Sesión S09: Pruebas de aceptación (2)

Pruebas de propiedades emergentes funcionales con Selenium WebDriver

Localización de elementos

Acciones sobre los elementos

Acciones de ratón y teclado

Navegación

Patrón de diseño page object

Vamos al laboratorio...

María Isabel Alfonso Galipienso
Universidad de Alicante eli@ua.es

Niveles de pruebas

VERIFICACIÓN

¿el producto está implementado correctamente?

El objetivo general (intención) es ENCONTRAR DEFECTOS

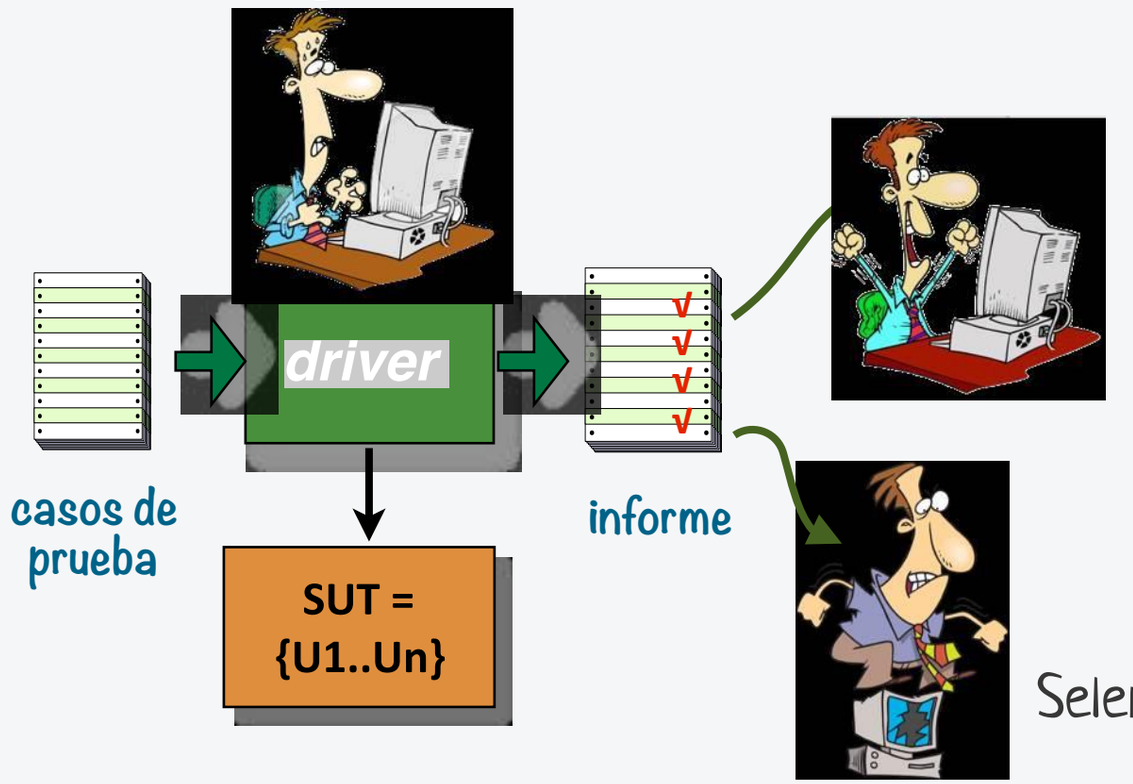
VALIDACIÓN

¿el producto es el correcto?

Las pruebas de aceptación determinan en qué grado el sistema satisface los CRITERIOS DE ACEPTACIÓN

- Business acceptance testing
- User acceptance testing (α, β)

Cada nivel tiene una granularidad y objetivos concretos diferentes. Hay un ORDEN temporal entre ellos.



Propiedades emergentes

- FUNCIONALES
  - Automatización de pruebas de aplicaciones Web
    - Selenium IDE
    - Selenium WebDriver
- NO funcionales

Sobre Selenium IDE

- Selenium IDE utiliza el patrón "Record and Playback Pattern"
  - La idea es permitir que el usuario "guarde" (record) las actividades de testing y las pueda ejecutar posteriormente (playback) utilizando la herramienta de pruebas
- Algunas ventajas de utilizar Selenium IDE son:
  - Podemos implementar tests más rápidamente, por lo que podemos crear grandes conjuntos de suites en horas en lugar de semanas
  - No se requiere ninguna experiencia previa con lenguajes de programación
  - La búsqueda de elementos en la página es muy fácil y rápida
- Algunos inconvenientes de utilizar Selenium IDE son:
  - Tests inflexibles debido a que la ejecución de los tests es idéntica a la grabación de los mismos (¿qué ocurre si necesitamos ejecutar cada test con un usuario diferente cada vez?)
  - Duplicación de código: no podemos reutilizar el código de los tests
  - No soporta la gestión de errores ni se pueden integrar los tests en el proceso de construcción del proyecto
- Las limitaciones indicadas se pueden superar utilizando algún lenguaje de programación. **WebDriver** nos permite utilizar varios lenguajes, entre ellos, java, para programar los tests de pruebas emergentes funcionales sobre aplicaciones web en diferentes navegadores

Características de Selenium WebDriver

- Proporciona un buen control del navegador a través de implementaciones específicas para cada uno de ellos
- Permite realizar una programación más flexible de los tests

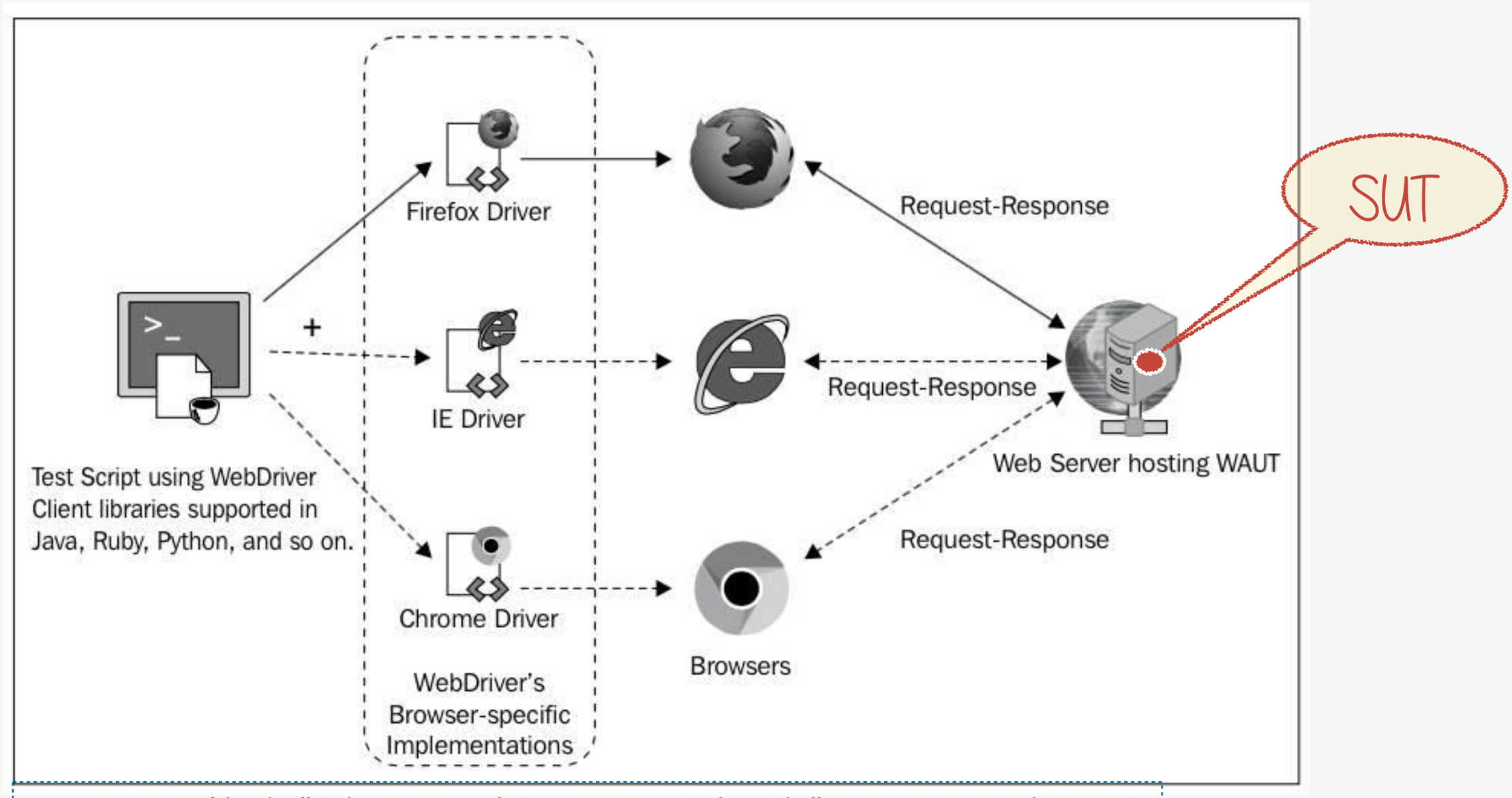
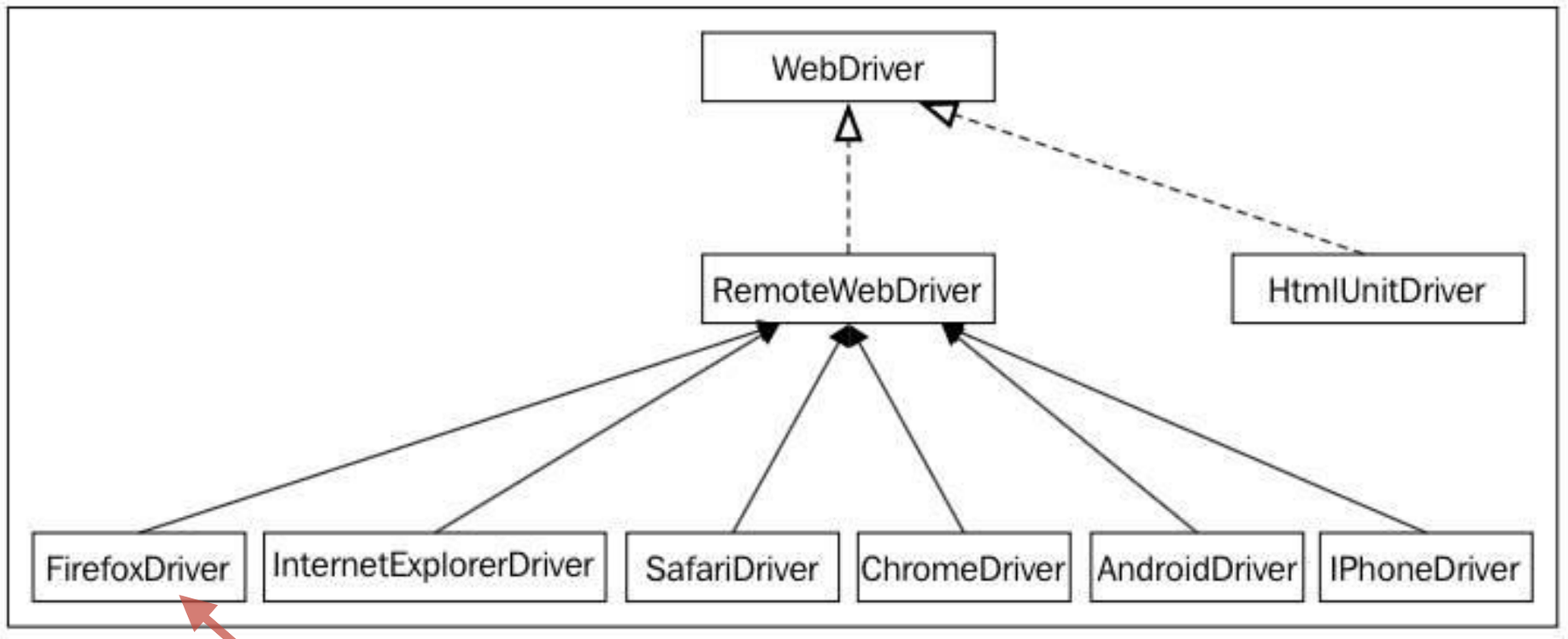


imagen extraída de "Selenium WebDriver practical guide". Satya Avasarala. 2014.

WAUT: **W**eb **A**pplication **U**nder **T**est

WebDriver interfaces y WebElements

- WebDriver es una interfaz cuya implementación concreta la realizan dos clases: RemoteWebDriver y HtmlUnitDriver




```
WebDriver driver = new FirefoxDriver();
driver.get("http://www.google.com");
WebElement searchBox = driver.findElement(By.name("q"));
searchBox.sendKeys("Packt Publishing");
searchBox.submit();
```

"driver" representa el navegador

abrimos una URL

localizamos un elemento en la página

interaccionamos con los elementos de la página

- Una página web está formada por elementos HTML, que son objetos de tipo **WebElement** en el contexto de WebDriver
- Una vez localizados los WebElements, podremos realizar acciones sobre ellos

Elementos HTML más usados

Los objetos WebElement representan elementos HTML

<select> <option ... → Drop Down

<input type="radio" ... → Radio Button

<input type="checkbox" ... → Check Box

<input type="text" ... → Text Box

<input type="password" ... → Password Box

<input type="button" ... → Button

Mecanismos de localización

- Antes de realizar alguna acción (entradas del usuario) con WebDriver, debemos LOCALIZAR el elemento que nos interese. Para ello utilizamos los métodos:

```
WebElement findElement(By by) throws NoSuchElementException
java.util.List<WebElement> findElements(By by)
```

  - Como parámetro de entrada se requiere una instancia de "By", que nos permite localizar elementos en una página web
  - El inspector de elementos de Firefox puede ayudarnos a localizar los elementos HTML de las páginas web cargadas por el navegador. Es tarea del desarrollador el elegir el "locator" adecuado para utilizarlo en findElement()

- Hay 8 formas de **localizar** un WebElement en una página web:
  - By.name(), By.id(), By.tagName(), By.className(), By.linkText(), By.partialLinkText(), By.xpath(), By.cssSelector()
  - Locator `css`: los "css selectors" son patrones de caracteres utilizados para identificar un elemento HTML basado en una combinación de etiquetas HTML, id, class, y otros atributos. Los formatos más comunes para los selectores css son:
    - css=tag#id → valor de etiqueta seguida de "#" y del valor del atributo id
    - css=tag.class → valor de etiqueta seguida "." y del valor del atributo class
    - css=tag.class[attribute=value]
    - css=tag[attribute=value]
    - css=tag:contains("inner text")

Ejemplos de locators CSS

- Suponemos que el código HTML de nuestra página web es:

```
1  <html>
2   <body>
3    <form id="loginForm">
4     <input class="required" name="username" type="text" />
5     <input class="required passfield" name="password" type="password" />
6     <input name="continue" type="submit" value="Login" />
7     <input name="continue" type="button" value="Clear" />
8    </form>
9  </body>
10 <html>
```

- css=form#loginForm (línea 3)
- css=input[name="username"] (4)
- css=input.passfield (5)
- css=input.required[type="text"] (4)

```
<td align="rigth">
  <font size="2" face="Arial, Helvetica, sans-serif">Password:</font>
</td>
```

- css=font:contains("Password:")

Ejemplo extraído de: https://www.seleniumhq.org/docs/02_selenium_ide.jsp#locating-by-css

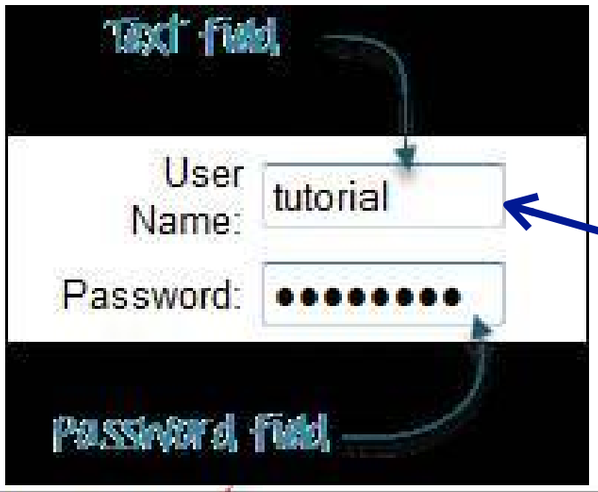
Acciones sobre los WebElements

- Una vez que hemos localizado el elemento que nos interesa, podemos ejecutar **ACCIONES** sobre ellos.
  - Cada tipo de elemento tiene asociado un conjunto diferente de posibles acciones. P.ej. sobre un elemento textbox, podemos introducir un texto o borrarlo
- Ejemplos de acciones:
  - sendKeys(secuencia de caracteres)
    - Se utiliza para introducir texto en elementos textbox o textarea
  - clear() se utiliza para borrar texto en elementos textbox o textarea
  - submit()
    - Puede aplicarse sobre un un elemento form, o sobre un elemento que esté dentro de un form. Envía el formulario de la página web al servidor en el que reside la aplicación web
- Ejemplos de acciones que pueden ejecutarse sobre **cualquier** WebElement:
  - getAttribute(), getLocation(), getText(), isDisplayed(), isEnabled(), isSelected()

Ejemplos de acciones

Imágenes extraídas de: http://www.guru99.com/accessing-forms-in-webdriver.html

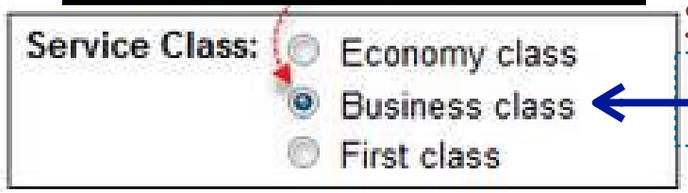
ver también: http://www.guru99.com/locators-in-selenium-ide.html



Introducir texto en un text box y password box

```
driver.findElement(By.name("username")).sendKeys("tutorial");
```

combinación de etiquetas html, id, class y otros atributos



seleccionar un radio box

```
driver.findElement(By.cssSelector("input[value='Business']")).click();
```

pulsar sobre un enlace de texto

```
driver.findElement(By.linkText("Register here")).click();
```

```
driver.findElement(By.partialLinkText("here")).click();
```

seleccionar elementos en un drop box

```
import org.openqa.selenium.support.ui.Select;
Select drpCountry =
    new Select(driver.findElement(By.name("country")));
drpCountry.selectByVisibleText("ANTARTICA");
```

Ejemplo de driver: acceso a Gmail

```
@Test
public void signIn() {
  WebDriver driver = new FirefoxDriver();
  String appUrl = "https://accounts.google.com";

  driver.get(appUrl);   //abrimos la página en el navegado
  driver.manage().window().maximize();

  String expectedTitle = "Inicio de sesión - Cuentas de Google";
  String actualTitle = driver.getTitle();
  Assert.assertEquals("Url incorrecta", expectedTitle,actualTitle);

  WebElement username = driver.findElement(By.id("Email"));
  username.clear();
  username.sendKeys("TestSelenium");  //tecleamos el usuario

  WebElement nextButton = driver.findElement(By.id("next"));
  nextButton.click();

  WebElement password = driver.findElement(By.id("Passwd"));
  password.clear();
  password.sendKeys("password123");  //tecleamos el password

  WebElement SignInButton = driver.findElement(By.id("signIn"));
  SignInButton.click();

  expectedTitle = "Inicio de sesión - Cuentas de Google";
  actualTitle = driver.getTitle();
  driver.close();  //cerramos el navegador
  Assert.assertEquals("Sign In Fallido", expectedTitle,actualTitle);
}
```

Tiempos de espera

- Podemos establecer tiempos de espera en nuestros tests para evitar errores debidos a que no se localiza un elemento en la página porque todavía se esté cargando (excepción `NoSuchElementException`)
  - Tiempo de espera **implícito**: es común a todos los WebElements y tiene asociado un timeout global para todas las operaciones del driver
  - Tiempo de espera **explícito**: se establece de forma individual para cada WebElement

```
...
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(10,TimeUnit.SECONDS);
driver.get("www.google.com");
...
```

timeout implícito

```
  ...
  //Create Wait using WebDriverWait.
  //This will wait for 10 seconds for timeout before
  //title is updated with search term
  WebDriverWait wait = new WebDriverWait(driver, 10);
  wait.until(ExpectedConditions.titleContains("selenium"));
  ...
```

timeout explícito

Si el elemento se carga antes del límite especificado se cancela el timeout

MÚLTIPLES ACCIONES (GRUPOS DE ACCIONES)

Las acciones agrupadas se ejecutan de forma secuencial

- Podemos indicar a WebDriver que realice múltiples acciones agrupándolas en una acción compuesta, siguiendo estos tres pasos:
  - Invocar la clase **Actions** para agrupar las acciones (1)
    - La clase Actions se utiliza para emular eventos complejos de usuario
  - Construir la acción (**Action**) compuesta por el conjunto de acciones anteriores (2)
  - Realizar (ejecutar) la acción compuesta (3)

```
WebDriver driver = new FirefoxDriver();
driver.get("http://www.example.com");
WebElement one = driver.findElement(By.name("one"));
WebElement three = driver.findElement(By.name("three"));
WebElement five = driver.findElement(By.name("five"));
// Add all the actions into the Actions builder
Actions builder = new Actions(driver); (1)
// Generate the composite action
Action compositeAction = builder.keyDown(Keys.CONTROL)
        .click(one)
        .click(three)          (2)
        .click(five)
        .keyUp(Keys.CONTROL)
        .build();
// Perform the composite action.
compositeAction.perform(); (3)
```

En este ejemplo el usuario selecciona los tres elementos (manteniendo pulsada la tecla Ctrl mientras realiza la selección)

Acciones basadas en el ratón

- El método **click()** se utiliza para simular que pulsamos el botón izquierdo del ratón:
  - public Actions click()
    - Pulsación del botón izquierdo del ratón, independientemente o no de que estemos sobre algún elemento de la página
    - Este método suele usarse combinado con otros, para crear una acción compuesta. P.ej.

```
WebElement one = driver.findElement(By.name("one")); Actions builder = new Actions(driver);
//Click on One
builder.moveByOffset(one.getLocation().getX()+border, one.getLocation().getY()+border).click();
builder.build().perform();
```

  - public Actions click(WebElement onElement)
    - Pulsación del botón izquierdo del ratón sobre un WebElement

```
WebElement one = driver.findElement(By.name("one")); Actions builder = new Actions(driver);
//Click on One
builder.click(one); builder.build().perform();
```

- Método public Actions moveToElement(WebElement toElement)

```
WebElement one = driver.findElement(By.name("one")); Actions builder = new Actions(driver);
//Click on One
builder.moveToElement(one).click(); builder.build().perform();
```

- Otros métodos que podemos utilizar son:
  - public Actions doubleClick(); (doble click con botón izquierdo)
  - public Actions contextClick(); (botón derecho del ratón)

Acciones basadas en el teclado

- métodos **keyDown()** y **keyUp()**
  - public Actions keyDown(Keys theKey) throws IllegalArgumentException
    - Se genera una excepción si el argumento no es una de las teclas Shift, Ctrl, Alt
  - public Actions keyUp(Keys theKey)
- método **sendKeys()**
  - public Actions sendKeys(CharSequence keysToSend)
    - Se utiliza para teclear caracteres en elementos de la página como text boxes, …
  - también se puede utilizar el método WebElement.sendkeys(CharSequence k)
- Ejemplo:

```
driver = new FirefoxDriver();
driver.get(baseUrl);
WebElement linkText = driver.findElement(By.linkText("Element"));

Actions builder = new Actions(driver);
builder.contextClick(linkText)  //activamos el menú contextual de linkText
       .sendKeys(Keys.ARROW_DOWN)
       .sendKeys(Keys.ENTER)  //seleccionamos la primera de las opciones del menú
       .build()
       .perform();
```

Operaciones de navegación

- Navegar a la página anterior: driver.navigate().back()
- Navegar a la página siguiente: driver.navigate().forward()
- Métodos de refresco: driver.navigate().refresh()
- Manejo de frames: driver.switchTo.frame(index)
- Manejo de ventanas: driver.switchTo.window(window)
  - Ejemplo:
    - Si tenemos varias ventanas:

```
driver.get(baseUrl);
String window1 = driver.getWindowHandle();
System.out.println("First Window Handle is: "+window1);

link = driver.findElement(By.linkText("Google Search"));
link.click();
String window2 = driver.getWindowHandle();
System.out.println("Second Window Handle is: "+window2);
System.out.println("Number of Window Handles so for: " +driver.getWindowHandles().size());
driver.switchTo().window(window1);
```

    - Si tenemos varias pestañas en una única ventana:

```
//Open a new tab using Ctrl + t
driver.findElement(By.cssSelector("body")).sendKeys(Keys.CONTROL +"t");
//Switch between tabs using Ctrl + \t
driver.findElement(By.cssSelector("body")).sendKeys(Keys.CONTROL +"\t");
```

WebDriver y Maven

- Necesitamos incluir la dependencia con la librería de WebDriver en nuestro proyecto Maven:

```
<dependencies>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>3.14.0</version>
  </dependency>
</dependencies>
```

- ¿Dónde implementaremos nuestros tests de aceptación?
  - Opción 1: en src/test/java, junto con el resto de drivers del proyecto. Serán tests ejecutados por failsafe. En src/main/java tendremos el código fuente de nuestro proyecto
  - Opción 2: en src/test/java de un proyecto maven independiente (el proyecto únicamente contiene los tests de integración). Serán tests ejecutados por surefire. En src/main/java tendremos código que usarán nuestros tests (clases del patrón de diseño page object)

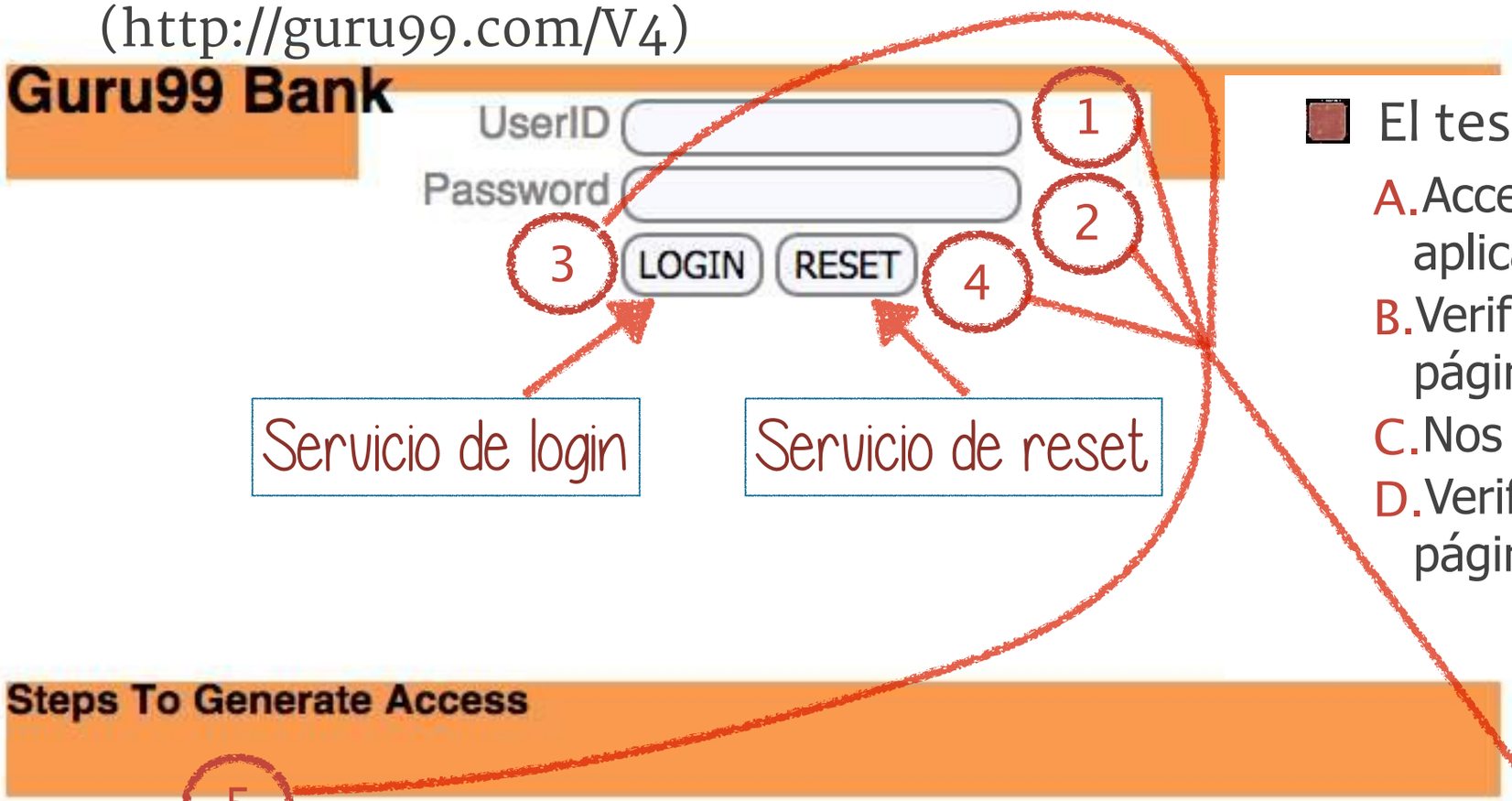
Mantenibilidad de Nuestros tests

- Los tests implementados para nuestra aplicación web, funcionarán siempre y cuando no se produzcan cambios en la aplicación
  - Si una o más páginas de nuestra aplicación web sufren cambios, tendremos que cambiar el código de nuestros tests (probablemente en muchos de ellos). P.ej. supongamos que un elemento de la página cambia su ID. Si dicho elemento es accedido desde N tests, tendremos que refactorizar todos ellos
- Para facilitar la mantenibilidad, y reducir la duplicación de código de nuestros tests es útil el patrón de diseño "**Page Object Pattern**"
  - Básicamente consiste en crear una clase para cada página web, en la que:
    - sus miembros (atributos) serán los elementos de la página web correspondiente, y
    - sus métodos serán todos los SERVICIOS que nos proporciona la página
- El API de Webdriver proporciona varios elementos para implementar este patrón:
  - Anotación @FindBy para inyectar los objetos que representan los elementos html de una página web
  - Clase PageFactory para obtener los objetos que representan las páginas html

Page Object Pattern. Ejemplo

ver http://www.guru99.com/page-object-model-pom-page-factory-in-selenium-ultimate-guide.html

- Supongamos que queremos implementar un test para una aplicación bancaria (http://guru99.com/V4)

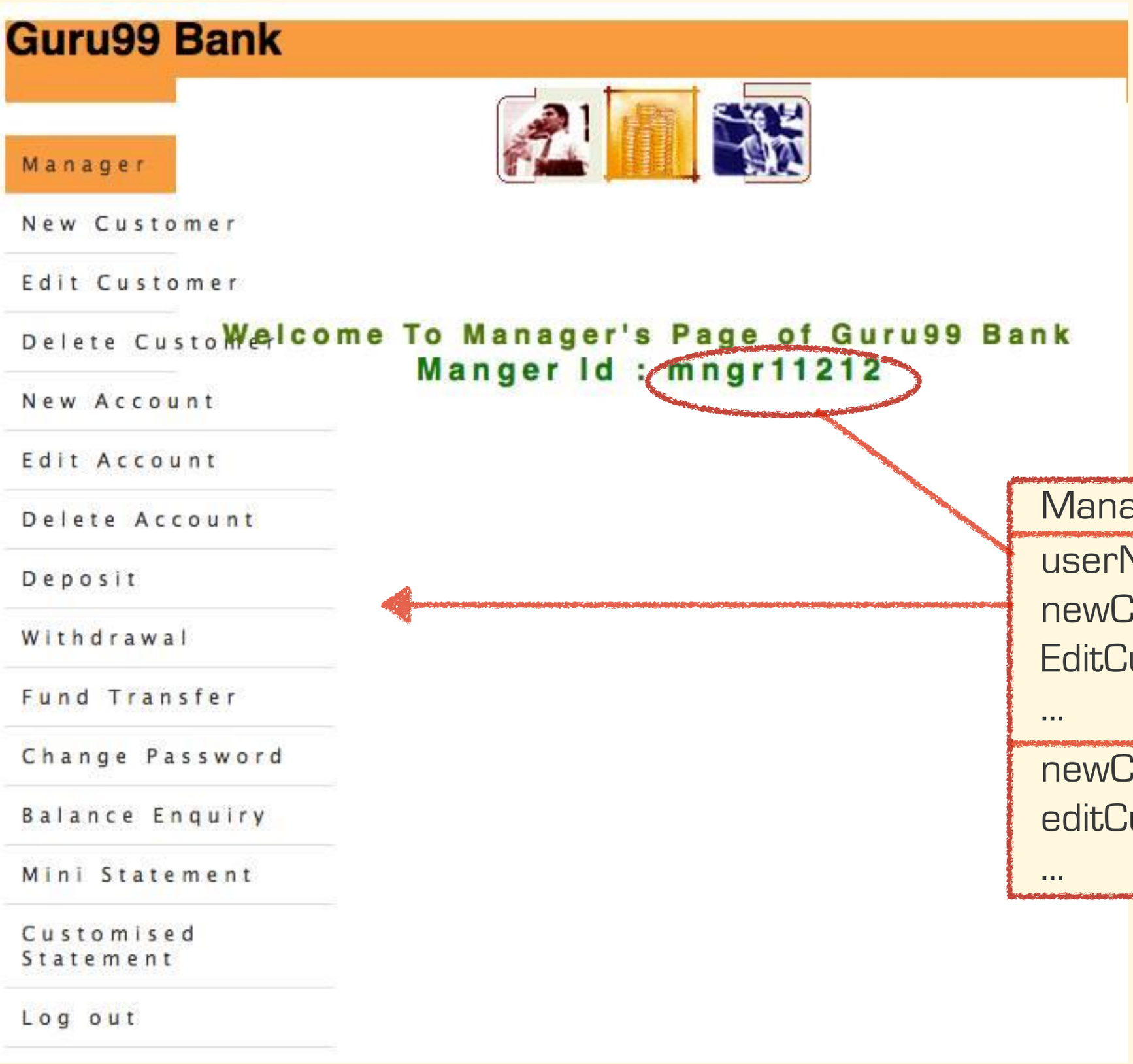


- El test consistirá en:
  - A. Accedemos a la url inicial de la aplicación
  - B. Verificamos que estamos en la página correcta
  - C. Nos logueamos
  - D. Verificamos que accedemos a la página correcta

- Los servicios se implementan como métodos
- Los elementos html se implementan como atributos del objeto PageObject

LoginPage	
userID password login reset info	elementos html
login() reset() inforegister()	servicios

Página web con los servicios del banco



Clases LoginPage y ManagerPage

- Estas clases estarán implementadas en src/main/java

```
public class LoginPage {
  WebDriver driver;
  WebElement userID;
  WebElement password;
  WebElement login;
  WebElement pTitle;

  public LoginPage(WebDriver driver){
    this.driver = driver;
    this.driver.get("http://demo.guru99.com/V4");
    userID = driver.findElement(By.name("uid"));
    password =
        driver.findElement(By.name("password"));
    login =
        driver.findElement(By.name("btnLogin"));
    pTitle =
       driver.findElement(By.className("barone"));
  }

  public void login(String user, String pass){
    userID.sendKeys(user);
    password.sendKeys(pass);
    login.click();
  }

  public String getPageTitle(){
    return  pTitle.getText();
  }
}
```

```
public class ManagerPage {
  WebDriver driver;
  WebElement homePageUserName;
  WebElement newCustomer;
  ...
  WebElement logOut;

  public ManagerPage(WebDriver driver){
    this.driver = driver;
    homePageUserName =
       driver.findElement(By.xpath("//table//
                                    tr[@class='heading3']"));
    newCustomer =
       driver.findElement(By.linkText("New Customer"));
    logOut =
       driver.findElement(By.linkText("Log out"));
  }

  public String getHomePageDashboardUserName(){
       return homePageUserName.getText();
  }
  ...
}
```

- Las clases que representan cada una de las páginas contienen código Webdriver y por lo tanto, dependen del código html de nuestra aplicación a probar

Clase TestLoginPage

- El test lo implementaremos en src/test/java

```
public class TestLoginPage {
    WebDriver driver;
    LoginPage poLogin;
    ManagerPage poManagerPage;

    @Before
    public void setup(){
        driver = new FirefoxDriver();
        poLogin = new LoginPage(driver);
    }

    @Test
    public void test_Login_Correct(){

        String loginPageTitle = poLogin.getLoginTitle();
        Assert.assertTrue(loginPageTitle.toLowerCase().contains("guru99 bank"));
        poLogin.login("mngr34733", "AbEvydU");
        poManagerPage = new ManagerPage(driver);

        Assert.assertTrue(poManagerPage
          .getHomePageDashboardUserName().toLowerCase().contains("manger id : mngr34733"));
        driver.close();
}
```

- El test **NO** contiene código webdriver, por lo tanto, es independiente del código html de nuestra aplicación a probar

- Acuérdate de cerrar el navegador después de cada test (o después de todos los tests)

PageFactory (I)

- La librería WebDriver nos proporciona la clase **PageFactory** para soportar el patrón PageObject
- La clase PageFactory proporciona objetos de nuestras clases PageObject
  - Para ello tendremos que: anotar los atributos de la clase PageObject con **@FindBy**,
  - y utilizar el método estático **PageFactory.initElements**() en el test:
    - ➡ initElements(WebDriver driver, java.lang.Class PageObjectClass)

```
public class LoginPage {
  WebDriver driver;
  @FindBy(name="uid") WebElement userID;
  @FindBy(name="password") WebElement password;
  @FindBy(name="btnLogin") WebElement login;
  @FindBy(className="barone") WebElement loginTitle;

  public LoginPage(WebDriver driver){
    this.driver = driver;
    this.driver.get("http://demo.guru99.com/V4");
  }

  public void login(String user, String pass){
        ...
  }

  public String getLoginTitle(){
        return  loginTitle.getText();
  }
}
```

```
public class ManagerPage {
  WebDriver driver;
  @FindBy(xpath="//table//tr[@class='heading3']")
  WebElement homePageUserName;
  @FindBy(linkText="New Customer")
  WebElement newCustomer;
  @FindBy(linkText="Log out") WebElement logOut;

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

  public String getHomePageDashboardUserName(){
    return homePageUserName.getText();
  }
}
```

PageFactory (II)

- Nuestro test usará la clase PageFactory para obtener instancias de las clases que representan las páginas web de la aplicación a probar

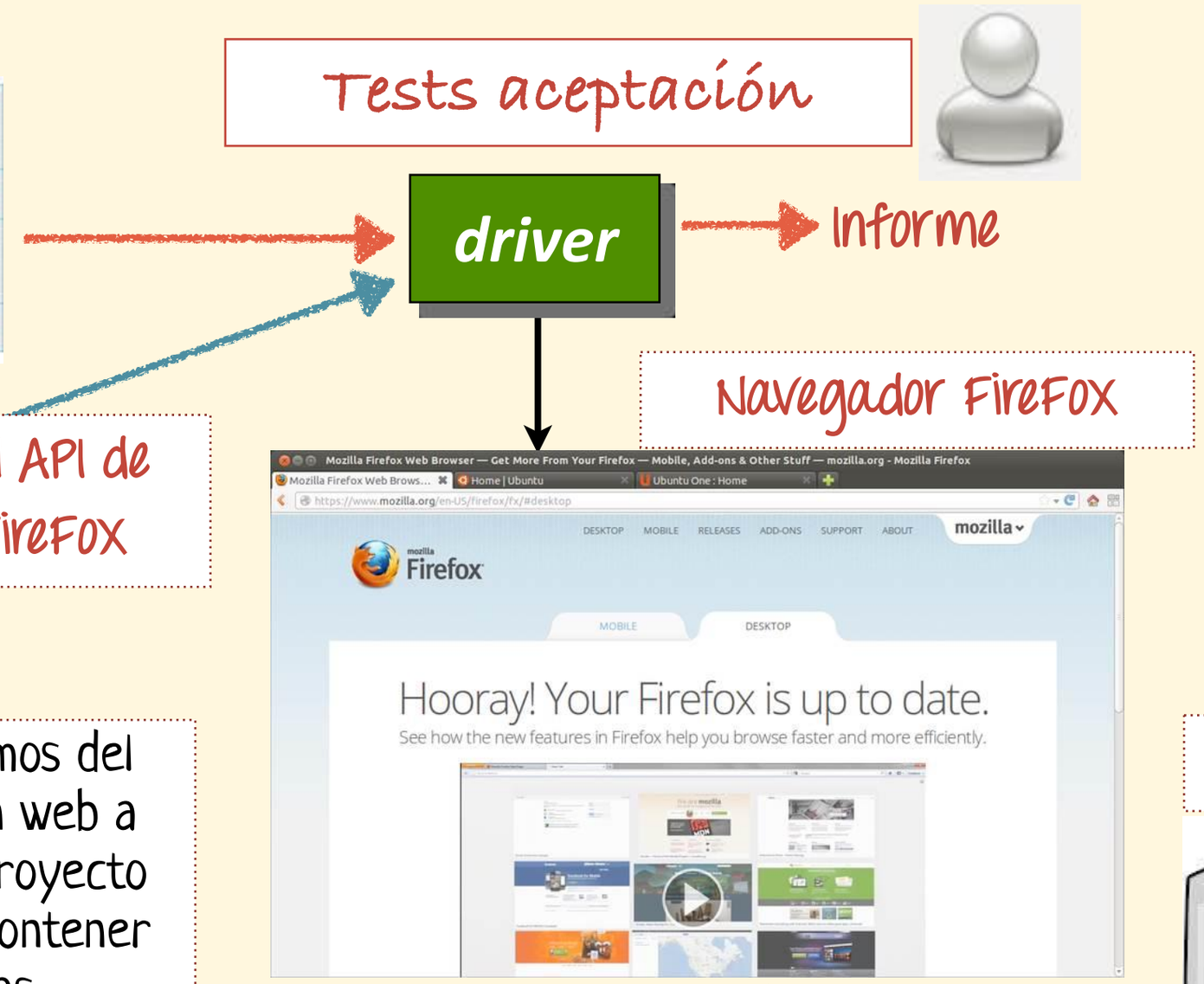
- El código del test queda mucho más "limpio" si esta línea la "movemos" al código del método LoginPage.login(), y hacemos que este método devuelva la nueva instancia de ManagerPage

```
public class TestLoginPage {
  WebDriver driver;
  LoginPage poLogin;
  ManagerPage poManagerPage;

  @Before
  public void setup(){
    driver = new FirefoxDriver();
    poLogin = PageFactory.initElements(driver, LoginPage.class);
  }

  @Test
  public void test_Login_Correct(){
     String loginPageTitle = poLogin.getLoginTitle();
     Assert.assertTrue(loginPageTitle.toLowerCase().contains("guru99 bank"));

     poLogin.login("mngr34733", "AbEvydU");
     poManagerPage = PageFactory.initElements(driver, ManagerPage.class);

     Assert.assertTrue(poManagerPage.getHomePageDashboardUserName()
                                    .toLowerCase().contains("manger id : mngr34733"));
     driver.close();
  }
}
```

Y AHORA VAMOS AL LABORATORIO...

Vamos a implementar tests de aceptación (para validar propiedades emergentes funcionales) para una aplicación web con selenium WebDriver

Camino	Datos Entrada	Resultado Esperado
C1	d1=... d2=... ...	r1
..		
CM	d1=... d2=... ...	rM



usaremos JUnit y el API de WebDriver para FireFox

Dado que no disponemos del código de la aplicación web a probar, usaremos un proyecto Maven que sólo va a contener código de pruebas

Referencias bibliográficas

- Selenium WebDriver Practical Guide. Satya Avasarala. Packt Publishing. 2014
  - Capítulos 1,2 y 9
- Selenium design patterns and best practices : build a powerful, stable, and automated test suite using Selenium WebDriver. Dima Kovalenko, Jim Evans, Jeremy Segal. Packt Publishing, 2014
  - Capítulo 7: The Page Object Pattern