Dynamic Proxy & Page Factory

על מנת להבין את הנושאים Dynamic Proxy ו- Page Factory המקום הטוב ביותר להתחיל בו הוא הגדרת הבעיה שנושאים אלה באו לפתור. מה יקרה בקטע הקוד הבא?

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

התשובה היא די ברורה במקרה הפשוט הזה- נמצא את האלמנט בעל הזיהוי `someId` על המסך (אנחנו מניחים שהוא קיים) ונלחץ עליו.
ומה יקרה במקרה הזה?

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

במקרה הזה תתקבל שגיאה. יותר נכון, `Exception`. וליתר דיוק `StaleElementException`. למה?
מכיוון ש- `WebElement` מייצג אובייקט ספציפי על דף ה- Web, או ליתר דיוק, על ה- DOM. ביצוע refresh של הדף יגרום לכל האובייקטים על ה- DOM להיהרס ולהיבנות מחדש. לכן, המופע של האובייקט שאותו אנו שומרים בזכרון ב- `WebELement` לא קיים יותר על הדף לאחר ה- refresh. לאחר ה- refresh אין שם את האובייקט אותו אנחנו מחפשים. על מנת ללחוץ על הכפתור, נצטרך לבצע `findElement` מחדש ורק אז ללחוץ על הכפתור.

המצב אפילו מורכב יותר כאשר מדובר ב- Page Objects. התבנית Page Objects נועדה לייצג דף אינטרנט, אשר השדות בו, ה- Fields מייצגים את השדות במסך והמתודות בו מייצגות את ההתנהגות הרצויה של הדף.
ניקח לדוגמא את דף ה-login:

```
public class LoginPage {

    private WebDriver driver;

    private WebElement userName;
    private WebElement password;
    private WebElement loginButton;

    public LoginPage(WebDriver driver){
       this.driver = driver;
       userName = driver.findElements(By.id('username'));
       password = driver.findElements(By.id('password'));
       loginButtone = driver.findElements(By.id('loginBtn'));
    }


    public void login(string userName, string password) {
        userName.sendKeys(userName);
        password.sendKeys(password);
        loginButton.click();
    }
}
```

אפשר לראות, שלושה `WebElements` בסיסיים - טקסט שם המשתמש, טקסט סיסמא וכפתור לוגאין.

האתחול שלהם מתרחש ב- constructor. מיד אפשר לראות שהבעיה הראשונה היא אתחול של המחלקה כאשר אנחנו לא נמצאים בדף הלוגאין. הקוד הבא לדוגמא יוביל לשגיאה:

```
driver.get('http://www.google.com');
LoginPage login = new LoginPage(driver);
```

למה? כי בעמוד google לא נמצא את האלמנטים שאנחנו מחפשים.

בעיה נוספת היא, שנניח ואתחלנו את המחלקה כשהיינו בעמוד הלוגאין, אך ביצענו refresh לדף מאיזושהי סיבה, תתקבל שגיאה. לדוגמא, הקוד הבא יוביל לשגיאה:

```
driver.get('http://www.login.com');
LoginPage login = new LoginPage(driver);
driver.navigate().refresh();
login.login('myUser', myPassword');
```

מה עושים? לפנינו מס' אפשרויות:

א. כל פעם ב- page object לפני שמשתמשים באלמנט, נבצע לו חיפוש. לדוגמא המחלקה תיראה בצורה הבאה:

```
public class LoginPage {

    private WebDriver driver;

    public LoginPage(WebDriver driver){
       this.driver = driver;
    }

    public void login(string userName, string password) {
        driver.findElements(By.id('username')).sendKeys(userName);
        driver.findElements(By.id('password')).password.sendKeys(password);
        driver.findElements(By.id('loginBtn')).loginButton.click();
    }
}
```

במימוש הזה, כל פעם שנקרא למתודה `login` נקבל אלמנט עדכני. הבעיה היא, שבצורת כתיבה זו גם נגרם שכפול קוד (כל פעם שאנחנו רוצים להשתמש באלמנט נצטרך לבצע לו חיפוש), וכמובן שהמחלקה `LoginPage` אינה מחלקה מייצגת. `PageObject` ממוצע הוא ארוך יותר ומשתמש בשדות שלו במס' מתודות שונות. כך לדוגמא אם משתנה הלוקייטור `'username'` ל-`'user'`(נניח שינוי של מתכנת באחת הגרסאות), נצטרך לחפש את כל המקומות בהם הגדרנו username ולהפוך אותם ל user.

אז אפשר להשתמש בפתרון ביניים - במקום שהשדות ייצגו את האלמנט, השדות ייצגו את אובייקט ה- `By` שמשמשים למציאת האלמנט ואז נחפש על פי ה- `By` כל פעם שאנחנו רוצים להשתמש. הקוד ייראה כך:

```
public class LoginPage {

    private WebDriver driver;
```

```
    private By userName;
    private By password;
    private By loginButton;

    public LoginPage(WebDriver driver){
       this.driver = driver;
       userName = By.id('username');
       password = By.id('password');
       loginButton = By.id('loginBtn');
    }


    public void login(string userName, string password) {
        driver.finElement(this.userName).sendKeys(userName);
        driver.finElement(this.password).sendKeys(password);
        driver.finElement(this.loginButton).click();
    }
}
```

זאת אופציה טובה יותר מאופציה ב', אך היא תדרוש כתיבה מסורבלת - שורות ארוכות וקוד שאינו קריא.

הפתרון האידיאלי שהיינו רוצים הוא פשוט להשתמש ב- WebElement ולהיות בטוחים שהוא עדכני. **אז איך עושים את זה?**

כותבי Selenium חשבו על הבעיה הזו והשתמשו בשיטות וכלים של java כדי לפתור את הבעיה. ליתר דיוק, הם השתמשו בתבנית עיצוב, design pattern, שנקראת dynamic proxy כדי לפתור את הבעיה.

מה זה proxy? התרגום המילולי של proxy הוא "בא כח". לדוגמא, אם אתם רוצים לקנות בית, ואין לכם מספיק ידע משפטי על מנת לבצע בצורה טובה את הרכישה, תשכרו עורך דין. הוא בא-הכח שלכם ומבצע פעולות בשמכם. עורך הדין הוא ה- proxy שלכם לצורך קניית הבית.

לשימוש ב- proxy תמיד יש סיבה. במקרה של עורך דין הסיבה היא לוודא שהעסקה תקינה ואנחנו מוגנים מבחינה משפטית.

דוגמא נוספת לשימוש ב proxy היא גישה לאינטרנט. כאשר משתמשים ב- proxy לצורך גישה לאינטרנט, כאשר אנחנו מבקשים מקום אחד שדרכו עוברת כל התעבורה מ ולאינטרנט. ישנן מס' סיבות למה להשתמש ב- proxy כזה- סינון תוכן לא ראוי, הגנה מפני תקיפות, או הקלטה של תעבורה נניח לצורך בדיקות עומסים. proxy הוא מקום אחד דרכו עוברת כל התעבורה, ולכן נפעיל עליו פעולה (כמו סינון תוכן או שמירת התעבורה בקובץ).

במקרה שלנו, dynamic proxy ישמש אותנו לצורך הבא - בכל פעם שמישהו ישתמש ב- `WebElement`, אזי באופן אוטומטי לא יינתן לו האובייקט הישן אלא מאחורי הקלעים יבוצע חיפוש חדש לאלמנט. וכך נהיה בטוחים שהאלמנט שקיבלנו הוא עדכני. אז בעצם dynamic proxy עוטף את `WebElement` ומשמש חוצץ ביננו ובינו על

מנת לממש את הלוגיקה הזו. כל פעם שנקרא ל- `WebElement`, בעצם נקרא ל- proxy במקומו, כאשר תפקידו של ה- proxy יהיה אחד - לבצע עבורנו `findElement` ולהביא לנו עותק מעודכן של האלמנט מה- DOM

איך מסמנים dynamic proxy? ואיך נפעיל אותו? נסמן כל שדה באופן הבא:

```
@FindBy(id='username')
WebElement userName;
```

ואז נצטרך לאתחל את האלמנטים על מנת שה- dynamic proxy ייכנס לפעולה. את זה נבצע על ידי שימוש במנגנון שנקרא PageFactory, מנגנון שנועד לאתחל את ה- proxy-ים השונים. המחלקה תיראה כך:

```
public class LoginPage {

    private WebDriver driver;

    @FindBy(id=”username”);
    private WebElement userName;

    @FindBy(id=”password”);
    private WebElement password;

    @FindBy(id=”loginBtn”);
    private WebElement loginButton;

    public LoginPage(WebDriver driver){
       this.driver = driver;
       PageFactory.initElements(driver, this);
    }


    public void login(string userName, string password) {
        userName.sendKeys(userName);
        password.sendKeys(password);
        loginButton.click();
    }
}
```

שימו לב לשורה:

```
PageFactory.initElements(driver, this);
```

ב- constructor שמאתחלת את האלמנטים, ובעצם מייצרת dynamic proxy לכל `WebElement`

מלה לכותבי CShrap:

ספריות ה- Selenium , לאחר שנכתבות ב- Java מתורגמות להרבה שפות - Python, C#, Ruby ועוד. הרעיון הוא לקחת את הגרסה היציבה ב- Java ולתרגם אותם לשפות נוספות. כמובן שמימוש בשפה שאינה Java דורש התאמה בהתאם לכללי השפה אליה מבוצע התרגום.

ב- 2018, בגרסת Selenium 3.11.0 קרה משהו בספריות Selenium עבור - Csharp. הופסקה התמיכה ב- PageFactory. ב- release notes נכתב - המנגנון לא יציב ומלא בתקלות, ובכל מקרה אינו יעיל ב- C# ולכן הוחלט להפסיק בו את התמיכה. הרבה ביקורת נוצרה סביב הנושא. כש - Jim Evans, המתחזק של קוד C# עבור Selenium הסביר את ההחלטה: (אפשר לקרוא על כך בלינק הזה):

בניגוד ל- java, ל- chsarp יש מנגנון שנקרא Property. Property בעצם מסוגל להחליף getters ו setters קלאסיים של שפות כמו java ו- cpp.

המוטיבציה לכתוב property דומה למוטיבציה לכתוב dynamic proxy - אנחנו רוצים מצד אחד לפשט את הגישה ולגשת ישר לאובייקט ולא להדרש להפעיל מתודות סביבו- זו כתיבה נכונה וקריאה יותר, ומצד שני לא רוצים לאפשר גישה ישירה לאובייקטים ללא ניהול ושמירה על כללי Object Oriented והלוגיקה הפנימית של המחלקה שלנו.

כלומר, אם נשתמש ב- property נוכל להגיע בדיוק לאותו מצב כמו dynamic proxy גם בלי להשתמש במנגנון. נניח בצורה הזו:

ואז אין באמת צורך ב- PageFactory ב- Csharp כי הגענו לאותן תוצאות גם בלעדיו. הקוד ייראה כך:

```
public class LoginPage {

    public WebElement UserName{
       get{
        return driver.findElement(By.id("username"));
       }
    }

    public WebElement Password{
       get{
        return driver.findElement(By.id("password"));
       }
    }

    public WebElement LoginButton{
       get{
        return driver.findElement(By.id("loginBtn"));
       }
    }


    public LoginPage(WebDriver driver){
       this.driver = driver;
    }


    public void login(string userName, string password) {
        UserName.sendKeys(userName);
        Password.sendKeys(password);
        LoginButton.click();
    }
}
```

חשוב לשים לב, שכאשר ב- Java לא מוצאים את האלמנט, המערכת לא מעיפה Exception אלא מכניסה לתוך האלמנט null. זה חשוב כי לפעמים נבחר לאתחל page object כאשר הדף עוד לא טעון, ועדיין נרצה שהאלמנטים בו יהיו עדכניים כשנרצה להשתמש בו, ולא נרצה ליפול. לכן, צורה נכונה יותר לכתוב היא זו:

```
public WebElement LoginButton{
   get{
     try{
        return driver.findElement(By.id("loginBtn"));
     } catch (Exception e){
        return null;
     }
  }
}
```

בהמשך של אותו קו מחשבה, אפשר במקום לבצע driver.findElement אפשר לכתוב wait שממתין לאלמנט ואז הסיכוי שיחזור לאחר שנמצא גבוה יותר.

סיכום

על מנת להגדיל ככל הניתן את יציבות הטסטים, תוך כדי שמירה על קוד מסודר ונקי, Selenium משתמש במנגנון `PageFactory` על מנת לאתחל את אובייקטי `WebElements`. מנגנון ה- `PageFactory` עוטף את אובייקטי `WebElement` ב- dynamic proxy וכך בכל פניה אליהם, מאחורי הקלעים מתבצע חיפוש - `findElement`.