverContainerData driverContainerData;
 protected AndroidDriver<AndroidElement> ad;
 protected TimeSpan pageLoadWaitTimeSpan = TimeSpan.FromSeconds(3);

 public ScreenBase() { }

 protected ScreenBase(AndroidDriver<AndroidElement> ad, DriverContainerData driverContainerData,
TimeSpan? pageLoadWaitTimeSpan = null,
 bool assertIsOpenAfterWait = false,
 bool doScreenShotAfterPageLoadWait = true)
 {
 this.ad = ad;
 this.driverContainerData = driverContainerData;
 if (pageLoadWaitTimeSpan.HasValue)
 {
 this.pageLoadWaitTimeSpan = pageLoadWaitTimeSpan.Value;
 }
 WaitUntilIsOpen();
 }

 private void WaitUntilIsOpen(bool assertIsOpenAfterWait)
 {
 WaitUntilIsOpen(pageLoadWaitTimeSpan);
 if (assertIsOpenAfterWait)
 {
 Assert.IsTrue(IsOpen(), $"Не удалось обнаружить окно {GetType().Name}");
 }
```

```
}
protected void WaitUntilIsOpen(TimeSpan PageLoadWaitTimeSpan)
{
WebDriverWait wait = new WebDriverWait(ad, PageLoadWaitTimeSpan);
wait.IgnoreExceptionTypes(typeof(WebDriverTimeoutException), typeof(TimeoutException));
try
{
_ = wait.Until(_ => IsOpen());
}
catch (WebDriverTimeoutException) { }
}
public virtual bool IsOpen()
{
try
{
return pageLoadedVerifierElement.Enabled;
}
catch (Exception)
{
return false;
}
}
}
```

При инициализации экземпляра наследника класса в конструктор-родитель передается информация о драйвере, информация об управляемом устройстве и время ожидания появления специального элемента, наличие которого на странице подтверждает, что данная страница этого класса отображена корректно.

Chain of Invocations

При разработке автотестов возникает проблема, когда разработчик не может быть уверен, можно ли выполнять то или иное действие в данный момент времени. Например, операция ввода пароля после входа в личный кабинет с корректными данными является некорректной. Для того, чтобы было очевидно, какой метод может быть вызван после вызова предыдущего, метод каждый раз должен возвращать экземпляр класса.

Код до применения шаблона:

```
[Test]
public void AuthorizationWithInvalidSMS()
{
RegistrationScreen registrationScreen = new RegistrationScreen(driver, cont.containerData);
LoginScreen loginScreen = registrationScreen.ClickSignInButton();
SmsScreen smsScreen = loginScreen.LoginAsValidUser(user.userData.username, user.userData.password);
ErrorPopupType1Screen errorPopup = smsScreen.InputIncorrectSmsCodeAndSubmit();
Assert.IsTrue(errorPopup.GetErrorText().Contains("Вы ввели неверный смс токен. У аВас осталось"));
}
```

Код после применения шаблона:

```
[Test]
public void AuthorizationWithInvalidSMS()
{
var errorPopup = new RegistrationScreen(driver, cont.containerData)
.ClickSignInButton()
.LoginAsValidUser(user.userData.username, user.userData.password)
.InputIncorrectSmsCodeAndSubmit();
Assert.IsTrue(errorPopup.GetErrorText().Contains("Вы ввели неверный смс токен. У Вас осталось"));
}
```

Применяя данный паттерн, разработчик не может совершить ошибку, когда действие с классом Page Object будет некорректно для текущего состояния приложения.

Управление состоянием AndroidDriver

Для управления Android-устройством в Appium под .NET предоставляется класс AndroidDriver<AndroidElement>. Создадим класс-обёртку, который управлял бы созданием драйвера, его уничтожением и требуемыми свойствами сервера.

```
public class DriverContainerData
{
```

```
public string deviceName { get; set; }
public Uri appiumUrl { get; set; }
public string app { get; set; }
public string appPackage { get; set; }
public string appActivity { get; set; }
}

public class DriverContainer
{
public AndroidDriver<AndroidElement> driver { get; private set; }

public readonly DriverContainerData containerData;
private readonly AppiumOptions options = new AppiumOptions();

public DriverContainer(DriverContainerData d)
{
this.containerData = d;

options.PlatformName = "Android";
options.AddAdditionalCapability(MobileCapabilityType.Udid, d.deviceName);
options.AddAdditionalCapability(MobileCapabilityType.App, d.app);
options.AddAdditionalCapability(AndroidMobileCapabilityType.AppPackage, d.appPackage);
options.AddAdditionalCapability(AndroidMobileCapabilityType.AppActivity, d.appActivity);
options.AddAdditionalCapability(MobileCapabilityType.NoReset, true);
options.AddAdditionalCapability(MobileCapabilityType.FullReset, false);
}

~DriverContainer()
{
KillDriver();
}

public void CreateDriver()
{
driver = new AndroidDriver<AndroidElement>(containerData.appiumUrl, options, TimeSpan.FromSeconds(300));
}

public void KillDriver()
{
if (driver != null)
{
try
{
driver.Quit();
}
catch { }
driver = null;
}
}
```

Фреймворк для тестирования NUnit

NUnit – это фреймворк модульного тестирования для всех языков .NET. Первоначально NUnit был портирован с языка Java (библиотека JUnit). Впоследствии NUnit был полностью переписан с добавлением новых функций и поддержкой широкого спектра платформ .NET.

NUnit использует атрибуты для идентификации тестов. Основными атрибутами являются [Test] – атрибут, указывающий непосредственно на тестовый метод, и атрибут [TestFixture], указывающий на класс, в котором хранится некоторое множество методов с атрибутом [Test], [SetUp] – атрибут, которым помечаются методы, выполняемые перед каждым тестом из Test Fixture, [TearDown]. Причем, методы, помеченные данным атрибутом, выполняются после каждого теста из Test Fixture.

Интересен механизм определения порядка вызова методов SetUp и TearDown при наследовании классов Test Fixture. А именно, если в базовом классе определён метод SetUp, то он будет вызван перед каждым тестовым методом в классе-наследнике. Если имеет место многоуровневое наследование, то будут вызваны все методы SetUp в порядке наследования классов. Методы TearDown вызываются в обратном порядке наследования [6, с. 23–27].

Иерархия классов типа Test Fixture

Благодаря особенностям реализации порядка вызова методов SetUp и TearDown в NUnit, мы имеем возможность построить приведение тестируемого приложения в определённое состояние к началу теста.

Для того, чтобы в каждом Test Fixture был Android Driver без необходимости дополнительных действий для работы с ним, создадим абстрактный класс TestFixtureBase:

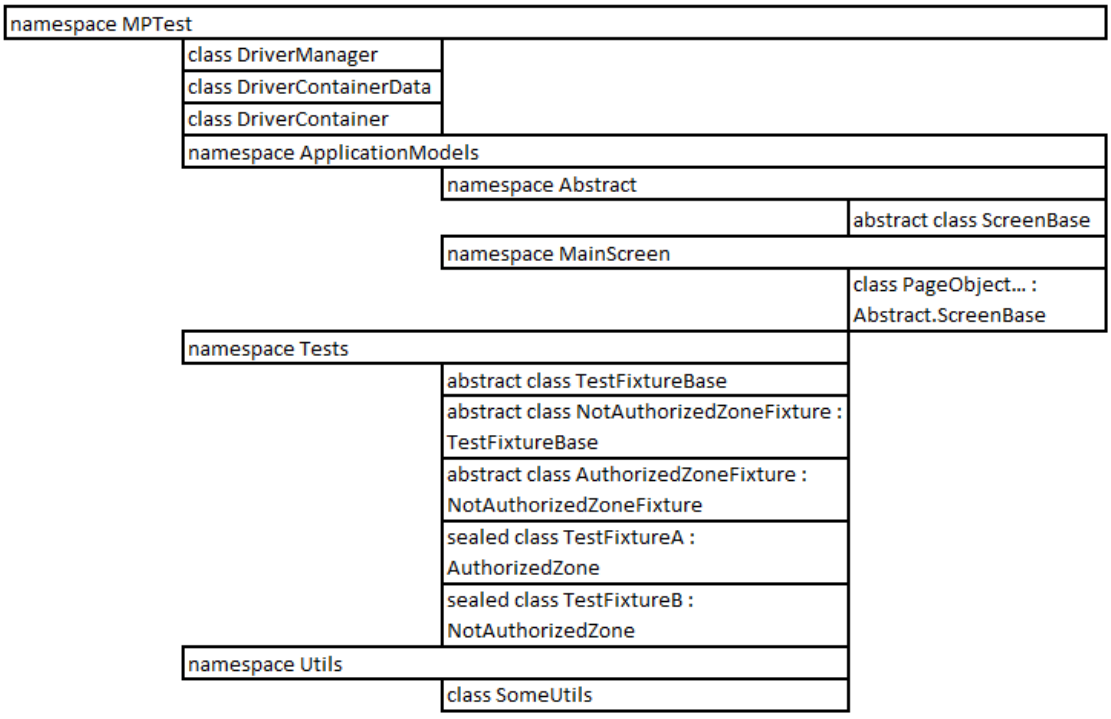
```
public abstract class TestFixtureBase
{
        protected DriverContainer cont;
        protected AndroidDriver<AndroidElement> driver;

        [SetUp]
        public void GetDriver()
        {
         cont = DriverManager.Instance.GetDriverContainer();
         driver = cont.driver;
        }
}
```

Метод GetDriver() осуществляет получение DriverContainer из управляющего класса DriverManager в начале каждого теста.

Классы Test Fixture с самими тестами дальше наследуются от данного базового. Благодаря порядку вызова методов SetUp и TearDown разработчик может быть уверен, что к началу каждого теста приложение будет находиться в определённом состоянии, которое предусмотрено конкретным классом с атрибутом Test Fixture.

Так, в иерархии классов TestFixture можно выделить Test Fixture, который приводил бы приложение к началу теста в состояние неавторизованной зоны, SetUp-метод которой не будет содержать никаких операторов, и Test Fixture авторизованной зоны, SetUp-метод которой будет содержать авторизацию в приложении.

С учётом описанного функционала и особенностей возможна следующая структура проекта:



При всем сказанном, как и для любого другого этапа разработки программного обеспечения, у этапа тестирования существуют характерные вопросы. Актуальность необходимости решения этих вопросов объясняется прежде всего, популярностью «гибких» (англ. agile) методологий разработки программного обеспечения.

Среди проблем автоматизированного тестирования можно выделить, прежде всего, высокие трудозатраты при первичном старте проекта автоматизации. Несмотря на то, что оно дает возможность уменьшить число рутинных операций и повысить скорость выполнения тестов, первичная разработка проекта и подготовка инфраструктуры для автоматизированного тестирования требует больших затрат.

Второй большой проблемой автоматизации тестирования является поддержка тестов в актуальном состоянии. Обновления тестов могут понадобиться как в случае изменения функционала, так и в случае изменения самих входных данных теста. Это характерно для обоих видов автоматизации. При реорганизации кода часто возникает необходимость обновить также юнит-тесты, а обновление кода собственно тестов, возможно, будет сравнимо по времени с изменением основного кода. Кроме того, в случае изменения интерфейса приложения возникает необходимость вновь переписать тесты, связанные с обновленными окнами, что при большом числе тестов может потребовать значительных ресурсов [6].

Также сложность представляет собой и сама процедура отбора тестов для автоматизации, поскольку далеко не все ситуации легко поддаются автоматизации.

Таким образом, автоматизация сценариев тестирования является дорогостоящим процессом. По этой причине автоматическое тестирование ценно на долгосрочных проектах с большим числом обновлений. На наш взгляд регрессионное тестирование является одним из лучших вариантов автоматизации тестирования.

Выводы:

1. Технология Appium позволяет реализовывать автоматизированные тесты мобильных приложений под Android с использованием протокола WebDriver.
2. Шаблоны проектирования Page Object и Chain of Invocations являются наиболее полезными при разработке автоматических тестов графического интерфейса.
3. Предложенная архитектура проекта, позволяет просто разрабатывать надёжные и поддерживаемые автотесты мобильных приложений на платформе .NET.

Перспективой развития данного исследования является исследование вопроса параллельного запуска тест-кейсов и Android-эмуляторов, а также интеграция разработанного приложения в процесс CI.

Список литературы:

1. Майерс Г, Баджетт Т., Сандлер К. Искусство тестирования программ. Изд-во Вильямс. 2016. 272 с.
2. Блэк Р. Ключевые процессы тестирования. Планирование, подготовка, проведение, совершенствование. Изд-во «Лори». 2011. 576 с.
3. Куликов С. С. Тестирование программного обеспечения. Базовый курс / С. С. Куликов. – 3-е изд. Минск: Четыре четверти. 2020. 312 с.
4. Артюхова А. Проблемы автоматизации тестирования и подходы к их решению // «CETERIS PARIBUS». №10. 2016. С. 5–10.
5. Воробьев Н. А., Бурмин Л.Н., Степанов Ю.А. Сравнительный анализ средств тестирования мобильных приложений // Евразийский Союз Ученых (ЕСУ). № 6(75). 2020. С. 36–38.
6. Курейчик В.М., Родзин С.И. Компьютерный синтез программных агентов и артефактов // Программные продукты и системы. 2004. № 1. С. 23–27.
7. Кент Бек. Экстремальное программирование: разработка через тестирование. Библиотека программиста. СПб.: Питер, 2003. 224 с.

References:

1. Myers G, Budgett T., Sandler K. The Art of Software Testing. Williams Publishing House. 2016. 272 p.
2. Black R. Key testing processes. Planning, preparation, implementation, improvement. Publishing house "Lori". 2011. 576 p.
3. Kulikov S. C. Software testing. Basic course / S. S. Kulikov. - 3rd ed. Minsk: Four quarters. 2020. 312 p.
4. Artyukhova A. Problems of test automation and approaches to their solution // "CETERIS PARIBUS". No. 10. 2016. Pp. 5–10.
5. Vorobiev N.A., Burmin L.N., Stepanov Yu.A. Comparative analysis of testing tools for mobile applications // Evrazijskij Soyuz Uchenyh (ESU). No. 6 (75). 2020.S. 36–38.
6. Kureichik V.M., Rodzin S.I. Computer synthesis of software agents and artifacts // Programmnye produkty i sistemy. 2004. No. 1. Pp. 23–27.
7. Kent Beck. Extreme Programming: Test Driven Development. Programmer's library. SPb .: Peter, 2003. 224 p.