Listeners & Extensions Part 1

על עקרונות SOLID שמעתן? SOLID אילו ראשי תיבות של 5 עקרונות חשובים כאשר כותבים Object Oriented Programming. אני ממליץ בחום לחפש על הנושא ברשת וללמוד אותו.
על החשיבות של קוד נקי, מאורגן ובנוי היטב קשה להפריז בדיבור. תשתיות שלא בנויות על קוד נקי ומאורגן, לרוב בסופו של דבר גורמות לתסריטי בדיקות אוטומטיים לא יציבים, דבר כשלעצמו גורם לאובדן אמון בתהליך הבדיקות האוטומטיות בכלל.

איך זה קשור לנושא?
כי באמצעות Listeners & Extensions ניתן לממש חלק מעקרונות ה- SOLID. אמנם לא נעסוק ב- SOLID עצמו במסגרת הפוסט הזה, אבל נזכיר שתי עקרונות חשובים מתוכו:
א. לכל יחידת קוד (class, method) צריך להיות **רק תפקיד אחד**.
ב. יחידות קוד צריכות להיות סגורות לשינויים מצד אחד (כדי להגביר יציבות ולמנוע תקלות), ולאפשר הרחבות, מצד שני (כדי לאפשר גמישות לשינויים והתפתחויות במהלך הקידוד).

Listeners & Extensions הם מנגנונים שנבנים על ידי כותבי frameworks, מנגנוני הרחבה שעושים בדיוק את זה - מאפשר לקוד התשתית להיות סגור לשינויים מצד אחד, ומצד שני פתוח להרחבות. כל תשתית טובה מאפשרת הרחבות. במקרה שלנו, Junit מאפשרת את זה, Selenium מאפשרת את זה וגם Appium מאפשרת את זה. כמובן שתשתיות נוספות הקיימות היום מאפשרות הרחבות, בצורה דומה - React, Angular, Spring, TestNG וכו'.
בפוסט הזה נדון בהרחבות של Selenium, ובפוסט הבא נדון בהרחבות של Junit.

הרחבה ב- Selenium היא מנגנון שנועד לאפשר לנו לבצע פעולות אוטומטיות, תוך כדי שתסריט האוטומציה שלנו רץ. נתבונן בדוגמא הבאה:

```
Log.info(“Going to click on the id”);
WebElement someElement = driver.findElement(By.id("someId"));
try{
    someElement.click();
    Log.info(“Just clicked on the id”);
```

```
    Report.info(“Just clicked on the id”);
} catch(Throwable T) {
   Log.error(“Could not find the id”);
   Report.error(“Could not find the id”);
   takeScreenshot(driver);
   Throw new Error(“quit the test”);
}
```

הדוגמא הזו מראה לנו תהליך בדיקה, שסה"כ יש בו 2 שורות רלבנטיות:

```
WebElement someElement = driver.findElement(By.id("someId"));
someElement.click();
```

וכל השאר זה רעש... רעש חשוב, אנחנו צריכים לכתוב לוגים וצריכים לכתוב דו"ח וצריכים לנהל שגיאות, אבל כל אלו משימות שאינן משימות בדיקה. אילו משימות צד. במקרה שלנו על 2 משימות ליבה - מציאת אלמנט ולחיצה עליו, **כתבנו 11 שורות קוד**.

עקרונית, היינו רוצים שהקוד שלנו יכיל רק את 2 השורות, אבל יחד עם זאת שכל הפונקציונליות שהגדרנו (כתיבה ללוג, כתיבה לדו"ח, שמירת תמונה בזמן שגיאה) תשמר. וכאן מגיע לידי ביטוי WebDriver Extension.

חשוב! לפני שנמשיך, חשוב לדעת שהמידע הטכני בפוסט הזה (שמות אובייקטים, מבנה מחלקות) מתייחס ל- Selenium 3.141.59 - גרסת Selenium 3 האחרונה הקיימת. נכון לזמן כתיבת שורות אלה, היא הגרסה היציבה ביותר, כאשר Selenium 4 נמצאת ב- Beta. בסוף הפוסט אתייחס במס' מילים גם ל- Selenium 4.

הכירו את אובייקט `EventFiringWebDriver`. אובייקט זה עוטף את `WebDriver` ובהתאם לשם שלו, כאשר מתרחשים אירועים מסויימים הוא "יורה" event ומפעיל מתודות. אילו מתודות? ואילו אירועים?
את האירועים האפשריים קבעו צוות הפיתוח של Selenium. את המתודות אנחנו נכתוב.
האירועים האפשריים להם נוכל לכתוב מתודות:

```
afterAlertAccetpt, beforeAlertAccetpt, afterAlertDismiss,
```

```
beforeAlertDismiss, afterClickOn, beforeClickOn, afterFIndBy, beforeFIndBy,
afterGetScreenshotAs, beforeGetScreenshotAs, afterGetText, beforeGetText,
afterNavigateBack, beforeNavigateBack, afterNavigateForward,
beforeNavigateForward, afterChangeValueOf, beforeChangeValueOf,
afterNavigationRefresh, berforeNavigationRefresh,  afterNavigateTo,
beforeNavigateTo, afterScript, beforeScript, afterSwitchWindow,
beforeSwitchWindow, onExcetption
```

כפי שניתן לראות באופן מיידי, כמעט לכל פעולה שאותה יכול אובייקט `WebDriver` לבצע, ניתן להגדיר מה יקרה לפני שהפעולה תתבצע ו/או אחריה. איך זה קורה? הנה דוגמא למחלקה שכותבת ללוג לפני מס' פעולות נבחרות:

```
public class LogListener implements WebDriverEventListener {

	public void beforeNavigateTo(String url, WebDriver driver) {
		Log.info("Before navigating to: '" + url + "'");
	}

	public void afterNavigateTo(String url, WebDriver driver) {
		Log.info("Navigated to:'" + url + "'");
	}

	public void beforeChangeValueOf(WebElement element, WebDriver driver) {
		Log.info("Value of the:" + element.toString()
				+ " before any changes made");
	}

	public void afterChangeValueOf(WebElement element, WebDriver driver) {
		Log.info("Element value changed to: " + element.toString());
	}

	public void beforeClickOn(WebElement element, WebDriver driver) {
		Log.info("Trying to click on: " + element.toString());
	}

	public void afterClickOn(WebElement element, WebDriver driver) {
		Log.info("Clicked on: " + element.toString());
	}

	public void beforeNavigateBack(WebDriver driver) {
```

```
            Log.info("Navigating back to previous page");
      }

      public void afterNavigateBack(WebDriver driver) {
            Log.info("Navigated back to previous page");
      }

      public void beforeNavigateForward(WebDriver driver) {
            Log.info("Navigating forward to next page");
      }

      public void afterNavigateForward(WebDriver driver) {
            Log.info("Navigated forward to next page");
      }

      public void onException(Throwable error, WebDriver driver) {
            Log.info("Exception occured: " + error);
      }

      public void beforeFindBy(By by, WebElement element, WebDriver driver) {
            Log.info("Trying to find Element By : " + by.toString());
      }

      public void afterFindBy(By by, WebElement element, WebDriver driver) {
            Log.info("Found Element By : " + by.toString());
      }

}
```

ועכשיו, נחבר את הקוד ל- WebDriver:

```
WebDriver originalDriver = new ChromeDriver();
EventFiringWebDriver driver = new EventFiringWebDriver(originalDriver);
LogListener listener = new LogListener();
driver.register(listener);

driver.get('http://www.login.com');
```

```
driver.get('http://www.google.com');
driver.navigate().refresh();
driver.navigate().back();
```

נסביר את הקוד:

שורה ראשונה - יצירת אובייקט `WebDriver`

שורה שניה - יצירת אובייקט `EventFiringWebDriver` על בסיס אובייקט ה- `WebDriver` שיצרנו

שורה שלישית- יצירת מופע מהטיפוס שהגדרנו קודם (`WebEventListener`)

שורה רביעית - רישום ה- listener באובייקט `EventFiringWebDriver`.

שורה חמישית ואילך - ביצוע פעולות אשר יירשמו באופן אוטומטי ללוג מבלי שנצטרך לרשום בעצמנו.

והתוצאה שתכתב ללוג תהיה:

```
Before navigating to: 'http://www.login.com'
Navigated to: `http://www.login.com`
Before navigating to: 'http://www.google.com'
Navigated to: `http://www.google.com`
Navigating back to previous page
Navigated back to previous page
```

אפשר להגדיר יותר מ- class אחד של Events. למען האמת, אפשר להגדיר אינסוף של מחלקות, שלכל אחת מהן ייעוד משלה. מחלקה אחת לדוגמא, תטפל בקריאה ללוגים, מחלקה אחרת תטפל בשגיאות, ומחלקה אחרת תכתוב לדו"ח. **כל מחלקה תעשה פעולה נפרדת, ועל ידי רישום של מס' מחלקות, מס' פעולות יתבצעו.**

לדוגמא, נניח ונבנה את המחלקות הבאות, בדומה לאופן שבו בנינו את `LogListener`:

`ReportListener, ErrorListener, LogListener`

ונרשום את כולם באופן הבא:

```
WebDriver originalDriver = new ChromeDriver();
EventFiringWebDriver driver = new EventFiringWebDriver(originalDriver);
```

```
LogListener listener = new LogListener();
ReportListener reportListeners = new ReportListener();
ErrorListener errorListener = new ErrorListener();

driver.register(listener);
driver.register(reportListeners);
driver.register(errorListener);
```

ואז, שתי הפעולות שהגדרנו בהתחלה:

```
WebElement someElement = driver.findElement(By.id("someId"));
someElement.click();
```

יבצעו את כל הפעולות שביקשנו מהן בקוד שממנו התחלנו - יכתבו ללוג, יכתבו לדו"ח, ובהנתן שגיאה, ישמרו צילום מסך.
את קטע הקוד שיוצר את האובייקט אפשר לשים ב- BeforeAll או במחלקה נפרדת.

Selenium 4

בסלניום 4 המנגנון שמתואר בפוסט זה כבר לא קיים. החליף אותו מנגנון `EventFiringDecorator` בשילוב עם מחלקה `.WebDriverListener` חשוב לציין שעקרון ההפעלה מאוד דומה לעקרון ההפעלה של Selenium 3, כפי שמתואר בפוסט הזה.

סיכום

איך כל זה קשור ל- SOLID? ומה אפשר ללמוד בצורה רחבה יותר?

על ידי יצירת מס' בלתי מוגבל של מחלקות, **שכל אחת מהן עוסקת בנושא אחד בלבד, ובהפרדה מוחלטת מאחרות**, נוכל לממש את מנגנון ההפרדה. כל קטע קוד יבצע רק דבר אחד. כאשר מערבבים מס' מטרות בקטע קוד אחד - נניח כתיבה ללוג וכתיבה לדו"ח, אז כאשר משתנה רק משהו אחד, נניח מנגנון הלוגים שלנו, נצטרך לשנות את קטע הקוד המשותף לשניהם, ואז נפגע כמעט בוודאות בקטע הקוד שמבצע כתיבה לדו"ח. רעיון ההפרדה נועד לאפשר לנו להמשיך ולהתפתח, לתקן ולהחליף חלקי קוד, מבלי לפגוע בקטעים אחרים, שאינם קשורים.

לכן מומלץ לכתוב מחלקות Listeners על פי נושאים, כפי שהודגם בפוסט הזה.
בצורה רחבה יותר, עקרון ההרחבה נכון לכל framework, והוא עובד בצורה דומה. לפעמים על ידי מימוש interface, לפעמים על ידי ירושה של מחלקה שממממשת את ה- listener, אבל כמעט תמיד באותו רעיון. לכן, בפעם הבאה שנראה לכן שעבור קטע קוד קטן צריך לכתוב מעטפת גדולה, והמהות הולכת לאיבוד, או שכותבים הרבה מאוד utilities, הדרך הנכונה היא לחפש את מנגנון ה- Extension הקרוב אליכן, ולרתום אותו לצרכים שלכן.

בפוסט הבא נדון במגנון דומה עבור תשתית ה- unit tests. אנחנו בחרנו ב- Junit 5 לצורך הדוגמאות, אבל העקרון זהה גם עבור junit 4 או TestNG.