



# FRAMEWORK PENTRU TESTAREA AUTOMATĂ A APLICAȚIILOR WEB FOLOSIND SELENIUM WEBDRIVER ȘI GHERKIN

PROIECT DE DIPLOMĂ

Autor: **Ruxandra-Mara IERIMA**

Conducător științific: **Prof.dr.ing. Honoriu VĂLEAN**

**2019**



**UNIVERSITATEA TEHNICĂ**

DIN CLUJ-NAPOCA

**FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE**

Vizat,

DECAN

**Prof.dr.ing. Liviu MICLEA**

DIRECTOR DEPARTAMENT AUTOMATICĂ

**Prof.dr.ing. Honoriu VĂLEAN**

Autor: **Ruxandra-Mara IERIMA**

**„Framework pentru testarea automată a aplicațiilor web  
folosind Selenium WebDriver și Gherkin”**

1. **Enunțul temei:** Realizarea unui framework de testare automată pentru o aplicație web, prin intermediul Selenium WebDriver și limbajului non-tehnic Gherkin.
2. **Conținutul proiectului:** *(enumerarea părților componente) Pagina de prezentare, Declarație privind autenticitatea proiectului, Sinteza proiectului, Cuprins, Introducere, Stadiul actual, Fundamentare teoretică, Implementarea soluției, Rezultate experimentale, Concluzii, Bibliografie, Anexe.*
3. **Locul documentației:** *Universitatea Tehnică din Cluj-Napoca*
4. **Consultanți:** Prof.dr.ing. Honoriu VĂLEAN
5. **Data emiterii temei:** 01.11.2018
6. **Data predării:** 12.07.2019

Semnătura autorului

\_\_\_\_\_

Semnătura conducătorului științific

\_\_\_\_\_



**UNIVERSITATEA TEHNICĂ**

DIN CLUJ-NAPOCA

**FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE**

**Declarație pe proprie răspundere privind  
autenticitatea proiectului de diplomă**

Subsemnatul(a) **Ruxandra-Mara IERIMA**,  
legitimat(ă) cu CI/BI seria CJ nr. 406221, CNP 2970402244485,  
autorul lucrării:

Framework pentru testarea automată a aplicațiilor web  
folosind Selenium WebDriver și Gherkin

elaborată în vederea susținerii examenului de finalizare a studiilor de licență la **Facultatea de Automatică și Calculatoare**, specializarea **Automatică și Informatică Aplicată**, din cadrul Universității Tehnice din Cluj-Napoca, sesiunea Iulie 2019 a anului universitar 2018-2019, declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, pe baza cercetărilor mele și pe baza informațiilor obținute din surse care au fost citate, în textul lucrării, și în bibliografie.

Declar, că această lucrare nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române și a convențiilor internaționale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în fața unei alte comisii de examen de licență.

În cazul constatării ulterioare a unor declarații false, voi suporta sancțiunile administrative, respectiv, *anularea examenului de licență*.

Data

12.07.2019

Ruxandra-Mara IERIMA

\_\_\_\_\_  
(semnătura)



## SINTEZA

proiectului de diplomă cu titlul:

### **„Framework pentru testarea automată a aplicațiilor web folosind Selenium WebDriver și Gherkin”**

Autor: **Ruxandra-Mara IERIMA**

Conducător științific: **Prof.dr.ing. Honoriu VĂLEAN**

1. Cerințele temei: Asigurarea calității unei aplicații web prin intermediul unui framework pentru testarea automată, bazat pe Selenium WebDriver și limbajul non-tehnic Gherkin.
2. Soluții alese: Tehnologiile folosite, pentru realizarea părții practice a acestei lucrări sunt framework-ul Selenium WebDriver folosit pentru interacțiunea cu elementele din pagina web, librăria Behave utilizată pentru scrierea testelor într-un limbaj natural Gherkin, susținute de codul scris în Python. Pentru raport s-a ales framework-ul Allure.
3. Rezultate obținute: În partea practică s-a implementat o suită formată din cinci teste, care au ca scop verificarea funcționalităților unui magazin online. Verificarea se realizează prin reproducerea acțiunilor unui utilizator. Aceste acțiuni constau în crearea unui cont nou, înregistrarea folosind diferite credențiale, adăugarea produselor în coș, sortarea produselor după diferite criterii și procesarea unei comenzi
4. Testări și verificări: Statisticile în urma rulării pot fi vizualizate în raportul generat prin intermediul framework-ului Allure. Validarea s-a realizat prin implementarea diferitelor scenarii pe baza necesităților unui utilizator.
5. Contribuții personale: Contribuția personală a constat în crearea cazurilor de testare distribuie pe fiecare funcționalitate în parte.
6. Surse de documentare: Partea teoretică a lucrării a fost realizată pe baza surselor bibliografice găsite din cărți de specialitate, articole de pe internet și secvențe de cod.

Semnătura autorului \_\_\_\_\_

Semnătura conducătorului științific \_\_\_\_\_

# Cuprins

<b>1</b>	<b>INTRODUCERE.....</b>	<b>3</b>
1.1	CONTEXT GENERAL .....	3
1.2	OBIECTIVE.....	4
1.3	SPECIFICAȚII .....	4
<b>2</b>	<b>STADIUL ACTUAL.....</b>	<b>5</b>
2.1	EVOLUȚIA CONCEPTULUI DE TESTARE .....	5
2.2	METODOLOGII UTILIZATE ÎN PROCESUL DE TESTARE .....	6
2.3	ATESTATE .....	8
<b>3</b>	<b>FUNDAMENTARE TEORETICĂ.....</b>	<b>9</b>
3.1	ASPECTE GENERALE DESPRE TESTARE .....	9
3.11	<i>Definirea și scopurile testării .....</i>	<i>9</i>
3.12	<i>Obiectivele testării .....</i>	<i>9</i>
3.13	<i>Nivele de testare .....</i>	<i>10</i>
3.14	<i>Testarea automată .....</i>	<i>12</i>
3.2	SELENIUM WEBDRIVER .....	14
3.21	<i>Introducere .....</i>	<i>14</i>
3.22	<i>Suita Selenium .....</i>	<i>14</i>
3.23	<i>Comenzi utile din Selenium WebDriver .....</i>	<i>15</i>
3.24	<i>Metodele de identificare a elementelor.....</i>	<i>16</i>
3.25	<i>Interacțiunea cu elementele web .....</i>	<i>22</i>
3.26	<i>Proiectarea testelor automate.....</i>	<i>24</i>
3.3	GHERKIN.....	26
3.31	<i>Introducere în BDD.....</i>	<i>26</i>
3.32	<i>Limbajul Gherkin.....</i>	<i>27</i>
<b>4</b>	<b>IMPLEMENTAREA SOLUȚIEI.....</b>	<b>31</b>
4.1	DESCRIEREA APLICAȚIEI .....	31
4.2	CONFIGURAREA PROGRAMELOR .....	33
4.3	IMPLEMENTAREA CODULUI .....	36
<b>5</b>	<b>REZULTATE EXPERIMENTALE.....</b>	<b>49</b>
6.1	RULAREA TESTELOR.....	49
6.2	GENERAREA RAPORTULUI DE VALIDARE .....	50
<b>6</b>	<b>CONCLUZII .....</b>	<b>52</b>
6.1	DEZVOLTĂRI ULTERIOARE .....	52

7	BIBLIOGRAFIE.....	54
8	ANEXE.....	56

# 1 Introducere

## 1.1 Context general

Trăim într-o eră în care aplicațiile *web* devin din ce în ce mai utilizate în aproape toate domeniile. Putem spune că aceste aplicații au devenit cruciale pentru firmele care prestează servicii, pentru cele care realizează vânzări *online*, cât și pentru cele care au ca scop informarea unei mase de oameni. Din cauza ritmului ridicat cu care acestea se dezvoltă, odată cu creșterea numărului de aplicații pe piață, cresc și așteptările clienților. O aplicație sau un site *web* pot fi cartea de vizită a companiei, cu cât acestea sunt mai informative, mai accesibile, mai sigure și mai rapide, cu atât este mai bună imaginea companiei construită în mintea clienților săi. Pentru a avea toate aceste calități, aplicațiile *web* trebuie să fie bine testate.

Testarea acestor aplicații este foarte importantă în asigurarea calității prin eliminarea defectelor, creșterea performanței și validarea funcționalităților. Alegerea testării automate ajută la reducerea semnificativă a defectelor, timpului dedicat testării și costurilor.

Această lucrare prezintă o opțiune pentru testarea funcțională a unui magazin *online*. Pentru aceasta s-a ales prezentarea unui *framework* de testare automată folosind *Selenium WebDriver* împreună cu un limbaj comun, non-tehnic *Gherkin*. Un *framework* de testare este un sistem care integrează toți pașii și programele folosite pentru realizarea testării. Acesta stabilește regulile de automatizare pentru testarea unui anumit produs.

S-a ales folosirea *framework*-ului *Selenium WebDriver* deoarece a fost dezvoltat exclusiv pentru testarea aplicațiilor *web*, însumând astfel toate funcționalitățile necesare pentru realizarea testării. A fost aleasă utilizarea limbajul *Gherkin*, un limbaj non-tehnic, deoarece poate fi înțeles de oricine, fără a fi nevoie de cunoștințe de programare.

În capitolul doi se prezintă cum s-a dezvoltat conceptul de testare și evoluția sa, focusată în ultimii ani asupra metodologiilor moderne. Tot în acest capitol se prezintă și certificările care se pot obține în cadrul acestui domeniu.

În capitolul trei se dezvoltă ideile care stau la baza procesului de testare, cât și clasificarea testării. Se pune accentul pe înțelegerea cât mai în detaliu a programelor folosite, *Selenium* și *Gherkin*.

În capitolul patru este prezentată implementarea unei suite cu cinci teste, exemplificând astfel modul de testare a principalelor funcționalități din aplicație.

În capitolul cinci este reprezentat modul în care se poate realiza raportul, ce conține rezultatele rulării. Din acest raport se pot extrage ușor defectele găsite.

În ultimul capitol, s-au evidențiat concluziile trase în urma realizării acestui proiect și ulterioarele dezvoltări posibile ale *framework*-ului.

## 1.2 Obiective

Principalul obiectiv al acestei lucrări constă în prezentarea unei soluții de testare automată având o structură organizată. Testele trebuie să fie scrise sub formă de scenarii simple, despre modul în care o aplicație ar trebui să se comporte din perspectiva unui utilizator.

Prin crearea testelor într-un limbaj natural, în limba engleză, putem implica persoanele din *management* să înțeleagă și chiar să conceapă teste, fără a fi nevoie ca acestea să aibă cunoștințe tehnice. Lucru care aduce un plus proiectului, deoarece aceste persoane cunosc mai bine cerințele afacerii și pot face recomandări în privința aceasta. Abordarea ajută echipele să se înțeleagă reciproc, creând o puternică colaborare.

Ca obiective generale ale lucrării putem defini următoarele:

- Implementarea modelului *Page Object*
- Crearea unei suite formată din cinci teste, care să acopere principalele funcționalități ale aplicației. Implementarea testelor se va realiza cu ajutorul limbajului *Gherkin* din cadrul procesului de dezvoltare a software-ului agil *BDD*(eng. *Behavior-Driven Development*)
- Rularea testelor, prin intermediul *runner*-ului *Behave*
- Generarea unui raport de gestionare a defectelor în urma rulării testelor

## 1.3 Specificații

Pentru realizarea obiectivelor prezentate mai sus, s-au definit următoarele activități:

- Modelul *Page Object* presupune crearea unor clase orientate pe obiect, unde fiecare pagină din aplicația *web* este reprezentată printr-o clasă. Acest model este un *design pattern* ce presupune, ca de fiecare dată când vrem să interacționăm cu un element dintr-o pagină, vom apela funcțiile declarate în clasa respectivă
- Implementarea testelor presupune în primul rând crearea unor scenarii simple, care să exprime foarte clar ce funcționalitate va fi testată. Scenariile vor fi scrise în limbajul *Gherkin*, cu ajutorul *framework*-ului *Behave*.
- Testele vor putea fi rulate în funcție de necesități, fie unul câte unul, grupate sau toate deodată
- Raportul generat trebuie să fie unul detaliat pentru a ne putea da seama de statusul testului, respectiv aplicației. Acest raport va conține informații cu privire la numărul testelor care au îndeplinit cerințele, cele care au găsit un defect în aplicație, dar și cele care conțin erori în scripturi. Alte informații folositoare pot fi: timpul în care au fost executate testele, erorile apărute, etc.



## 2 Stadiul actual

Testarea software are ca scop oferirea de informații cu privire la calitatea unui produs sau a unui serviciu. În investigarea produsului supus testării se va lua în considerare contextul în care acesta va fi utilizat. Pentru a putea înțelege și estima riscurile pe care le reprezintă implementarea produsului în dezvoltare, testarea software oferă o perspectivă obiectivă.

Această testare reprezintă procesul de identificare a defectelor și a erorilor unui produs prin observarea diferențelor dintre un comportament dorit și un comportament observat cu ajutorul diferitelor tehnici de testare. Un alt aspect al testării software reprezintă validarea produsului finit prin verificarea faptului că acesta corespunde așteptărilor.

Testarea este foarte importantă deoarece chiar și unele erori minore dintr-o aplicație pot avea efecte negative precum costuri suplimentare sau desconsiderarea calității acelui produs de către client.

Acest concept nu este unul nou apărut, însă dezvoltarea și identificarea de noi tehnologii este în continuă creștere datorită standardelor și așteptărilor ridicate ale clienților [1].

### 2.1 Evoluția conceptului de testare

Conceptul de testare a început să fie tot mai adoptat în zilele noastre. Despre cum a luat naștere acest concept și a început să se dezvolte, *Gelperin și W.C. Hetzel* au realizat o clasificare a fazelor și obiectivelor testării:

#### **Orientarea spre depanare (până în 1956)**

Această perioadă a fost orientată mai mult spre partea *hardware*, unde testarea a fost adesea asociată cu depanarea: neexistând o diferență clară între teste și depanare. Pe partea de *software*, defectele nu erau considerate atât de importante, în general persoanele care se ocupau cu implementarea codului făceau și partea de testare.

Prima referire la termenul de “bug” în domeniul informatic, a luat naștere când pe un circuit electronic defectuos s-a descoperit o molie.

#### **Orientarea spre demonstrație (1957-1918)**

Diferența dintre depanare și testare începea să se distingă în această perioadă. Accentul pe testare și pe remedierea defectelor a crescut, persoanele implicate în dezvoltarea de programe informatice fiind mult mai conștiente de această problemă. Scopul sistemelor *software* era acela de a îndeplini cerințele.

#### **Orientarea spre defectare (1979-1982)**

În această perioadă se propune, pe lângă procesul de testare, o serie de activități de analiză și control cu scopul creșterii calității produsului. Această abordare vine ca o rezolvare a testării bazate pe demonstrație, unde setul de date introdus pentru validarea procesului putea fi ales involuntar în favoarea produsului.

### Orientarea spre evaluare (1983-1987)

Principalul scop a fost evaluarea produsului în timpul ciclului de viață al acestuia, cât și alegerea diferitelor metode de testare și validare în funcție de caracteristicile sistemului, în ideea creșterii calității. În această perioadă au apărut și primele locuri de muncă dedicate de tester.

### Orientarea spre prevenire (1988-prezent)

Pentru a reduce costurile de dezvoltare și de întreținere a unui sistem software, testarea a fost introdusă încă din primele faze, în paralel cu implementarea produsului. Încep să fie definite primele activități principale ale testării precum: planificarea, analiza, proiectarea, implementarea, execuția și întreținerea. Prin respectarea acestor metode, defectele apărute în urma implementării aplicației scad considerabil.

Conceptul de testare devine din ce în ce mai utilizat în cam toate domeniile, din această cauză se caută îmbunătățiri constante, și o evoluție cât mai rapidă asupra acestui concept. Pe parcurs s-au dezvoltat metodologii moderne care se concentrează pe integrarea procesului de testare cu restul proceselor care ajută la dezvoltarea produsului [1].

## 2.2 Metodologii utilizate în procesul de testare

Metodologiile de testare a software-ului reprezintă diferite strategii sau abordări utilizate în testarea unei aplicații pentru a certifica faptul că sistemul funcționează la așteptările clientului. Ce le diferențiază pe aceste metodologii de alte tehnici de testare, este faptul că echipele de dezvoltare și cele de testare lucrează strâns legat una cu cealaltă.

- **Testarea *Waterfall*** – reprezintă un proces secvențial cu diferite faze prezentate în Figura 2.1, unde faza următoare începe doar în momentul în care toate etapele din faza anterioară au fost finalizate.

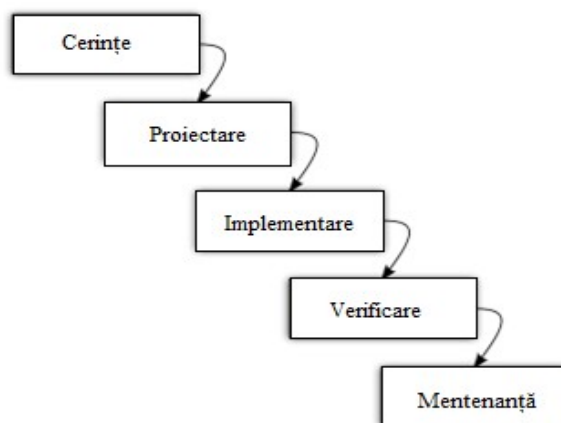


Figura 2.1. Etapele testării Waterfall [2]

1. Prima fază a modelului este faza de „Cerințe” în care toate cerințele proiectului sunt definite în întregime înainte de a începe testarea. În această

fază, echipa de testare definește domeniul de aplicare al testării, strategia de testare și elaborează un plan detaliat de testare.

2. În faza de „Proiectare” echipa stabilește detaliile tehnice precum limbajele de programare care urmează să fie folosite.
  3. A treia fază este cea de „Implementare” unde se creează testele și se implementează funcționalitățile care ajută la dezvoltarea acestora.
  4. Faza de „Verificare” reprezintă procesul de rulare al testelor, și comunicarea rezultatelor.
  5. Ultima fază, cea de „Mentenanță”, este etapa de întreținere a produsului *software*, care poate necesita schimbări ulterioare, în funcție de cerințele clientului [2].
- **Testarea Agile** - este un proces de testare care urmează principiile *Agile* ale dezvoltării software, unde cerințele sunt specificate treptat de către client. Procesul este unu continuu în care exista o integrare între implementare și testare, scopul comun fiind obținerea unei calități ridicate.

Testarea Agile este nestructurată în comparație cu abordarea în cascadă, existând o planificare minimă. Fiindcă procesul de testare se realizează chiar de la începutul proiectului, erorile pot fi rezolvate mult mai rapid. Avantajele acestei metodologii sunt flexibilitatea, adaptabilitatea la schimbări și o mai bună determinare a problemelor prin întâlniri zilnice între echipa de testare și cea de dezvoltare [3].

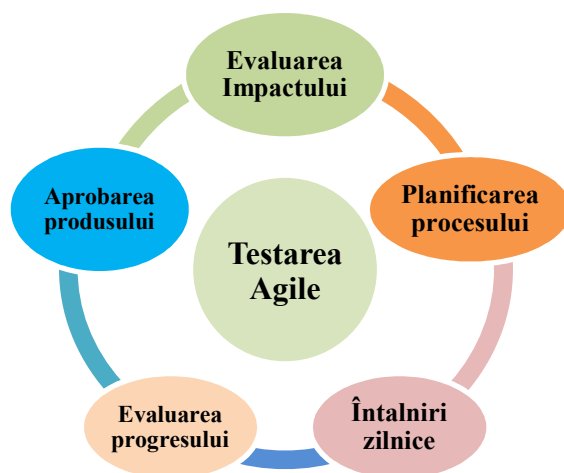


Figura 2.2. Etapele testării Agile [3]

- **Metodologia Scrum** - activitățile de testare din cadrul acestui proces implică estimarea efortului de către tester cu privire la timpul necesar pe care îl dedica pentru testarea unei anumite secțiuni din aplicație. O altă activitate importantă din acest proces este aceea de prioritizare a defectelor după gradul de severitate pe care acestea le prezintă [4].

## 2.3 Atestate

Testarea a devenit un aspect foarte important într-un proiect, iar de aceea se caută din ce în ce mai mulți oameni capabili în acest domeniu, cu cât mai multe cunoștințe. Odată cu această nevoie de oameni bine pregătiți au apărut mai multe certificări și cursuri specializate.

Unul dintre cele mai cunoscute certificări este ISTQB(*eng. International Software Testing Qualification Board*). Este o organizație non-profit de certificare a calificărilor de testare software care operează la nivel internațional, fondată din 2002. Această organizație pune la dispoziție programe de *training* pentru dobândirea de cunoștințe și abilități necesare unui bun tester. Aceștia au definit sistemul de certificare ISTQB, care a devenit lider mondial în certificarea competențelor de testare software. Cursurile sunt oferite doar de către entități acreditate. În cadrul acestuia există trei nivele de certificări:

1. *Foundation Level* – acest nivel prezintă aspecte generale despre testarea *software*, fiind adresat tuturor persoanelor implicate în acest proces.
2. *Advance Level* – pentru a putea susține acest examen este nevoie de a avea primul certificat. În acest nivel sunt prezentate principii de testare mai avansate, de aceea sunt obligatorii cunoștințele dobândite din primul nivel. În funcție de direcția aleasă în carieră sau de preferințele persoanei se poate alege una din cele 3 sub-categorii ale acestui nivel:
  - *Test Analyst* se referă la metodele de testare din cadrul tehnicii *Black Box*
  - *Test Manager* se ocupă de planificarea și controlul procesului de testare
  - *Technical Test Analyst* această categorie cuprinde testarea componentelor, unde este necesară cunoașterea tehnicii *White Box*
3. *Expert Level* – acest nivel prezintă cunoștințe despre mai multe ramuri din testare. A fi un expert în testare înseamnă a poseda și a prezenta abilități și cunoștințe speciale, derivate din pregătire și experiență. Scopul final fiind de a putea aplica aceste cunoștințe în situații reale. Pentru a intra în posesia unui astfel de certificat este obligatoriu obținerea nivelelor anterioare, iar pe lângă asta este obligatoriu ca participantul să aibă minim cinci ani vechime în acest domeniu.

Modul de examinare constă din susținerea unui examen de tip grilă, cu răspunsuri multiple. Trebuie completat minimum 65% din examen pentru a primi certificarea. Nivelul 3 conține o examinare adițională ce constă din realizarea unui eseu. O dată intrat în posesia certificării, aceasta este valabilă pe viață și nu necesită reînnoiri periodice [5].

## 3 Fundamentare teoretică

### 3.1 Aspecte generale despre testare

#### 3.1.1 Definirea și scopurile testării

Unele dintre definițiile și citatele des întâlnite care descriu conceptul de testare sunt:

„Testarea este procesul prin care se execută un program cu intenția de a găsi erori”  
(Myers)

„Procesul de exercitare sau de evaluare a unui sistem manual sau automatizat înseamnă să se verifice dacă îndeplinește condițiile specificate sau pentru a identifica diferențele dintre rezultatele așteptate și cele reale.” [6]

„Procesul constă în toate activitățile din ciclul de viață al unui produs, atât din punct de vedere static, cât și dinamic, referindu-se la planificarea, pregătirea și evaluarea produselor software și a produselor de lucru aferente pentru a determina dacă îndeplinesc cerințele specificate, pentru a demonstra că sunt adecvate scopului și pentru a detecta defectele.” [7]

O concluzie pe care o putem trage în urma definițiilor enumerate mai sus este aceea că scopul testării constă în asigurarea calității unui produs, la standardele cerute de către client. De aceea, când o organizație investește într-un produs software, acesta are obligația de a evalua calitatea produsului din perspectiva utilizatorilor. Testarea Software este procesul care ajută la realizarea acestei evaluări într-un mod cât se poate de obiectiv.

Unul dintre marile avantaje aduse de testare este diminuarea pierderilor. *NIST*(Institutul Național de Standarde și Tehnologie) a realizat un studiu în anul 2002 care arată că defectele și erorile apărute în produsele software au cauzat pierderi anuale estimate la 59.5 miliarde de dolari în economia S.U.A.

#### 3.1.2 Obiectivele testării

În funcție de caracteristicile fiecărui proiect în parte, testarea are diferite obiective:

- Să evalueze cerințele, proiectarea și codul
- Să verifice dacă au fost îndeplinite toate cerințele specificate
- Să valideze dacă produsul testat cuprinde toate caracteristicile și se ridică la așteptările utilizatorilor sau a altor părți interesate
- Mărirea încrederii în nivelul de calitate a produsului
- Prevenirea defectelor
- Găsirea defectelor
- Reducerea nivelului de risc pentru calitatea necorespunzătoare a software-ului
- Creșterea longevității produsului [8]

### 3.13 Nivele de testare

Fiecare nivel de testare reprezintă o instanță a procesului de testare, cuprinzând factori care influențează acest proces din cadrul unei organizații precum: bugetul și resursele, complexitatea, standarde interne și externe cât și cerințele contractuale [8].

**Testarea componentelor** – este știută și sub numele de testarea unității(*eng. Unit Testing*), unde sunt testate unitățile/componentele individuale ale *software*-ului. O componentă este definită ca cea mai mică parte testabilă din aplicație. Aceasta poate reprezenta structuri de cod și date, clase, module de baze de date sau chiar componente mai ample, ele depinzând de complexitatea aplicației. Acest nivel de testare este primul efectual asupra codului, imediat după ce acesta a fost scris. De aceea, testarea componentelor este efectuală de obicei de către *developeri*, necesitând un grad de cunoștințe de programare mai ridicat.

Scopul este de a valida faptul că fiecare unitate a *software*-ului funcționează așa cum a fost proiectată [8].

**Testarea integrării** – aceasta presupune testarea interacțiunilor dintre componentele sistemului. De exemplu pentru a verifica direcționarea de la o pagină la alta prin intermediul unui buton de „*LogIn*”. Scopul acestei testări nu este de a verifica funcționalitatea componentelor luate individual, deoarece aceasta a fost făcută în testarea componentelor, descrisă mai sus. Principalul obiectiv al acestei testări este de a vedea cum comunică componentele între ele, iar în cazul unei componente nou integrată în sistem se verifică cum se integrează aceasta cu cele deja existente. Chiar dacă fiecare componentă a fost testată individual, în general aceste componente sunt dezvoltate de mai mulți *developeri* care pot avea viziuni diferite, de aceea testarea integrării este un aspect foarte important. Există unele tehnici folosite la această testare precum:

Tehnica „*Big Bang*” presupune ca prima oară, toate componentele să fie integrate iar mai apoi testate. Această tehnică este folosită de obicei în proiectele de dimensiuni mici, deoarece dezavantajul este de a apărea un număr mare de erori doar la sfârșit.

Tehnica „*Incremental*” este o abordare progresivă care este împărțită în două metode, prima fiind testarea de jos în sus care presupune ca testarea să înceapă de la cele mai inferioare nivele și să crească progresiv până când toate modulele au fost testate. Cea de a doua metodă, se numește testarea de sus în jos unde testarea începe de la cel mai înalt nivel din ierarhie, urmând mai apoi să fie testate și restul nivelelor [8].

**Testarea sistemului** – se referă în principiu la comportamentul întregului sistem, în această parte toate componentele sunt puse împreună și verificate atât din punct de vedere funcțional cât și non-funcțional. Această metodă verifică ca fiecare intrare din aplicație să corespundă cu ieșirea așteptată, folosind diferite tehnici care includ: rapoarte de analiză a riscurilor, diagrame de stare, modele de comportament al sistemului sau modele de utilizare. În prezent există mai mult de 50 de tipuri de testare a sistemului, însă cele mai des întâlnite sunt [9]:

- *Usability Testing (Testarea Utilizabilității)* este o tehnică de testare care constă în evaluarea gradului de ușurință cu care este utilizat un sistem/produs de către un grup de utilizatori reprezentativi. Numărul utilizatorilor solicitați diferă de la un proiect la altul în funcție de complexitatea acestuia. În general se supraveghează utilizatorii în încercarea lor de a-și îndeplini sarcinile. Adesea această tehnică se realizează încă de la începutul dezvoltării produsului și până la finalizarea acestuia. Principalul beneficiu al acestei testări este identificarea timpurie a problemelor existente cu privire la utilizarea produsului din perspectiva unui utilizator.
- *„Load Testing” (Testarea Încărcării)* aceasta determină performanța unui sistem în condițiile existente din viața reală. Obiectivul principal este de a determina comportamentul sistemului în cazul în care de exemplu, mai mulți utilizatori l-ar folosi simultan. Alte obiective importante sunt sustenabilitatea aplicației și verificare infrastructurii acesteia, dacă este suficientă pentru rulare. Un site web al unei companii aeriene nu a reușit să gestioneze 10000 de utilizatori în timpul unei oferte de festival, o companie de jocuri nu și-a luat măsuri în cea ce privește creșterea traficului în urma unei campanii publicitare care a dus atât la pierderea banilor investiți în campanie, cât și la vânzările potențiale de jocuri. Acestea sunt doar câteva dintre exemplele care dovedesc importanța acestei testări și efectele negative pe care le poate avea un sistem care nu adoptă această tehnică înainte ca produsul să iasă pe piață.
- *„Regression Testing”(Testarea de regresie)* este o tehnică folosită pentru a ne asigura că schimbările recente în program nu prezintă un efect negativ asupra celor deja existente. Testarea prin regresie constă în re-executarea cazurilor de testare asupra sistemului, după adăugarea unor noi componente, pentru a avea o bună funcționare a acestuia. Această tehnică se poate aplica asupra tuturor testelor, dezavantajul este că necesită timp și resurse mari, sau asupra unor părți din sistem, selectarea testelor fiind făcută în funcție de funcționalitățile critice și cele mai des utilizate din aplicație.
- *„Functional Testing”(Testarea funcțională)* verifică ca toate funcționalitățile aplicației să corespundă cu cerințele prin realizarea unui set vast de date. Acestea sunt introduse în aplicație și vor fi comparate cu rezultatele așteptate. Această testare nu presupune acces la codul sursă, ea putând fi realizată atât automat cât și manual. Funcționalitățile care pot fi testate sunt de la cele mai de bază cum ar fi navigarea dintr-o pagină în alta, accesibilitatea sistemului pentru utilizator, și până la cele mai importante funcționalități precum înregistrarea unui cont sau plasarea unei comenzi.

**Testarea de acceptare** – această testare se realizează când proiectul este aproape de final, testarea sistemului fiind făcută din perspectiva utilizatorilor. Aceasta nu are ca scop, ca în cazul celorlalte tehnici găsirea defectelor, ci mai degrabă se concentrează pe gradul de

adecvare a utilizării și ușurința cu care se poate folosi aplicația. Găsirea unui număr mare de defecte în această fază trebuie să ridice un semn de întrebare, deoarece acestea prezintă un risc major asupra proiectului. Obiectivul principal al testării de acceptare este de a construi încrederea că utilizatorul poate folosi aplicația pentru a-și îndeplini nevoile. Una din cele mai cunoscute metode folosite este „*Alpha and Beta testing*”, o procedură care se utilizează pe site-ul firmei, efectuată de către potențialii utilizatori, unde se urmărește validarea că aplicația funcționează în condițiile așteptate, în diferite medii de utilizare.

Toate tehnicile prezentate mai sus sunt utilizate cu scopul creșterii calității produselor. Acestea necesită însă un timp îndelungat și resurse pentru a putea fi executate. După cum știm trăim în era tehnologiilor informaționale care este în continuă creștere, iar din aceasta cauză tot mai multe firme au început să adopte testarea automată pentru aplicațiile web cu scopul reducerii timpului necesar testării, precum și riscurile generate de către defecte [8].

### 3.14 Testarea automată

Testarea automată reprezintă executarea testelor prin intermediul unor programe automate, fără a necesita intervenția omului. Această metodă s-a dezvoltat datorită nevoii de a reduce considerabil timpul execuției cât și de a elimina munca repetitivă realizată mai ales în testarea regresivă [10].

Testarea manuală și testarea automată acoperă două zone vaste. În cadrul fiecărei categorii, sunt disponibile metode de testare specifice, diferențele dintre acestea două sunt prezentate în Tabelul 3.1 [11]:

*Tabel 3.1 Diferențele dintre testarea manuală și cea automată*

Parametrii	Testarea Automată	Testarea Manuală
Timp	Este foarte rapidă	Necesită mult timp și un număr mare de resurse umane
Investiție	Implică investiții în programe de automatizare și în oameni calificați	Implică resurse umane
Eficiența costurilor	Nu este rentabilă pentru un volum mic	Nu este rentabilă pentru un volum mare
Fiabilitate	Este o metodă sigură fiind realizată prin intermediul programelor	Nu este foarte precisă din cauza posibilelor erori umane
Testarea experimentală	Testarea aleatorie nu este permisă	Se poate realiza testare aleatorie



Tabel 3.1 Diferențele dintre testarea manuală și cea automată

Parametrii	Testarea Automată	Testarea Manuală
Modificarea interfeței	Orice mică modificare în interfața aplicației poate afecta execuția testelor	Modificările ulterioare în interfața aplicației nu influențează testarea manuală
Cunoștințe de programare	Necesită cunoștințe de programare	Nu este nevoie de cunoștințe de programare
Raport	Generarea raportului se poate face simplu, după execuția testelor cu ajutorul programelor	Raportul necesită mai mult timp pentru realizarea lui și adesea nu este atât de intuitiv
Executarea paralelă	Testele pot fi executate în paralel pe diferite platforme reducând timpul execuției	Testarea în paralel se poate realiza manual, dar necesită un număr mare de persoane

Pe lângă faptul că testarea automată poate reduce semnificativ numărul de defecte care ajung în produsul finit, aceasta aduce totodată o sumedenie de alte avantaje:

- **Viteză ridicată** este unul dintre cele mai mari avantaje ale testării automate deoarece detectarea defectelor din timp reduce semnificativ riscurile
- **Evaluarea obiectivă** prin prisma faptului că testele sunt create pe baza cerințelor clientului la începutul proiectului, fără a mai necesita intervenția omului pe parcursul proiectului
- **Volum mare de date** care poate fi executat într-un timp redus
- **Testarea de regresie** deoarece oricât de mică ar fi o modificare nou introdusă în aplicație, sistemul poate fi testat integral odată cu rularea tuturor testelor
- **Acces ușor** la o statistică cu privire la calitatea produsului, prin rapoarte generate automat în urma rulării, cuprinzând date referitoare la progresul testului [12]

Cu toate că testarea automată aduce foarte multe beneficii în cadrul unui proiect asta nu înseamnă garantarea succesului deoarece fiecare program nou introdus într-o organizație necesită eforturi pentru obținerea beneficiilor de lungă durată.

Odată cu introducerea testării automate pe un proiect, pot apărea și unele riscuri precum subevaluarea costurilor și eforturilor pentru introducerea unui nou instrument care ajută la automatizarea testelor, estimarea greșită a timpului necesar pentru instruirea personalului sau efortul necesar pentru mentenanța testelor.

Procesele software au devenit din ce în ce mai utilizate cam în marea majoritate a domeniilor, din această cauză testarea automată a devenit tot mai populară în cadrul companiilor, marele avantaj fiind economisirea timpului și a banilor. Pentru a putea fi realizată, aceasta necesită diferite programe automatizate.

## 3.2 Selenium WebDriver

Selenium este unul dintre cele mai cunoscute *framework*-uri de testare din lume, reprezintă un set cu diferite instrumente *software*, fiecare având o abordare diferită care vine în sprijinul automatizării testelor. Este ideal pentru scrierea testelor funcționale și rularea acestora pe o mulțime de *browsere*. Selenium se poate folosi pe diferite platforme, precum *Windows*, *Linux* și *macOS* [13].

### 3.2.1 Introducere

Selenium a luat naștere în 2004 când *Jason Huggins* testa o aplicație și și-a dat seama că ar putea reduce semnificativ timpul dacă ar putea să nu testeze aplicația manual or de câte ori o schimbare avea loc în sistem. El a dezvoltat o bibliotecă *Javascript* care ar putea realiza interacțiunile cu pagina, permițându-i să reia automat testele pe diferite *browsere*. Această bibliotecă a devenit în cele din urmă *Selenium Core*, care stă la baza tuturor funcționalităților *Selenium Remote Control (RC)* și *Selenium IDE*.

În 2006, un inginer de la *Google* a început să lucreze la un proiect numit *WebDriver*, proiect apărut pentru a elimina limitările de *Javascript* și de *browser* prin implementarea unei interacțiuni de pagină cu ajutorul unui *API* orientat pe obiecte. Deoarece cele două proiecte își completau nevoile unul altuia, *Selenium* și *WebDriver* s-au unit formând astăzi *Selenium 2.0* [13].

### 3.2.2 Suita Selenium

Selenium cuprinde o suită cu patru programe precum cele din figura 3.1:

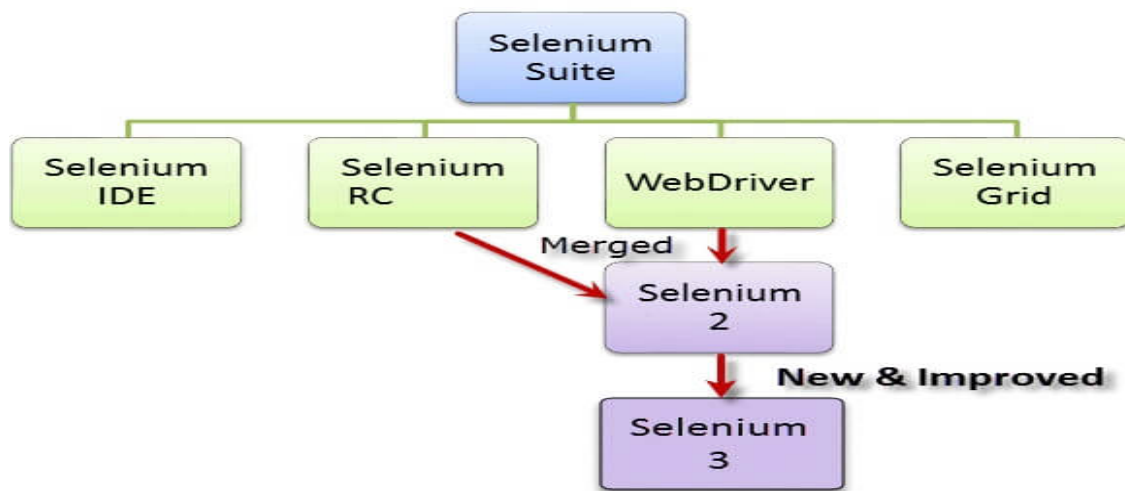


Figura 3.1. Suita Selenium [14]

### 1. *Selenium IDE(Integrated Development Environment)*

Este un program prototip utilizat pentru crearea scripturilor. Acesta prezintă o interfață ușor de utilizat pentru realizarea testelor automate. Cea mai interesantă caracteristică a programului Selenium IDE este opțiunea de a înregistra acțiunile utilizatorului în timp ce acestea sunt executate, ca mai apoi să le transcrie sub formă de script, în una din numeroasele limbaje de programare pe care le conține [13].

### 2. *Selenium RC(Remote Control)*

Știut și sub denumirea de *Selenium 1*, a fost pentru mult timp principalul proiect înainte ca *Selenium 1* și *Selenium WebDriver* să se unească. În prezent *Selenium RC* este depreciat și nu mai are parte de o continuă dezvoltare sau întreținere [13].

### 3. *Selenium WebDriver*

Știut sub numele de Selenium 2, care din 2016 are o nouă versiune, Selenium 3. Este proiectul de bază din toată suita Selenium, cuprinzând o mulțime de noi caracteristici precum un *API(eng. Application Programming Interface)* mai coerent și orientat spre obiecte. Spre deosebire de *Selenium 1*, unde serverul *Selenium* era necesar pentru a rula testele, *Selenium WebDriver* nu are nevoie de un server special pentru a executa teste. În schimb, *WebDriver* pornește direct o instanță a *browserului* și o controlează. *Selenium WebDriver* suportă mai multe limbaje de programare: *Python*, *Ruby*, *Java*, *C#*, etc. [13].

### 4. *Selenium Grid*

Acesta vine în ajutorul programului *Selenium RC*, cu o soluție pentru rularea testelor la o scală mai largă și pentru rularea pe diferite medii de operare. Un mare avantaj al programului *Selenium Grid* este de a putea rula testele în paralel pe mai multe mașini de la distanță. În fiecare caz, *Selenium Grid* îmbunătățește foarte mult timpul necesar pentru a rula suita de teste, utilizând procesarea paralelă.

De curând s-a lansat o nouă versiune *Selenium 4 Alpha*, în Aprilie 2019. Această nu este încă o versiune oficială, data exactă a lansării neștiindu-se încă. O schimbare majoră ar fi schimbarea protocolului *JSON* cu protocolul *W3C* [13].

#### 3.23 Comenzi utile din Selenium WebDriver

Acest framework este foarte utilizat datorită ușurinței prin care poți comunica cu elementele unei aplicații prin intermediul unor comenzi. Cele mai comune și utilizate funcții din *Selenium WebDriver* scrise în limbajul de programare *Python* sunt prezentate în continuare [15]:

Pentru a putea crea o instanță de *browser*, primul lucru pe care trebuie să îl facem este de a importa librăria *webdriver*. Iar prin comanda prezentată mai jos se va crea o instanță a *browserului Chrome*. Se pot alege și alte opțiuni de *browser* precum: *Firefox*, *Internet Explorer*, *Opera* sau *Safari*.

```
1. from selenium import webdriver
2.
3. driver = webdriver.Chrome()
```

Pentru a putea termina această instanță, există două metode „*close()*” și „*quit()*”. Prima metodă va închide doar un singur *tab*, dar dacă acesta este singurul *tab* deschis, implicit *browserul* se va închide și el. Cea de a doua metodă va termina instanța de browser și îl va închide indiferent de câte *tab*-uri sunt deschise. Metodele prezentate mai sus, sunt ilustrate în secvențele următoare:

```
1. driver = webdriver.Chrome()
2. driver.quit() # driver.close()
```

Primul pas pentru a putea testa o aplicație *web*, este de a naviga către pagina site-ului respectiv. Pentru a naviga putem folosi comanda „*get*”, care are ca parametru adresa *URL* la care vrem să ajungem. *WebDriver*-ul v-a aștepta până când pagina este încărcată complet.

```
1. driver.get("http://www.google.com")
```

### 3.24 Metodele de identificare a elementelor

Locatorii reprezintă așa zisele comenzi de care *Selenium* are nevoie pentru a identifica elemente precum căsuțe, butoane, text etc. dintr-o pagină *web*. Locatorii reprezintă o parte foarte importantă din automatizarea testelor. Chiar dacă pot părea ușor de identificat, dacă acești locatori nu sunt bine aleși de la început, acest lucru poate duce la interpretarea greșită a rezultatelor.

De aceea *Selenium* conține mai multe tipuri de locatori, care se pot folosi, în funcție de cum au fost elementele declarate. Tipurile de locatori existenți în *Selenium* sunt ilustrați în figura 3.2:

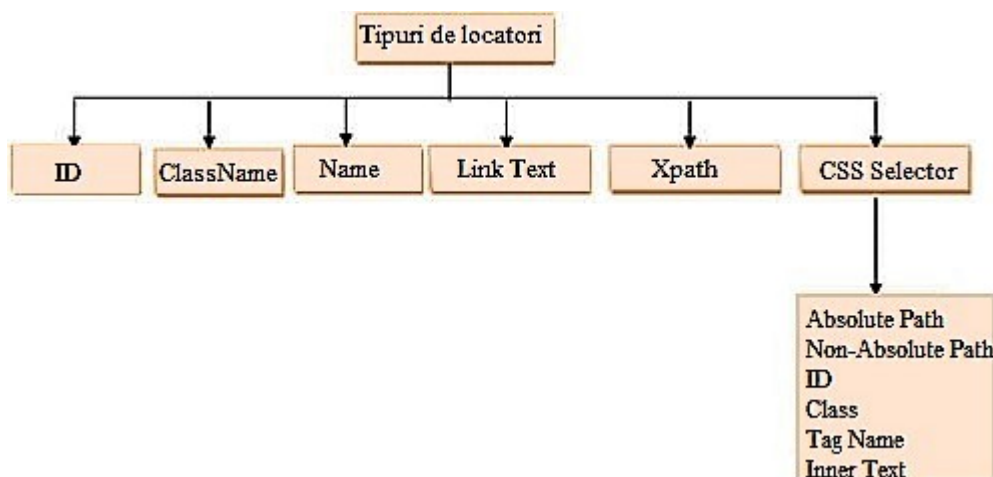


Figura 3.2 Tipuri de locatori din Selenium [16]

Funcțiile utilizate care ajută la găsirea elementelor din pagină sunt „*find\_element*”/ „*find\_elements*”. Aceste metode sunt private, însă librăria cuprinde și metodele publice ca de exemplu „*find\_element\_by\_id*”, „*find\_element\_by\_name*”, „*find\_element\_by\_class*” etc.

Există mai multe metode prin care putem găsi un element, depinde foarte mult de cum a fost declarat acesta de către *developer*. În continuare se vor prezenta tipurile de locatori, utilizați cu ajutorul funcției „*find\_element()*”, care are ca parametru obiectul de interogare „*by*”, cu ajutorul căruia putem alege tipul locatorului. Cel de al doilea parametru este locatorul identificat pentru elementul ales [16].

Primul lucru necesar pentru a putea folosi obiectul de interogare „*by*” în metodele prezentate mai sus este de a importa acest obiect din librăria „*common.by*”:

```
1. from selenium.webdriver.common.by import By
```

## 1. By ID

Dacă cunoaștem *id*-ul unui element putem folosi această metodă, care returnează primul element găsit cu *id* dat ca parametru. Dacă nici un element nu a fost găsit cu *id*-ul dorit, se va returna un mesaj de eroare.

Această metodă de identificare este foarte utilă atunci când elementele au *id*-uri unice. De exemplu dacă vrem să identificăm o casuță pentru *login* care are codul sursă:

```
1. <html>
2. <body>
3. <form id="loginForm">
```

Metoda pentru localizarea elementului va fi apelată ca în secvența de mai jos:

```
1. login_form = driver.find_element(By.ID, 'loginForm')
```

## 2. By Name

Această metodă este foarte asemănătoare cu cea anterioară, singura diferență este căutarea elementelor după nume și nu după *id*. Dacă vrem să localizăm un câmp pentru completarea unei adrese de email, care are codul sursă:

```
1. <html>
2. <body>
3. <form id="loginForm">
4. <input name="username" type="text" />
```

Metoda pentru localizare elementului va arăta ca în secvența de mai jos:

```
1. username = driver.find_element(By.NAME, 'username')
```

### 3. By Class Name

Pentru a identifica un element după această metodă, este necesar să cunoaștem numele clasei. Ca în cazurile prezentate mai devreme, dacă s-au găsit mai multe elemente cu același nume de clasă, primul găsit va fi returnat, iar în cazul în care nici un element nu a fost găsit, se va returna excepția „*NoSuchElementException*”. Mai jos este prezentat codul sursă pentru un element cu numele clasei „*className*”:

```
1. <html>
2. <body>
3.   <p class="className">...</p>
```

Metoda pentru localizare elementului în funcție de numele clasei este prezentată în secțiunea de mai jos:

```
1. className= driver.find_element(By.CLASS_NAME 'className')
```

### 4. By Tag Name

Această metodă este folosită în cazul în care nu putem identifica elementul dorit după nici un alt locator, precum „*id*”, „*name*”, „*class name*” sau în cazul în care vrem să localizăm un element anume dintr-un tabel.

De cele mai multe ori această identificare este evitată din cauză ca foarte mult elemente din aceeași pagină au *tag name-ul* identic și există posibilitatea mare ca elementul dorit să nu fie găsit. Mai jos este prezentat codul sursă al unui element ce reprezintă titlul unei secțiuni:

```
1. <html>
2. <body>
3.   <h1>Title</h1>
```

Metoda pentru localizare elementului „*Title*” este prezentată în secvența de mai jos:

```
1. title= driver.find_element(By.TAG_NAME, 'h1')
```

### 5. By Link Text

Se va folosi această metodă când dorim să identificăm *hyperlink*-urile dintr-o pagină *web*. Putem utiliza metoda de identificare doar în cazul în care cunoaștem textul *link*-ului. Este utilă atunci când se testează navigarea dintr-o pagină în alta. Mai jos este prezentat codul sursă a unui *hyperlink* dintr-o pagină, prin metoda *Link Text*:

```
1. <html>
2. <body>
3.   <a href="cancel.html">Cancel</a>
```

Locatorul realizat cu ajutorul *Link Text*-ului poate conține atât textul complet sau doar parțial. Apelarea funcției se poate observa în secvența de mai jos :

```
1. link = driver.find_element(By.LINK_TEXT, 'Cancel')
2. partial_link = driver.find_element(By.PARTIAL_LINK_TEXT, 'Can')
```

## 6. By Xpath

*Xpath* este un limbaj folosit pentru localizarea pozițiilor pe baza conținutului dintr-un document *XML* (*eng. Extensible Markup Language*). Acest limbaj oferă posibilitatea de a naviga prin nodurile documentului *XML* pentru a satisface criteriile impuse, acesta fiind reprezentat sub forma unui arbore.

Pentru a putea identifica un element dintr-o pagină *web* putem folosi două opțiuni [17]:

- *Absolute Xpath* – acest tip de locator reprezintă o cale care pornește chiar de la rădăcină, pornește de la părinte până la elementul dorit. Acest locator începe întotdeauna cu simbolul „/”.

Avantajele acestei metode de identificare a elementelor sunt viteza cu care sunt găsite acestea și faptul că fiecare element are o cale unică. Un mare dezavantaj în schimb, este că la cea mai mică schimbare în aplicație, structura arborului va fi diferită făcând ca traseul să nu fie posibil.

Codul sursă al elementului dintr-un HTML este prezentat în secvența de mai jos:

```
1. <html>
2. <body>
3. <form id="loginForm">
```

Un exemplu de identificare a elementului printr-un Xpath absolut este prezentat în secvența de mai jos:

```
1. loginForm=driver.find_element(By.XPATH, "/html/body/form[1]")
```

- *Relative Xpath* – pentru acest tip de locator, calea nu începe de la rădăcină, ci poate să înceapă de oriunde din interiorul structurii documentului, fiind o cale parțială și nu una absolută. Acest locator începe cu simbolul „//” care semnifică căutarea elementului oriunde în pagină.

Avantajul acestei metode este posibilitatea folosirii diferitelor atribute pentru identificarea elementelor și faptul că nu este așa de lung ca un locator care folosește *Xpath* absolut. Un dezavantaj este viteza cu care este găsit elementul, fiind mai lentă ca la *Xpath*-ul relativ.

În secvența de mai jos sunt prezentate două metode diferite de identificare pentru elementul prezentat mai sus, folosind *Relative Xpath*:

```
1. login_form = driver.find_element(By.XPATH, "//form[1]")
2. login_form = driver.find_element(By.XPATH, "//form[@id='loginForm']")
```

Metoda de identificare a elementelor prin *Xpath* este mai complexă și mai sigură. Probabilitatea ca elementul dorit să fie găsit este foarte mare, de aceea această metodă este aleasă în cele mai multe cazuri, fiind printre cele mai sigure.

## 7. By Css Selector

Un selector *CSS* este o combinație între un selector al unui element și o valoare, care împreună identifică elementul dintr-o pagină *web*. Ele sunt șiruri de caractere care reprezintă *tag-uri*, *id-uri*, attribute și clase *HTML*. Această metodă poate să nu funcționeze pe diferite *browsere*, versiunea acestora fiind creată diferit.

Există diferite utilizări ale acestei metode, în funcție de attributele elementului după care acesta trebuie identificat. Câteva dintre aceste utilizări sunt prezentate mai jos:

```
1. <html>
2. <body>
3.   <form id="loginForm">
4.     <div class="className">
5.       <input id="user">
```

- După *Absolute Path* – locatorul creat după această metodă conține simbolul „>” pentru a naviga de la părinte și până la elementul dorit. Mai jos este prezentat un exemplu de cum am putea scrie un locator după *Absolute Path* cu *CSS Selector*:

```
1. user=driver.find_element(By.CSS_SELECTOR, "form>div>input")
```

- După *Non-Absolute Path* – Dacă vrem să identificăm spre exemplu un element care este copilul altui element, atunci pentru cazul de mai sus, nu vom mai folosi simbolul „>” ci un spațiu alb:

```
1. user=driver.find_element(By.CSS_SELECTOR, "form div input")
```

- După *Id* – Metoda aceasta este bine să fie folosită atunci când avem *Id-uri* unice pentru fiecare element. Simbolul folosit pentru localizarea elementului după *Id* este „#”:

```
1. user=driver.find_element(By.CSS_SELECTOR, "#input")
```

- După *Class* – Pentru identificarea elementelor după clasă se va folosi simbolul „.”:

```
1. className=driver.find_element(By.CSS_SELECTOR, ".className")
```



- După *Tag Name* – Este folosită în special în combinație cu alte atribute precum clasa sau *id*-ul pentru a diferenția elementele cu *Tag Name* identic. Pentru a identifica un element după CSS Selector folosind Tag Name nu este nevoie de nici un simbol anume și poate fi folosit ca în exemplul de mai jos:

```
1. <html>
2. <body>
3. <h1>Welcome</h1>
```

Apelarea metodei se va realiza precum în secvența de mai jos:

```
1. tagName=driver.find_element(By.CSS_SELECTOR,"h1")
```

Dacă vrem să combinăm mai multe atribute, apelarea metodei se va face ca în exemplul de mai jos:

```
1. <html>
2. <body>
3. <form id="loginForm"></form>
```

Metoda apelată va arăta ca în secvența de mai jos:

```
1. login_form = driver.find_element(By.CSS_SELECTOR,"form[id='loginF
orm']")
```

- După *Inner Text* – Se poate utiliza atunci când vrem să localizăm un element după un text spre exemplu. Această metodă folosește pseudo-clasa „*contains()*”. Dacă vrem să localizăm titlul unei pagini după text, unde codul sursă arată ca în exemplu de mai jos:

```
1.<html>
2.<body>
3.<h1>Tesing</h1>
```

Locatorul creat după combinația între *Tag Name* și *ID*:

```
1. title = driver.find_element(By.CSS_SELECTOR,'h1:contains("Testing
")')
```

Cu toate că aceste funcții sunt foarte utile în identificarea elementelor dintr-o pagină, există cazuri când spre exemplu, *Id*-ul unui element se schimbă la fiecare rulare a unui test. Acești locatori sunt cunoscuți sub numele de locatori dinamici.

Pentru a rezolva această problemă putem folosi funcția „*contains()*” prezentată mai sus, sau putem folosi funcțiile de identificare a atributelor unui element. Spre exemplu

funcția „*getAttribute*” care poate identifica orice tip de atribut fie el *id*, numele clasei sau un *tag name*. Astfel, dacă vrem să identificăm *id*-ul unui element, chiar dacă acesta se schimbă la fiecare încărcare a paginii, îl putem obține prin intermediul acestei funcții [16].

### 3.25 Interacțiunea cu elementele web

Localizarea elementelor este foarte importantă în testarea unei aplicații *web*, însă pentru a putea reproduce acțiunile unui utilizator avem nevoie să interacționăm cu aceste elemente, după ce ele au fost identificate.

O acțiune des întâlnită este aceea de a completa un câmp, spre exemplu o adresă de e-mail. Pentru a putea realiza acest lucru putem folosi funcția prezentată mai jos [15]:

```
1. element.send_keys("e-mail")
```

Înainte de a folosi această funcție trebuie să ne asigurăm că, câmpul este gol. Ștergerea conținutului se poate realiza utilizând secțiunea de mai jos:

```
1. element.clear()
```

Pentru apăsarea butoanelor din pagină se va folosi funcția:

```
1. element.click()
```

Pentru acțiunile de „*drag and drop*”, pentru mutarea elementelor una peste alta sau pentru mutarea acestora într-un loc anume există o clasă specială „*ActionChains*”:

```
1. from selenium.webdriver import ActionChains
2.
3. action_chains = ActionChains(driver)
4. action_chains.drag_and_drop(element1, element2).perform()
```

Aplicațiile web devin tot mai moderne, odată cu asta și structura lor devine tot mai complicată, având multe ferestre și cadre. Pentru a putea interacționa cu un element dintr-o anumită fereastră este nevoie să ne mutăm dintr-un cadru în altul. Ca să putem realiza acest lucru putem folosi sintaxa de mai jos:

```
1. driver.switch_to_window("windowName")
```

Aplicațiile *web* moderne folosesc o tehnică numită *AJAX*(*Asynchronous JavaScript and XML*) folosită pentru crearea paginilor *web* interactive. Atunci când o pagină se încarcă, unele elemente din aceasta s-ar putea să fie prezente la un interval de timp diferit unele de altele. Pentru a evita o eroare cum că elementul nu este prezent, se folosesc funcții de așteptare. Există două tipuri de funcții de așteptare: implicită și explicită, prezentate mai jos:

- *Implicit Wait*

Acest tip de așteptare este folosit atunci când cunoaștem timpul necesar pentru încărcarea unui element în pagină, sau știm că acesta nu este disponibil imediat. Această metodă nu este foarte recomandată deoarece timpul setat în test, poate să difere de la o rulare la alta.

Funcția se numește „*implicitly\_wait()*” și are ca parametru timpul dorit de așteptare, exprimat în secunde. Dacă nu este setat ca parametru nici un număr, această funcție va lua o valoare implicită de așteptare zero. Mai jos este prezentat un exemplu în care este utilizată metoda:

```
1. from selenium import webdriver
2.
3. driver = webdriver.Chrome()
4. driver.implicitly_wait(5)
```

- *Explicit Wait*

Metoda aceasta este cel mai des utilizată deoarece nu ne restricționează sa cunoaștem un timp exact în care elementul va fi disponibil. Dacă spre exemplu știm că un element se încarcă destul de greu în pagină, și trebuie să interacționăm cu acesta, putem folosi clasa „*WebDriverWait*” în combinație cu condițiile puse la dispoziție din clasa *EC(Expected Condition)*.

În exemplul de mai jos se prezintă o modalitate de așteptare după un element, până prezența acestuia în pagină este localizată într-un interval de zece secunde. Dacă elementul nu a fost localizat în acest interval de timp, eroarea „*TimeoutException*” va fi returnată.

```
1. from selenium import webdriver
2. from selenium.webdriver.common.by import By
3. from selenium.webdriver.support.ui import WebDriverWait
4. from selenium.webdriver.support import expected_conditions as EC
5.
6. driver = webdriver.Chrome()
7. element = WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.ID, "dynamicElement")))
```

Clasa *EC* conține o multitudine de condiții care pot fi folosite în funcție de nevoile noastre. Câteva dintre aceste sunt prezentate mai jos:

- |  |  |
|--|--|
| • <i>title_contains</i>                | • <i>presence_of_element_located</i>   |
| • <i>visibility_of_element_located</i> | • <i>text_to_be_present_in_element</i> |

- *visibility\_of*
- *element\_to\_be\_selected*
- *invisibility\_of\_element\_located*
- *staleness\_of*
- *element\_to\_be\_clickable*
- *element\_selection\_state\_to\_be*
- *presence\_of\_all\_elements\_located*
- *alert\_is\_present*

*WebDriverWait* va verifica odată la 500 de milisecunde condiția, în intervalul de timp impus. Dacă nu ne sunt de nici un ajutor condițiile existente din clasa *EC*, putem sa folosim o condiție personalizată [15].

Un alt aspect important în testarea *web* o reprezintă *cookie*-urile. Acestea permit strângerea de informații despre acțiunile utilizatorului dintr-o aplicație. Informațiile adunate constau în păstrarea datelor de autentificare, paginilor vizitate sau păstrarea produselor dintr-un coș de cumpărături chiar dacă utilizatorul s-a *delogat* din contul lui.

Scopul acestor *cookie*-uri este acela de a ușura utilizarea aplicației de către utilizator. Însă pentru testarea automată aceste informații păstrate pot duce la teste picate, cu un rezultat care nu indică un defect în aplicație. De aceea este necesar ca înainte de fiecare rulare, aceste *cookie*-uri să fie eliminate [15].

Mai jos este prezentată sintaxa necesară pentru ca toate *cookie*-urile din instanța de *browser* utilizată sa fie șterse:

```
1. driver = webdriver.Chrome()  
2. driver.delete_all_cookies()
```

### 3.26 Proiectarea testelor automate

După ce a fost evaluată viabilitatea proiectul de testare automată ținând cont de scopurile, riscurile, timpul și resursele financiare, iar acesta a fost adoptat pentru a testa aplicația *web*, s-a luat în considerare folosirea unui program special pentru testare, precum *Selenium*. Următorul pas este de a alege o structură pentru scrierea scripturilor automate.

Pentru un proiect de dimensiuni mai mari avem nevoie de o structură bine definită. Cu cât o aplicație este mai complexă cu atât și numărul testelor automate va crește. Pentru ca totul să fie cât mai bine organizat avem nevoie să împărțim testele în suite pe categorii, în funcție de specificațiile fiecăruia. O suită de teste reprezintă o grupare a cazurilor de testare similare.

Pentru a putea verifica funcționalitățile sistemului prin intermediul *UI(User Interface)* trebuie să reproducem acțiunile unui utilizator. O abordare tot mai populară este modelul *Page Object Model(POM)* deoarece ajută la întreținerea testelor.

## Page Object Model

Acest model constă în separarea testelor propriu-zise de definirea funcționalităților și a elementelor din aplicație. În cadrul acestui model fiecare pagină a unei aplicații va reprezenta o clasă, iar fiecare acțiune/funcționalitate va fi reprezentată printr-o metodă.

Numele clasei trebuie să fie sugestiv, de exemplu dacă pagina din aplicație este pagina principală, numele clasei ar putea fi „*Home* “. La fel se întâmplă și în cazul funcționalităților, dacă avem un buton de autentificare, numele metodei trebuie să reprezinte acțiunea pe care o execută, de exemplu „*click\_log\_in\_button* “ [18].

Beneficiile acestui model sunt:

- Fiabilitate
- Acoperirea din punct de vedere al testării va fi mai mare deoarece putem să vedem ușor din structura codului ce părți ale aplicației nu au fost testate
- Structura proiectului este ușor de înțeles
- Crearea codului reutilizabil care poate fi apoi împărțit cu restul testelor fără a fi nevoie ca acesta să fie scris în mai multe părți
- Reducerea cantității de cod duplicat
- Există un singur depozit pentru serviciile sau operațiunile disponibile din aplicație, în loc să fie împărțite în teste
- Schimbările din interfața aplicației nu au un impact atât de mare asupra testelor, deoarece necesită schimbări într-un singur loc [18]

Unul din marile avantaje al utilizării *Page Object Model* este ușurința cu care este realizată mentenanța testelor. Utilizarea obiectelor de pagină permite, de asemenea, o întreținere ușoară. Deoarece codul de testare este acum reutilizabil și încapsulat în cadrul metodelor și claselor, acest lucru ușurează întreținerea.

Această abordare creează, de asemenea, teste mai ușor de citit și de înțeles, testele funcționale automate pot spune exact cum trebuie să se comporte aplicația.

Pe lângă toate aceste beneficii ale modelului *Page Object*, acesta vine și cu câteva dezavantaje precum timpul și efortul investit în realizarea modelului, mai ales pe un proiect care conține foarte multe pagini și funcționalități.

În figura de mai jos este prezentată arhitectura modelului *Page Object* implementată pe o aplicație *web*:

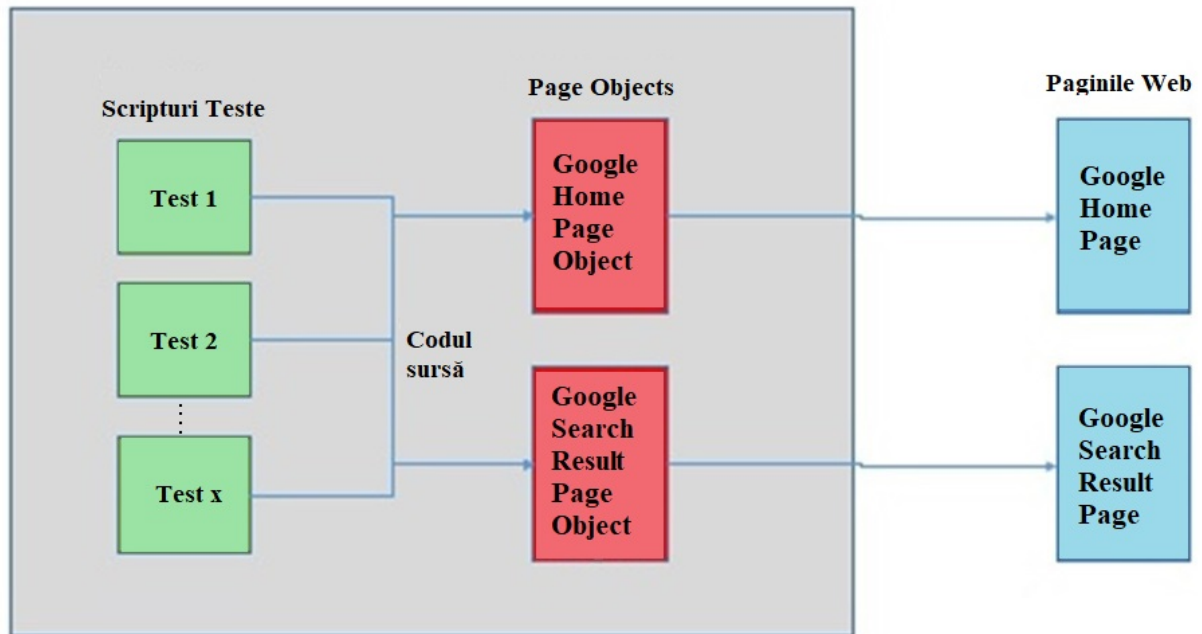


Figura 3.3 Arhitectura modelului Page Object [19]

După ce structura proiectului a fost aleasă, se poate trece la implementarea codului propriu-zis pe baza modelului *Page Object*. Ulterior se mai pot adăuga diferite metode care ajută la o mai bună înțelegere despre cum este implementată testarea automată pe un proiect. În acest mod putem implica toate persoanele din proiect, chiar dacă acestea nu sunt familiarizate cu limbajele de programare.

Una dintre metodele des întâlnite în multe companii care ajută la o mai bună înțelegere a proiectului de către *bussnis analyst* este limbajul *Gherkin* care este prezentat în subcapitolul 3.3.

### 3.3 Gherkin

Este un limbaj folosit în procesul de dezvoltare a software-ului agil *BDD*(eng. *Behavior-Driven Development*), proces care a luat naștere din dorința de a încuraja colaborarea dintre *developeri*, *testeri* cât și persoanele din *bussnis* care nu au cunoștințe tehnice. Această soluție încurajează echipele să folosească exemple concrete pentru a adopta o viziune comună a modului în care trebuie să se comporte aplicația [20].

#### 3.31 Introducere în BDD

*BDD* a fost înființat ca o extensie a *TDD*( *Test-Driven Development*) care la rândul ei folosea *DSL*, limbaje simple de scris specifice domeniului.

Deși *BDD* este în principal o idee despre modul în care dezvoltarea de *software* ar trebui să fie gestionată atât din punct de vedere al afacerii, cât și de cunoștințe tehnice, practica *BDD* presupune utilizarea unor instrumente software. Cele mai multe proiecte

software implică echipe constituite di mai mulți oameni care lucrează în colaborare împreună, astfel încât comunicarea de înaltă calitate este esențială pentru succesul proiectului. O bună comunicare nu este doar despre descrierea elocventă a ideilor unei singure persoane. Persoanele trebuie să solicite păreri constant pentru a se asigura că au fost înțelese corect. [21].

O caracteristică importantă a acestui proces este trecerea de la gândirea de „teste“ la gândirea de „comportament“. Cu BDD, toate testele sunt teste de acceptare a clienților, scrise într-un limbaj simplu(uman), astfel încât părțile non-tehnice interesate să le poată înțelege.

### **Practici BDD**

- Stabilirea obiectivelor din partea mai multor părți interesate necesare pentru implementarea unei viziuni
- Folosirea exemplelor pentru a descrie comportamentul aplicației sau a secțiunilor de cod utilizate
- Automatizarea acestor exemple pentru testele de regresie
- Utilizarea cuvântului „*should*“ atunci când este descris comportamentul *software*-ului pentru a ajuta la clarificarea responsabilității și pentru a permite interogarea funcționalității *software*-ului
- Utilizarea cuvântului „*ensure*“ atunci când se descriu responsabilitățile *software*-ului pentru a diferenția rezultatele din domeniul de aplicare al codului față de efectele secundare ale altor elemente [20]

De a lungul timpului s-au dezvoltat mai multe *framework*-uri bazate pe procesul BDD precum *Cucumber*, *Behave*, *Lettuce*, *SpecFlow*, etc.

Limbajul *Gherkin* poate fi înțeles printr-un instrument de automatizare numit "*Behave*". Asta înseamnă că programul poate interpreta limbajul *Gherkin* și îl poate folosi pentru a efectua teste automate.

*Behave* este un *framework* de testare, bazat pe procesul de dezvoltare a *software*-ului agil BDD, folosit împreună cu limbajul de programare *Python*. Acesta a fost creat după *framework*-ului *Cucumber* care este folosit împreună cu *Java*.

### **3.32 Limbajul Gherkin**

Limbajul *Gherkin* este un limbaj natural, care folosește o sintaxă simplă foarte asemănătoare cu limba engleză, dar într-un mod mai structurat. Pe lângă limba engleză, acesta suportă încă 74 de alte limbi.

Oricine poate înțelege funcționarea testului scris în *Gherkin* și ce intenționează să facă acesta. *Gherkin* oferă un impact puternic neașteptat, permițând oamenilor să vizualizeze sistemul înainte de a fi construit. Oricare dintre oamenii non-tehnici care se ocupă de

partea de *bussnis* ar citi testele scrise în acest limbaj ar putea să îl înțeleagă, și să își expună punctul de vedere. Cu ajutorul acestui proces, ei pot spune dacă testul reflectă înțelegerea lor cu privire la ceea ce ar trebui să facă sistemul sau pot duce chiar la gândirea altor scenarii care trebuie luate în considerare [20].

Testele scrise cu ajutorul limbajului *Gherkin* elimină multe neînțelegeri, cu mult înainte ca ambiguitățile din cod să apară.

### **Sintaxa *Gherkin***

*Gherkin* folosește un set de cuvinte cheie speciale pentru organizarea structurii cât și pentru o înțelegere cât mai bună a specificațiilor.

Un test scris în *Gherkin* este format din mai mulți pași numiți *steps*. Mai jos sunt prezentate cele zece cuvinte cheie existente în limbajul *Gherkin* [20]:

- *Given*
- *When*
- *Then*
- *And*
- *But*
- *Scenario*
- *Feature*
- *Background*
- *Scenario Outline*
- *Examples*

#### **1. *Feature***

Scopul cuvântului cheie „*Feature*” este de a oferi o descriere la un nivel înalt a unei funcții *software* și de a grupa scenariile care fac parte din aceeași categorie în funcție de funcționalitatea testată.

Primul lucru pe care trebuie să îl facem când vrem să scriem un test folosind sintaxa *Gherkin* este de a începe cu, cuvântul cheie *Feature*, urmat de simbolul „:” și un text scurt care descrie scopul testului.

Mai jos este prezentată o secvență ce conține cuvântul cheie *Feature* care descrie pe scurt funcționalitatea testului, aceea de căutare în pagina *Google* :

**Feature:** Google Searching

#### **2. *Scenario***

Acest cuvânt cheie este folosit pentru a descrie scenariul care urmează a fi creat și pentru a-i da un titlu acestuia. Deoarece un *Feature* poate avea mai multe scenarii, acordarea fiecărui scenariu câte un titlu ajută la o mai bună înțelegere a ceea ce este testat fără a trebui ca toți pașii să fie citați. De aceea titlul scenariului ar trebui să fie succint.

Un exemplu de cum ar trebui să arate un scenariu este prezentat mai jos:

**Feature:** Google Searching

**Scenario:** Searching text

#### **3. *Background***

Aceasta stabilește contextul pentru toate scenariile din *feature*-ul respectiv.



Dacă se observă că mai multe scenarii au date comune, sau pași care sunt identici, și necesită să fie executați în aceeași ordine, pași aceștia pot fi trecuți în *Background* pentru a elimina repetarea codului.

Pașii declarați în *Background* vor fi rulați înainte de fiecare scenariu. Scenariile pot avea în continuare alți pași declarați, dar se vor executa numai după ce ultimul pas din *Background* a fost rulat.

În secvența de mai jos este prezentată o utilizare a acestui cuvânt cheie:

**Feature:** Google Searching

**Background:**

Given I go to Google page

**Scenario:** Searching text

#### 4. Given

Sunt pași care descriu contextul inițial al sistemului, înainte ca acțiunile care reprezintă interacțiunea cu sistemul să se întâmple, adică înainte ca pașii din *When* să fie executați.

Exemple de pre-condiții, care s-ar putea scrie:

Given I am logged in

Given User has 34\$ in account

#### 5. When

Acest pas descrie interacțiunea propriu-zisă cu sistemul, acțiunile utilizatorului asupra lui. Această acțiune trebuie să ducă întotdeauna la un rezultat. Este bine ca într-un scenariu să nu fie mai mult de un pas definit cu, cuvântul cheie *When*.

Exemple de acțiuni posibile:

When I add 7 products to the cart

When I create a new account

#### 6. Then

Acest pas trebuie să reprezinte o validare a sistemului, reflectând comportamentul acestuia, ce ar trebui să se întâmple după ce o acțiune a utilizatorului a fost realizată. Este o comparație între pasul definit înainte (*When*) și pasul definit în sintaxa *Then*.

În secvența de mai jos este prezentată o utilizare a acestui cuvânt cheie, în concordanță cu acțiunile prezentate anterior:

Then Cart should contain 7 products

Then I verify if the account was created

#### 7. And

Este un cuvânt de legătură care elimină duplicarea cuvintelor cheie sau simplifică propozițiile prea lungi dintr-un anumit pas. În exemplul de jos este prezentată folosirea acestuia:

Given I load the website  
Given I go to Sing In page      →      Given I load the website  
And I go to Sing In page

### 8. *But*

La fel ca și în cazul cuvântului cheie *And*, prezentat mai sus, este un cuvânt de legătură care ajută la eliminare unor cuvinte cheie care se repetă. Mai jos este prezentat cum poate fi folosit:

**Then I should see First Name** → **Then I should see First Name**  
**Then I shouldn't see Second Name** **But I shouldn't see Second Name**

### 9. *Examples*

Acestea prezintă exemple concrete pentru datele de intrare cât și pentru datele de ieșire. Aceste exemple sunt definite pentru a înțelege mai bine cum ar trebui să funcționeze anumite acțiuni ale utilizatorului, sau sistemul propriu-zis în situații concrete.

Mai jos este prezentat cum este definit un exemplu în limbajul *Gherkin*:

**Examples:**

Name	OriginalBalance	NewBalance
Eric	100	55
Gaurav	100	60
Ed	1000	800

### 10. *Scenario Outline*

Este utilizat pentru rularea același scenariu de mai multe ori, însă cu diferite date de intrare. Valorile sunt declarate cu ajutorul cuvântului cheie *Examples*.

## 4 Implementarea soluției

### 4.1 Descrierea aplicației

Partea practică constă în realizarea unui *framework* de testare automată pe baza modelului *Page Object*, realizat cu *Selenium WebDriver* împreună cu tehnici de *BDD*(*Behavior-Driven Development*) folosind un limbaj natural, *Gherkin*.

Acest framework prezintă un model pentru testarea automată a unei aplicații web, un magazin *online*. Framework-ul conține cinci teste care exemplifică cum pot fi testate funcțiile majore ale magazinului.

1. *test\_create\_account* – se testează funcționalitatea de a crea un cont nou
2. *test\_cart\_info* – se verifică informațiile produselor adăugate în coș
3. *test\_login* – aici se verifică posibilitate de logare cu diferite combinații de utilizator și parolă
4. *test\_complete\_order* – se verifică funcționalitatea de finalizare a unei comenzi
5. *test\_sort\_products* – se verifică că produsele sunt afișate corect în funcție de metoda de sortare aleasă

Structura proiectului trebuie organizată foarte bine încă de la început, aceasta constă în crearea unui cod curat a cărui logică și dependențe sunt clare, precum și modul în care *folder*-ele sunt organizate în sistemul de fișiere. În figura 4.1 este prezentată structura proiectului:

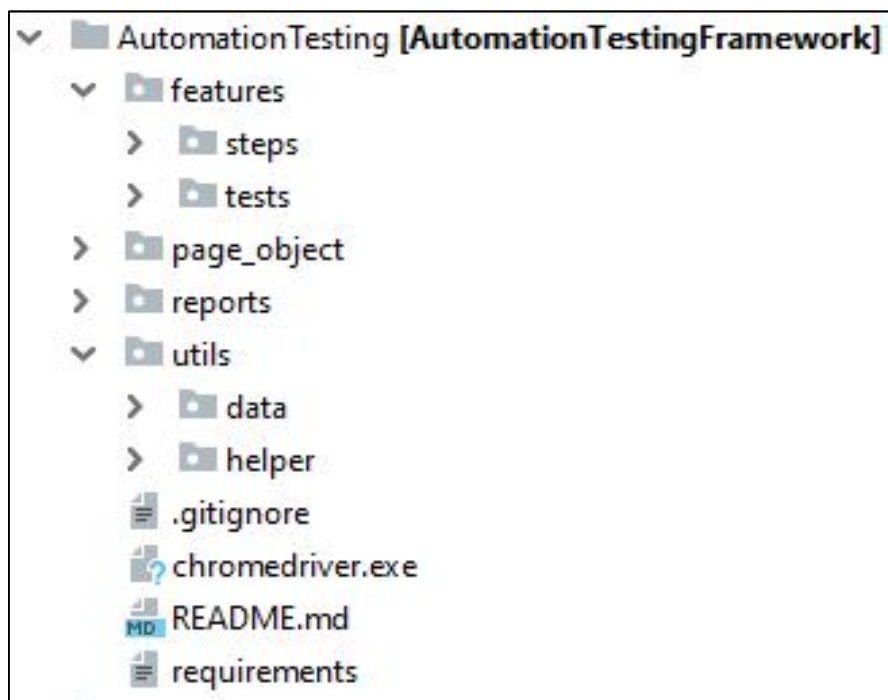


Figura 4.1 Structura proiectului

Organizarea granulară a fișierelor poate fi văzută ca un dezavantaj, îngreunând lucrul cu diferite clase din diferite pachete.

Avantajul unui cod scris în felul următor este flexibilitatea și ușurința cu care acesta poate fi întreținut, dacă trebuie să facem o schimbare, nu este necesar să schimbăm peste tot, ci doar într-un loc.

Fișierul **features** este denumit după cuvântul cheie *Features* din *Gherkin*, deoarece aici se desfășoară partea de *BDD*. Acesta conține alte două fișiere:

- Fișierul **tests** – în acest fișier sunt cele cinci teste propriu-zise, scrise în limbajul *Gherkin*
- Fișierul **steps** – aici sunt definiți toți pașii utilizați în teste. Implementarea fiecărui pas presupunând crearea unei funcții

În fișierul **page\_object** este implementat modelul *Page Object*. Aici sunt declarate paginile *web* și funcționalitățile acestora fiind reprezentate prin clase și funcții.

În fișierul **reports** sunt rapoartele cu rezultatul testelor, generate în urma rulării. Despre executarea și generarea raportului se vorbește mai pe larg în capitolul 5.

Fișierul **utils** conține două *foldere*:

- Fișierul **data** – unde sunt stocate toate datele de intrare folosite în teste. Aceste date sunt stocate în fișiere de tip *CSV(eng .Comma Separated Values)*
- Fișierul **helper** – conține clase care ajută la eliminarea duplicării codului, deoarece conțin funcții care pot fi reutilizate în întreaga structură. Pe lângă aceste clase, mai există un fișier de proprietăți unde sunt stocate informațiile statice ale *framework*-ului precum tipul *browser*-ului

Fișierul **.gitignore** conține tipul sau numele fișierelor pe care nu dorim să le avem în *repository*-ul nostru creat pe platforma *GitHub*. Fișiere tranzitorii din proiect, de obicei fișiere care nu sunt utile să fie știute de alți colaboratori, cum ar fi produsele de compilare.

Structura conține executabilul **chromedriver.exe** care este necesar pentru testarea aplicației în browserul *Chrome*.

Fișierul cu numele **README.md** conține informații despre alte fișiere și directoare dintr-un proiect. Aceste informații sunt legate de configurare, instalare, instrucțiuni de utilizare, etc.

Fișierul **requirements** este un fișier de tip text, ce conține cerințele aplicației *web*. Cum ar trebui să se comporte sistemul, ce funcționalități trebuie testate și ce date de ieșire trebuie să avem.

Acest fișier nu este obligatoriu să fie în structura *framework*-ului, există foarte multe programe care sunt special folosite pentru definirea cerințelor.

Pentru implementarea testelor, și pentru setarea întregului *framework* s-au folosit diferite programe și *framework*-uri. Instalarea și configurarea acestora vor fi prezentate în subcapitolul 4.2.

## 4.2 Configurarea programelor

Pentru realizarea acestui *framework* de testare s-a ales limbajul *Python* deoarece este un limbaj scriptat și mult mai compact față de alte limbaje de programare. Limbajul este scris dinamic astfel încât tipul variabilelor nu trebuie declarat. Altă caracteristică importantă a limbajului pentru care este folosit în testarea automată, este ușurința cu care pot fi înțelese testele, și codul în general deoarece *Python* conține sintaxe simple.

Primul pas este descărcarea limbajului de programare *Python* de pe *site*-ul oficial, iar mai apoi instalarea acestuia. Versiunea aleasă pentru acest proiect este 3.7, aceasta fiind și ultima lansată.

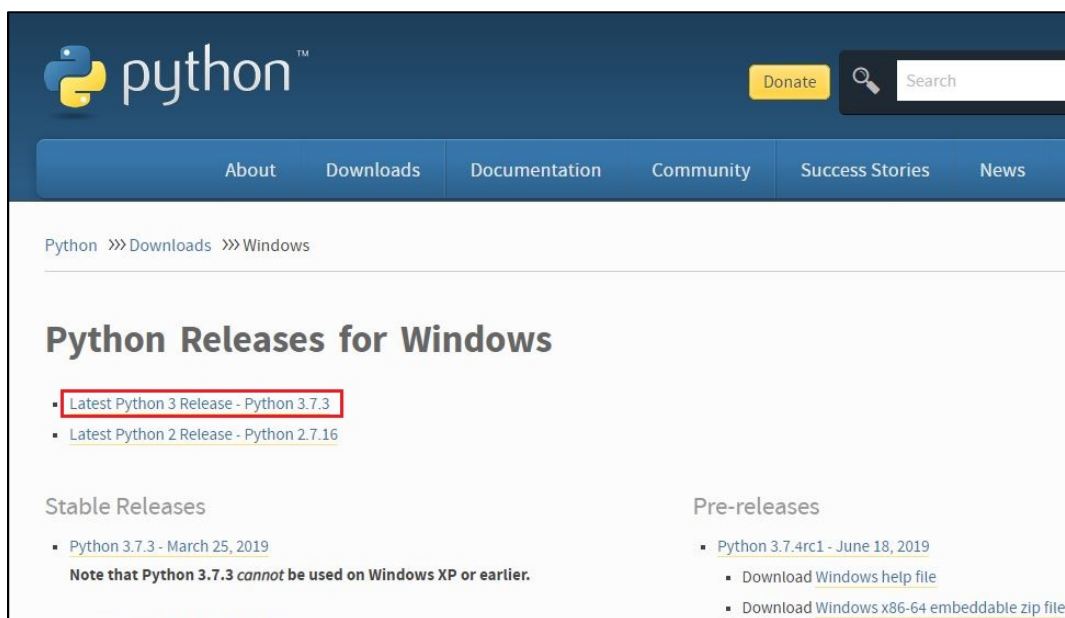


Figura 4.2 Pagina oficială Python [22]

Al doilea pas constă în descărcarea unui mediu de dezvoltare(*IDE*). Aceste medii de dezvoltare oferă integrarea cu sisteme de control al versiunilor, oferă suport în analizarea codului și în depanarea acestuia.

În această lucrare s-a ales mediul *Pycharm Professional* fiind un mediu dezvoltat pentru limbajul de programare *Python* și care totodată integrează foarte bine *framework*-ul de testare *Behave*. Există și varianta *Pycharm Community*, însă aceasta nu conține integrarea cu *Behave*.

În figura de mai jos este prezentată pagina oficială de unde poate fi descărcat mediul de dezvoltare:

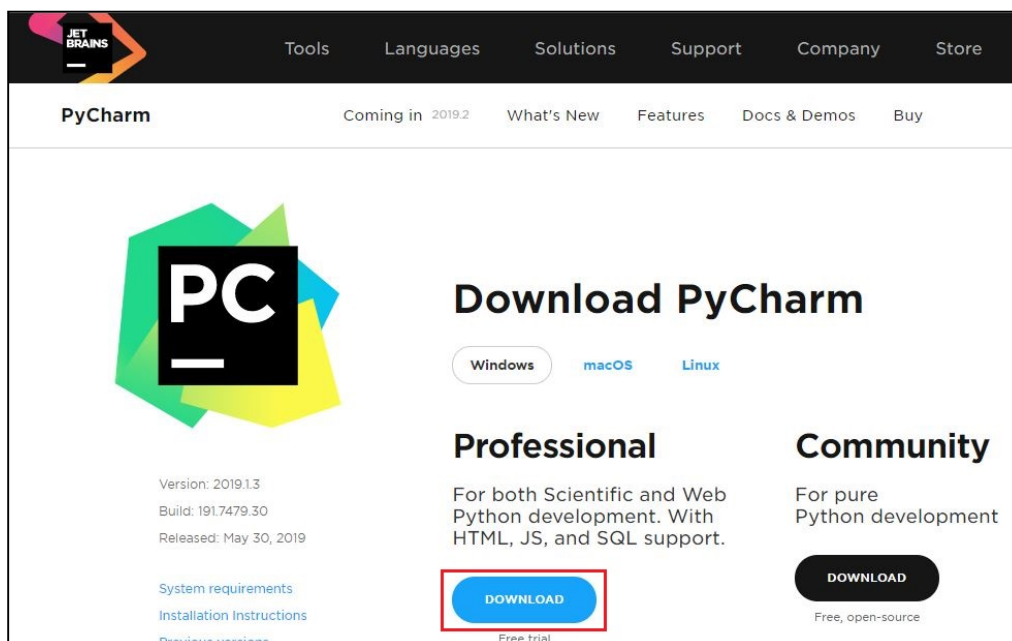


Figura 4.3 Pagina oficială Pycharm [23]

Un lucru foarte important pentru setarea proiectului este configurarea unui mediu virtual(*eng. virtual environment*). Scopul principal al mediilor virtuale este gestionarea setărilor și dependențelor unui anumit proiect, izolat de alte proiecte *Python*. Instrumentul *virtualenv* vine însoțit de *PyCharm*, astfel încât nu are nevoie de o instalare separată. Mai jos este prezentat modul în care trebuie selectat mediul virtual:

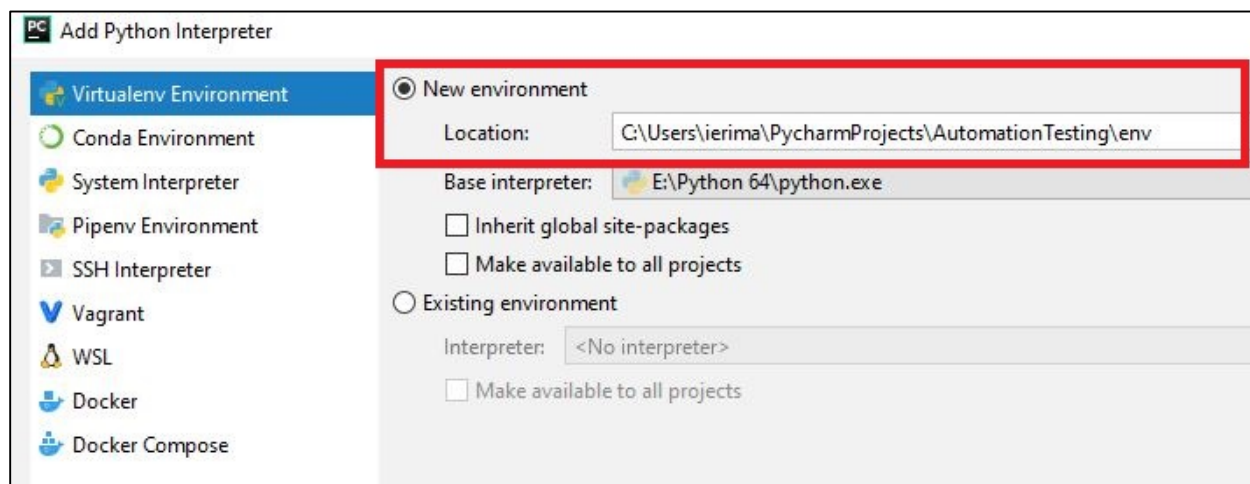


Figura 4.4 Configurarea mediului virtualenv

După ce am ales limbajul de programare și mediul de dezvoltare potrivit, următorul pas constă în instalarea *framework*-ului de testare Selenium *WebDriver*, care a fost prezentat în

subcapitolul 3.2. Instalarea se poate face descărcând *driver-ul Selenium*, accesând pagina oficială a acestora, sau se poate instala prin intermediul sistemului de gestionare *pip*.

*Pip* este un sistem de gestionare utilizat pentru instalarea și gestionarea pachetelor *software* scrise în *Python*. Acesta este inclus în mod implicit pentru versiunea *Python 3.4* sau versiunile ulterioare.

Pentru a instala Selenium, trebuie să deschidem *command prompt* și să introducem următoarea comandă:

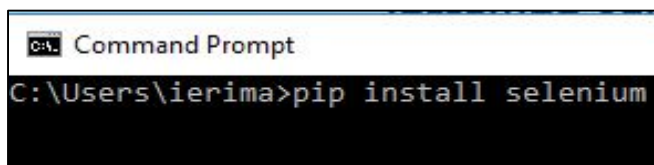


Figura 4.5 Comanda pentru instalare Selenium

Pentru rularea testelor, vom avea nevoie de un *browser*. Pentru acest proiect s-a ales *ChromeDriver* fiind printre cele mai populare din lume. Executabilul va trebui descărcat de pe site-ul oficial [24], iar mai apoi adăugat în structura proiectului.

Ultimul lucru pe care trebuie să îl facem pentru configurarea proiectului este să instalăm framework-ul *Behave* prin intermediul sistemului de gestionare *pip*, folosind comanda „*pip install behave*”.

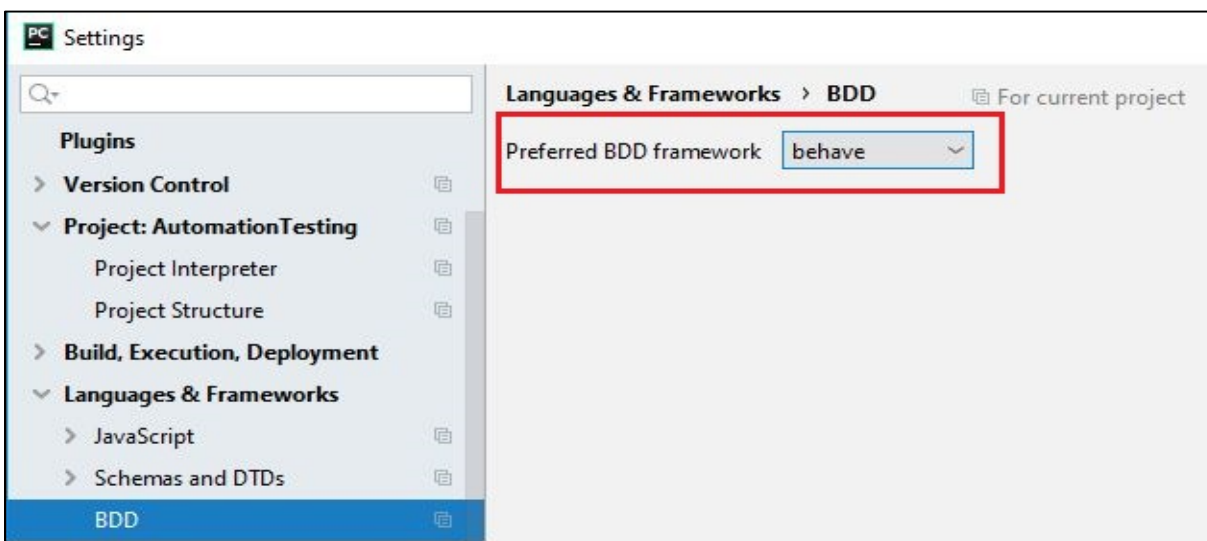


Figura 4.6 Adăugarea framework-ului behave

Opțional se poate adăuga modulul *node.tools*, care conține funcțiile „*assert*” de verificare și validare a rezultatelor. Acest modul se instalează cu comanda „*pip install node*”. Funcțiile utilizate în acest proiect sunt:

- *assert\_true()*
- *assert\_false()*
- *assert\_equal()*

### 4.3 Implementarea codului

În acest capitol se va prezenta implementare codului pentru realizarea testelor. Se vor prezenta clasele ce implementează testele efective, cât și celelalte fișiere adiționale.

Mai jos este prezentată arhitectura *framework*-ului pe baza căruia s-a realizat implementarea codului.

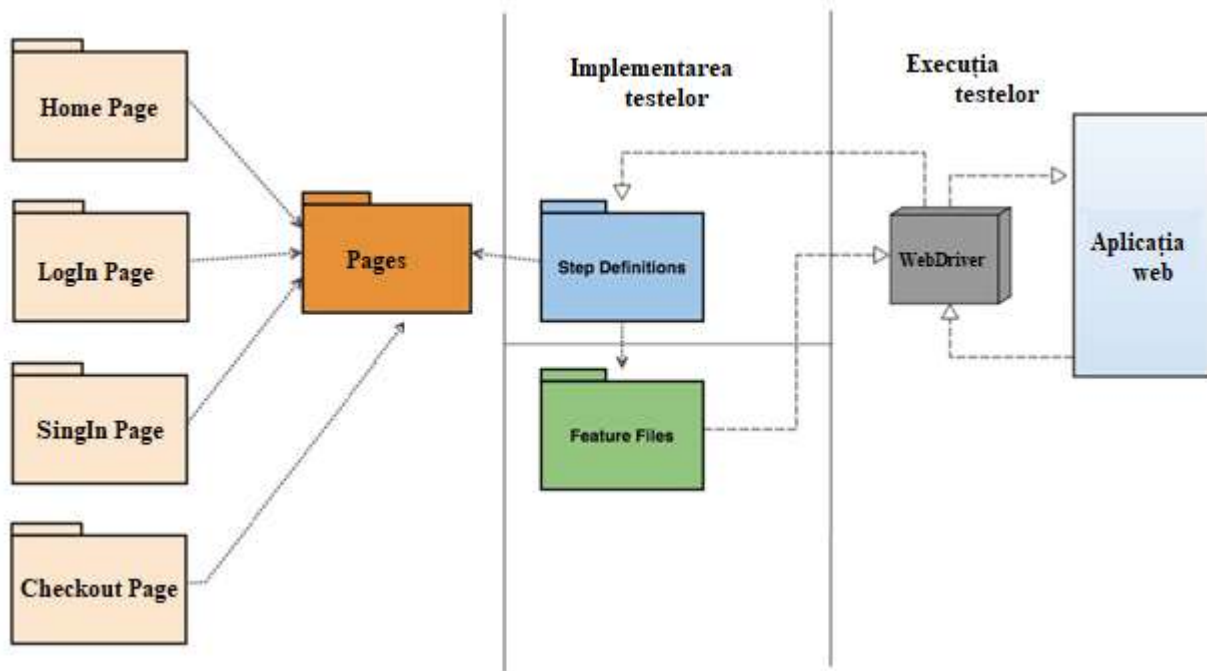


Figura 4.7 Arhitectura framework-ului

În figura de mai jos este prezentată diagrama claselor ce constituie modelul *Page Object*. Din această diagramă se pot vedea relațiile între clase.

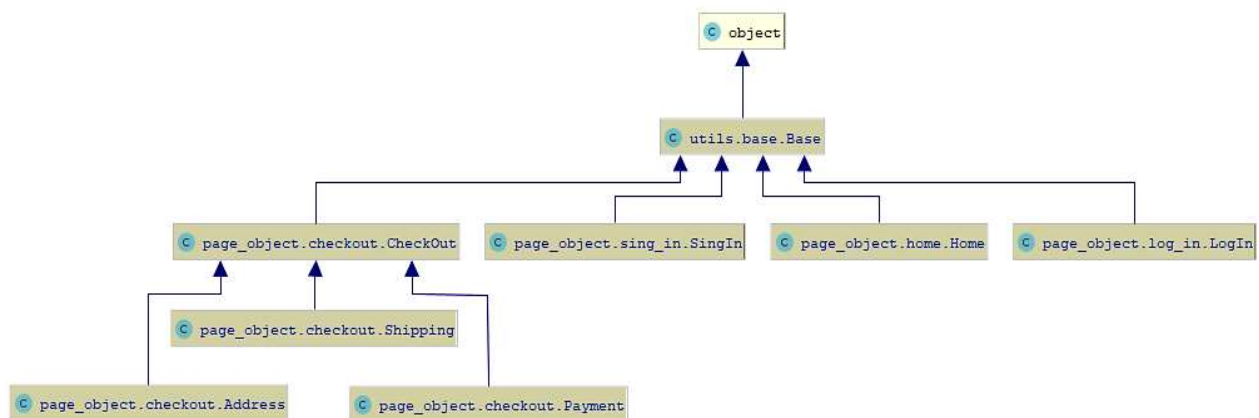


Figura 4.8 Diagrama claselor pentru modelul Page Object



Înainte de crearea efectivă a testelor avem nevoie de o configurare comună pentru toate testele. Această configurare a fost făcută în fișierul *environment.py* din folder-ul *features*. În acest fișier am declarat funcțiile care vor fi executate înainte ca execuția testelor să înceapă, după ce execuția s-a încheiat sau înainte ca fiecare test să ruleze.

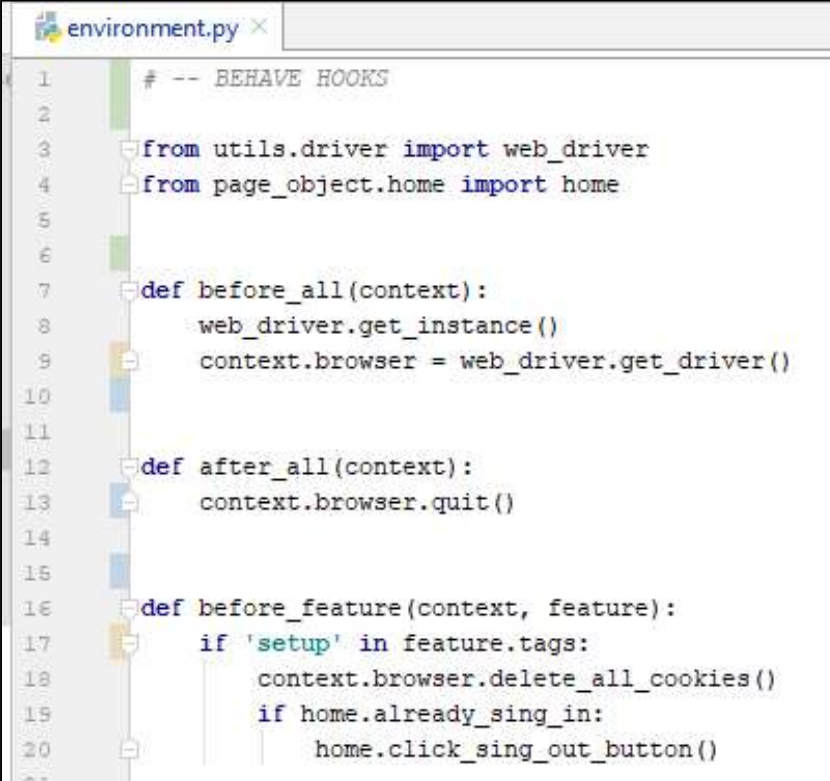
Cu ajutorul funcției *before\_all()* am apelat metodele de inițializare a *browser*-ului din clasa *Driver()*. Deci înainte ca execuția testelor să înceapă se va realiza o instanță a *browser*-ului Chrome.

Cu ajutorul funcției *after\_all()* am încheiat această instanță, așadar după execuția testelor, *tab*-urile din *browser* se vor închide.

Funcția *before\_feature()* va executa funcțiile declarate în interiorul ei, înainte ca fiecare *Feature(test)* să fie executat. Pentru ca această funcție să fie apelată, trebuie să folosim *tag*-uri declarate cu simbolul „@nume\_tag”, înaintea testului unde vrem să utilizăm funcția.

Deoarece vom rula toate testele în aceeași instanță de browser pentru a câștiga timp, trebuie să ne asigurăm că nici o setare de la testul anterior nu a rămas, afectând execuția testului curent.

Mai jos este prezentat fișierul *environment.py*:



```
1  # -- BEHAVE HOOKS
2
3  from utils.driver import web_driver
4  from page_object.home import home
5
6
7  def before_all(context):
8      web_driver.get_instance()
9      context.browser = web_driver.get_driver()
10
11
12  def after_all(context):
13      context.browser.quit()
14
15
16  def before_feature(context, feature):
17      if 'setup' in feature.tags:
18          context.browser.delete_all_cookies()
19          if home.already_sing_in:
20              home.click_sing_out_button()
```

Figura 4.9 Fișierul *environment.py*

Clasa `Driver()` conține funcțiile legate de *browser* și de instanța acestuia. Pentru a avea o singură instanță de browser am folosit *pattern-ul* „*Singleton*”.

```
## Driver class -setup for the driver
class Driver():
    instance = None

    @classmethod
    def get_instance(cls):
        if cls.instance is None:
            cls.instance = Driver()
        return cls.instance

    def __init__(self):
        chrome_options = Options()
        chrome_options.add_argument("--disable-infobars")
        chrome_options.add_argument("--start-maximized")
        self.driver = webdriver.Chrome(chrome_options=chrome_options)

    def get_driver(self):
        return self.driver

# close the browser window
def tearDown(self):
    self.driver.quit()

def load_website(self):
    self.driver.get(values.BASEURL)

web_driver = Driver.get_instance()
```

„*get\_instance*” este o metodă de clasă care este utilizată pentru a crea instanță clasei *Driver*. Pentru a putea utiliza această clasă, metoda trebuie apelată: „*Driver.get\_instance()*”, asigurându-ne astfel că numai o instanță a clasei (obiectul *singleton*) este utilizată în întregul proiect.

Pentru realizarea testelor în limbajul *Gherkin*, am creat fișiere cu extensia „*feature*” în *folder-ul* *features*. În continuare se va prezenta implementarea celor cinci teste prin exemplificarea secvențelor principale de cod.

### Testarea funcționalității de creare a unui cont nou

În figura 4.9 este prezentat testul ***test\_create\_account.feature*** scris în limbajul *Gherkin*. În acest test se testează funcționalitatea de creare a unui nou cont, completând doar informațiile obligatorii. După ce contul a fost creat cu succes, se verifică dacă informațiile folosite vor fi afișate corect după autentificare.

```

3  @setup
4  Feature: Create new account
5  """
6  After a new account was created the
7  home page should display the user name( First Name Last Name)
8  at the top-right corner of the page
9  """
10 Background: I navigate to Sing In page
11     Given I load the website
12     And I go to Sing In page
13     Then Create Account field should be displayed
14
15 """
16 An account can be created just
17 with mandatory information
18 """
19 Scenario: Create an account with mandatory information
20
21     Given I insert 'random_email' into Create account field
22     When I create a new account for 'random_email' with mandatory user information
23     Then I verify if the account 'random_email' was created

```

Figura 4.10 Implementarea testului `test_create_account.feature`

În *Background* sunt declarate pre-condițiile care trebuie executate înainte ca testarea funcționalității să înceapă.

Încărcarea site-ului pe pagina principală a acestuia se realizează prin intermediul funcției „*load\_website*”, apelată în pasul „*I load the website*”:

```

@given("I load the website")
def step(context):
    web_driver.load_website()

```

Se apelează funcția „*get*” care are ca parametru *URL* site-ului web declarat într-un fișier de proprietăți „*values*”.

```

def load_website(self):
    self.driver.get(values.BASEURL)

```

Navigarea către pagina *Sing In* este realizată în pasul „*I go to Sing In page*”, unde este apelată funcția „*click\_sing\_in\_button()*” în care se va realiza apăsarea butonului *Sing In*.

```

@given("I go to Sing In page")
def step_impl(context):
    home.click_sing_in_button()

```

Apoi se verifică dacă este afișat câmpul pentru crearea unui nou cont, pentru a ne asigura că începerea testului se va realiza în condițiile dorite.

Toți pașii descriși în interiorul scenariului reprezintă testul propriu-zis. Pentru crearea unui nou cont, trebuie să introducem un email care nu a fost deja înregistrat. Deoarece nu avem acces la baza de date pentru a putea șterge utilizatorul după fiecare

rulare a testului, vom folosi email-uri generate *random* cu ajutorul funcției „*generate\_random\_emails()*” din clasa *DataHandler()*:

```
def generate_random_emails(self):
    return [self.get_one_random_name(self.letters) + '@' +
            self.get_one_random_domain(self.domains) for i in range(1)][0]

def get_one_random_domain(self, domains):
    return random.choice(domains)

def get_one_random_name(self, letters):
    return ''.join(random.choice(letters) for i in range(7))
```

Pentru completarea email-ului se va folosi funcția „*enter\_email\_to\_create\_account()*” care are ca parametru adresa de email pe care vrem să o introducem.

```
# Types email on 'Email Address' field from Create An Account
def enter_email_to_create_account(self, email):
    field = self._create_account_email_field_id
    if element.is_element_present(By.ID, field):
        self.type_into_a_field(By.ID, field, email)
    else:
        raise TypeError("Element '%s' can not be found" % field)
```

Pentru introducerea informațiilor obligatorii în crearea unui nou cont, s-au creat funcții diferite pentru fiecare tip de informație, prezente în clasa *LogIn()*.

```
# Fill all the user information
log_in.select_gender(csv_name, DataHandler().test_data(create_account,
'title', email))
log_in.enter_first_name(csv_name, DataHandler().test_data(create_account,
'first_name', email))
```

Aceste informații sunt luate dintr-un fișier *CSV*(eng. *Comma Separated Values*):

	A	B	C	D	E	F	G
1	email	title	first_name	last_name	password	date_of_birth	newsletter
2	ruxi.ierima@gmail.com	Mrs	Ruxandra	Ierima	Ruxandra11	2/4/1997	yes
3	<a href="mailto:alex.pop@yahoo.com">alex.pop@yahoo.com</a>	Mr	Alex	Pop	alex	9/2/1980	yes
4	random_email	Mrs	Anamaria	Popescu	ana_popescu3	26/6/1978	yes

Figura 4.11 Fișierul CSV

Datele sunt extrase din fișierul *CSV* cu ajutorul funcției *test\_data()*, prin intermediul modului *pd*, importat din librăria *pandas*.

Mai jos este prezentată funcția *test\_data()*, din clasa *DataHandler()*:

```
class DataHandler():

    # Read data from csv files
    # header - the header of a column
    # row_value - row header for witch you want to get data
    def test_data(self, file_name, header, row_value):

        # name of file to read from
        r_filenameCSV = '../..AutomationTesting/features'+file_name+'.csv'

        # read the data
        csv_read = pd.read_csv(r_filenameCSV)

        for index, value in enumerate(csv_read[0]):
            if value == row_value:
                location_in_csv = index
                break
        return csv_read[header][location_in_csv]
```

Pentru a verifica dacă contul s-a creat cu succes este nevoie să ne logăm cu *email*-ul și parola introduse la crearea contului. Această verificare se realizează prin executarea altor pași deja existenți.

```
context.execute_steps(u"""
    given I sing in with email:{email} and pass:{password}
    when I press the Sing In button
    then I verify if the account name is:{expected_account_name}
    """.format(email=random_email, password=password,
expected_account_name=account_name))
```

Se verifică dacă numele și prenumele utilizatorului din CSV sunt afișate odată ce ne-am logat.

```
@then("I verify if the account name is:{expected_account_name}")
def step_impl(context, expected_account_name):
    current_account_name = home.get_account_name
    assert_equal(current_account_name, expected_account_name)
```

În figura de mai jos este prezentată pagina pentru crearea unui nou cont, unde se vor completa informațiile:

YOUR PERSONAL INFORMATION	YOUR ADDRESS	Zip/Postal Code *
<b>Title</b> <input type="radio"/> Mr. <input type="radio"/> Mrs.	<b>First name *</b> <input type="text"/>	<input type="text"/>
<b>First name *</b> <input type="text"/>	<b>Last name *</b> <input type="text"/>	<b>Country *</b> <input type="text" value="United States"/>
<b>Last name *</b> <input type="text"/>	<b>Company</b> <input type="text"/>	<b>Additional information</b> <input type="text"/>
<b>Email *</b> <input type="text" value="vvfrfr@gmail.com"/>	<b>Address *</b> <input type="text"/>	<b>You must register at least one phone number.</b> <b>Home phone</b> <input type="text"/>
<b>Password *</b> <input type="text"/>	<b>Address (Line 2)</b> <input type="text"/>	<b>Mobile phone *</b> <input type="text"/>
(Five characters minimum) <b>Date of Birth</b> <input type="text"/> - <input type="text"/> - <input type="text"/>	<b>City *</b> <input type="text"/>	<b>Assign an address alias for future reference. *</b> <input type="text" value="My address"/>
<input type="checkbox"/> Sign up for our newsletter! <input type="checkbox"/> Receive special offers from our partners!		<input type="button" value="Register &gt;"/>

Figura 4.12 Pagina pentru crearea unui cont nou

### Testarea funcționalității de finalizare a unei comenzi

În acest test se vor verifica informațiile referitoare la adresa de livrare în momentul plasării unei comenzi și se va testa dacă o comandă poate fi finalizată, plătind prin două modalități: prin card bancar sau plățirea la sosirea coletului.

```

test_complete_order.feature x
12  """Delivery address should be displayed
13  when user want to complete an order
14  """
15  Scenario: Check delivery address
16    Given I add product 'Faded Short Sleeve T-shirts,Printed Summer Dress' to the cart
17    When I go to delivery address
18    Then I check delivery address for account: 'ruxi.ierima@gmail.com'
19
20  """User should be able to pay for an order via
21  bank wire or via
22  """
23  Scenario Outline: Payment methods
24    Given I add product 'Faded Short Sleeve T-shirts,Blouse' to the cart
25    When I proceed to checkout
26    And I choose to pay via <Paymethod>
27    Then The confirmation of the <Paymethod> payment should be displayed
28    And I should be able to submit my order
29
30  Examples:
31    | Paymethod |
32    | bank      |
33    | check     |

```

Figura 4.13 Implementarea testului test\_complete\_order.feature



Primul pas constă în adăugarea produselor dorite în coș. Pentru a realiza acest lucru trebuie să utilizăm o funcție *for* pentru a itera prin lista de produse. Folosim clasa *ActionChains()* pentru manipularea evenimentelor *mouse*-ului, pentru a ne muta de la un produs la altul.

```
# Add specific product to the cart by name
# product_name - The name of the desired product
def add_specific_product_in_cart(self, desired_product_name):
    product_containers = self.driver.find_element(By.ID,
"homefeatured").find_elements_by_tag_name("li")

    for index, product in enumerate(product_containers):
        product_name = product.text.split('\n')[0]
        if desired_product_name == product_name:
            ActionChains(self.driver).move_to_element(product).perform()
            i = index + 1
            _add_to_cart_locator_xpath =
'//*[@id="homefeatured"]/li[%s]/div/div[2]/div[2]/a[1]/span' % i
            self.driver.find_element(By.XPATH,
add to cart locator xpath).click()
            break
```

După ce produsele au fost adăugate în coș, și am mers la secțiunea de finalizare a cumpărăturilor, trebuie să verificăm dacă adresa de livrare corespunde cu adresa adăugată la crearea contului.

Deci se va verifica adresa din CSV corespondentă contului cu care ne-am înregistrat, și adresa afișată.

```
@then("I check delivery address for account: '{email}'")
def step_impl(context, email):
    address.verify_account_information(email)
```

Pentru informațiile afișate am creat un dicționar unde vom stoca aceste date. Extragerea acestora se va face comparând atributele din lista declarată în codul *HTML*.

```
def get_delivery_address(self):
    address_information={'name':None,
                        'address':None,
                        'city_state_postal_code':None,
                        'country':None,
                        'phone':None
    }

    all_information = self.driver.find_element(By.CSS_SELECTOR ,
self._delivery_address_css).find_elements(By.TAG_NAME, 'li')

    for information in all_information:
        class_attribute = information.get_attribute('class')
        if class_attribute == 'address_firstname address_lastname':
            address_information['name'] = information.text
```

Pentru cel de al doilea scenariu, primii pași sunt la fel, singurele diferențe ar putea fi produsele adăugate. Alegerea metodei de plată se va face în pasul „*I choose to pay via*

<Paymethod>”, unde *Paymethod* va lua două valori, *banck* și *check*, astfel se vor rula două teste, dar fără a duplica codul.

Funcțiile utilizate pentru verificarea metodei de plată aleasă cât și verificarea funcționalității de plasare a unei comenzi sunt prezentate în secvențele de mai jos:

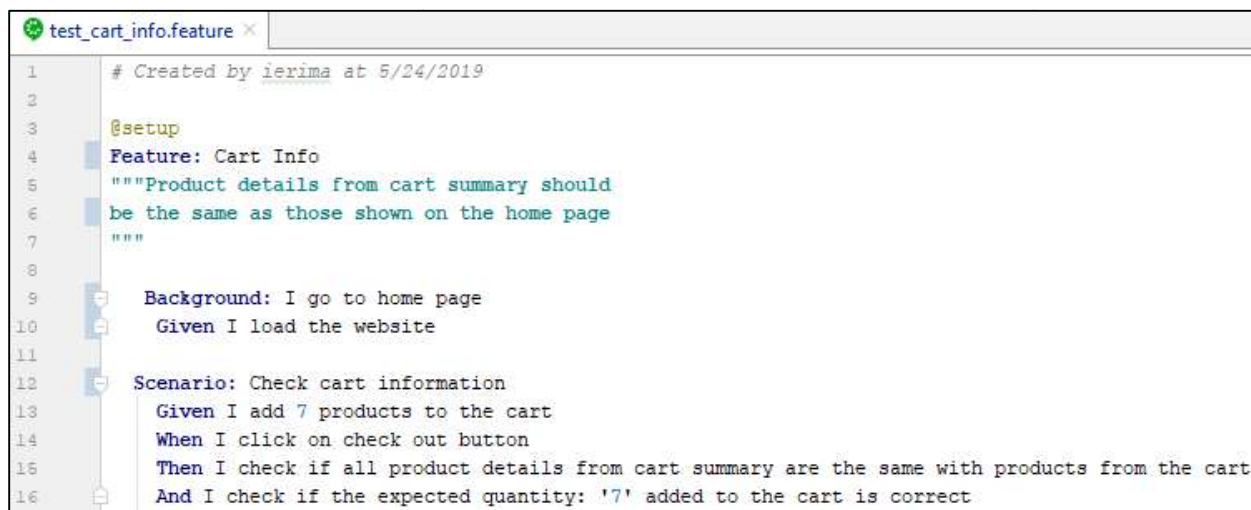
```
@then("The confirmation of the {payment_method} payment should be displayed")
def step_impl(context, payment_method):
    assert payment_method.upper() in
    payment.get_chosen_payment_method.upper()

@then("I should be able to submit my order")
def step_impl(context):
    payment.click_confirm_order()
    assert_equal(True, payment.is_order_complete, 'The order is complete with
    chosen payment method')
```

### Testarea funcționalității de adăugarea a produselor

În testul *test\_cart\_info.feature* se verifică dacă caracteristicile produselor introduse în coș sunt afișate corect în secțiunea „Summary”, în cadrul procesării comenzilor. Aceste caracteristici constau în numele produsului, cantitatea și prețul lor.

Mai jos este prezentată implementarea testului în limbajul *Gherkin*:



```
test_cart_info.feature x
1 # Created by ierima at 5/24/2019
2
3 @setup
4 Feature: Cart Info
5     """Product details from cart summary should
6     be the same as those shown on the home page
7     """
8
9     Background: I go to home page
10    Given I load the website
11
12    Scenario: Check cart information
13        Given I add 7 products to the cart
14        When I click on check out button
15        Then I check if all product details from cart summary are the same with products from the cart
16        And I check if the expected quantity: '7' added to the cart is correct
```

Figura 4.14 Implementarea testului *test\_cart\_info.feature*

Rezumatul comenzii se poate vedea după ce produsele au fost selectate. Informațiile produselor sunt afișate sub forma unui tabel, precum în figura 4.15.







Product	Description	Avail.	Unit price	Qty	Total
	Faded Short Sleeve T-shirts SKU : demo_1 Color : Orange, Size : S	In stock	\$16.51	1 - +	\$16.51
	Blouse SKU : demo_2 Color : Black, Size : S	In stock	\$27.00	1 - +	\$27.00
	Printed Dress SKU : demo_3 Color : Orange, Size : S	In stock	\$26.00	1 - +	\$26.00
	Printed Dress SKU : demo_4 Color : Beige, Size : S	In stock	\$50.99	1 - +	\$50.99

Figura 4.15 Caracteristicile produselor afișate în tabel

Pentru extragerea informațiilor am folosit o funcție *for*, astfel am parcurs fiecare rând din tabel, adăugând informațiile într-o listă.

```
rows = body_element.find_elements(By.TAG_NAME, "tr")

for row in rows:
    product_properties=row.find_elements(By.TAG_NAME,"td")
    for product_property in product_properties:
        if product_property.get_attribute("class") == 'cart_description':
            product_name.append(product_property.text.split('\n')[0])
```

Tot într-o listă s-au introdus și informațiile afișate în coșul de cumpărături, verificarea dintre cele două s-a făcut prin compararea acestor liste.

### Testarea funcționalității de „Log In ”

Acest test constă în testarea securității sistemului. Se vor introduce combinații diferite de utilizator și parolă cu scopul de a vedea dacă sunt afișate mesaje de eroare.

```

"""
When a user want to login with an INVALID email/password
should NOT be able to login successfully and an error message should
appear: 'Invalid email address.'
"""
Scenario Outline: Sing in with invalid email and password

    Given I sing in with email:<Username> and pass: <Password>
    When I press the Sing In button
    Then I check if the expected error message:Invalid email address. is displayed in Sing In field

Examples:
    | Username | Password |
    | ruxi | Ruxandra11 |
    | dsjds | dads |

```

Figura 4.16 Testul test\_sing\_in.feature (email și parolă invalide)

```

"""
When a user want to login with an VALID email and a WRONG password
should NOT be able to login successfully and an error message should
appear: 'Authentication failed'
"""
Scenario: Sing in with valid email and invalid password

    Given I sing in with email:ruxi.ierima@gmail.com and pass:ffwfwfwf
    When I press the Sing In button
    Then I check if the expected error message:Authentication failed. is displayed in Sing In field

```

Figura 4.17 Testul test\_sing\_in.feature (email valid și parolă invalidă)

## Testarea funcționalității de sortare a produselor

În acest test se verifică dacă procesul de sortare a produselor, în funcție de anumite criterii funcționează.

```

4 Feature: Sort search results
5 """
6
7 When a user perform a product search
8 and choose a sort method the results should
9 be displayed by chosen method and should be relevant
10 """
11
12 Scenario Outline: Sort products
13     Given I load the website
14     When I search product 'dress'
15     And I sort results by <sort_method>
16     Then Results should be displayed by <sort_method>
17     And All the results should be relevant for search 'dress'
18
19 Examples:
20     | sort_method |
21     | Price: Lowest first|
22     | Price: Highest first|

```

Figura 4.18 Implementarea testului test\_sort\_products.feature

Primul pas ce trebuie implementat este introducerea unui *string* în secțiunea de căutare. Rezultate căutării vor fi afișate în partea de jos a paginii, unde putem să le vizualizăm după diferite criterii. Pentru acest test s-au folosit două dintre aceste metode de sortare: afișarea produselor ascendent și descendent în funcție de prețul acestora.

În funcția de mai jos se verifică dacă rezultatele obținute sunt relevante pentru produsul căutat:

```
@then("All the results should be relevant for search '{expected_result}'")
def step_impl(context, expected_result):
    results_list = home.get_list_of_results_name
    for result in results_list:
        assert_in(str(expected_result).upper(), str(result).upper())
```

Pentru verificarea produselor sortate s-a realizat o comparație între lista ce conține prețurile afișate în pagină și lista sortată.

```
@then("Results should be displayed by {sort_method}")
def step_impl(context, sort_method):
    displayed_products_price = home.get_list_of_results_price
    expected_list = home.get_list_of_results_price
    if 'Lowest' in sort_method:
        expected_list.sort()
    if 'Highest' in sort_method:
        expected_list.sort(reverse=True)

    assert_equal(displayed_products_price, expected_list)
```

Toate clasele care constituie modelul *PageObject* moștenesc clasa *Base* deoarece aceasta conține metodele comune pentru toate testele. Mai jos sunt prezentate câteva dintre aceste metode :

Introducerea textului într-un câmp:

```
##Types a string into a speciffic field
def type_into_a_field(self,how,where,what):
    element_field=self.driver.find_element(by=how, value=where)
    if element_field.get_attribute("value") is not None:
        element_field.clear()
    element_field.send_keys(str(what))
```

Selectarea elementelor dintr-o listă *dropdown*:

```
def select_value_from_dropdown_list(self,how,where,value):
    select = Select(self.driver.find_element(by=how, value=where))
    select.select_by_visible_text(value)
```

Selectarea elementelor de tip *radio*:

```
def select_radio_button(self,how,where,value):
    radio_buttons = self.driver.find_elements(by=how,value=where)
    for item in radio_buttons:
        if value in item.text and not item.is_selected():
            item.click()
```

Mutarea *mouse*-ului pe un element:

```
def move_to_element(self, how, what):
    element = self.driver.find_element(by=how, value=what)
    actions = ActionChains(self.driver)
    actions.move_to_element(element).perform()
```

**Introducerea datei într-un *datepicker*:**

```
#value format: 'dd/mm/yyyy'
def type_into_date_picker(self, how, where, value):
    data_list = str(value).split('/')
    dictionary_data = {'days': None,
                       'months': None,
                       'years': None
                      }
    for index, key in enumerate(dictionary_data):
        dictionary_data[key] = (data_list[index])

    data_elements = self.driver.find_elements(by=how, value=where)

    for index, item in enumerate(data_elements):
        element = item.find_element(By.ID, str(list(dictionary_data.keys())
[index]))
        Select(element).select_by_value(data_list[index])
```

**Verificarea dacă un buton este deja selectat:**

```
# Checks if a button is already selected
def check_if_button_is_selected(self, how, where):
    return self.driver.find_element(by=how, value=where).is_selected()
```

## 5 Rezultate experimentale

### 6.1 Rularea testelor

Pentru execuția testelor prezentate anterior, vom folosi *runner*-ului *Behave* prin intermediul mediului de dezvoltare *Pycharm*. Testele pot fi rulate pe rând, toate odată, sau se pot selecta doar testele dorite.

Acest lucru se face din configurări, selectând calea spre numele fișierului ce conține testul. Calea fiecărui fișier fiind despărțită de simbolul „/”.

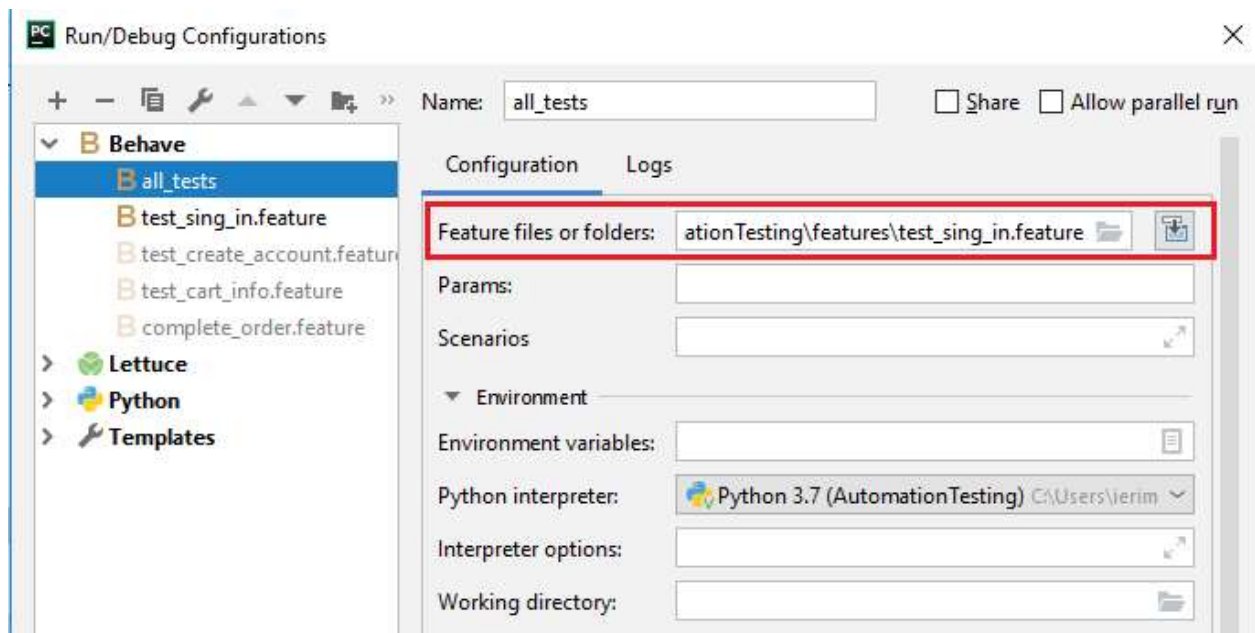


Figura 5.1 Configurarea testelor pentru rulare

Rezultatul în urma rulării se poate observa în partea de jos a ecranului, în secțiunea „Run”. Pentru rularea celor cinci teste prezentate în capitolul 4, se poate observa din figura 5.2 că dintr-un total de 64 de pași executați, 59 de pași au fost executați cu succes, trecând de partea de validare, trei pași au picat, reprezentând astfel erori în aplicație iar alți doi au fost ignorați. Aceștia au fost ignorați deoarece pași anteriori nu au trecut de partea de validare.

În partea din stânga se poate observa statusul fiecărui pas, cât și timpul de execuție în care a fost rulat fiecare pas în parte. În urma mesajelor afișate în partea dreaptă a ecranului se poate vedea din ce motiv testele nu au trecut de partea de validare.

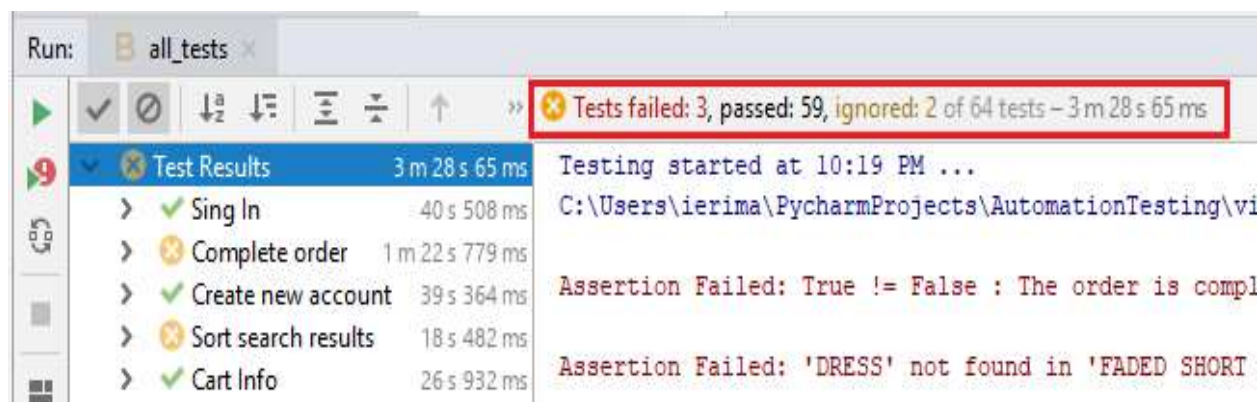


Figura 5.2 Rezultatul în urma rulării testelor

Rularea testelor se poate face și din linia de comandă, fără a mai fi nevoie să folosim mediul *Pycharm*. Tot ce trebuie să facem este să deschidem *cmd.exe* și să introducem diferite comenzi, folosite în rularea testelor.

Pentru a rula un test care are extensia *.feature*:

```
behave -i nume_test.feature
```

Dacă spre exemplu vrem să rulăm doar un anumit scenariu, putem folosi următoarea comandă:

```
behave -n 'Numele Scenariului'
```

## 6.2 Generarea raportului de validare

Documentarea rezultatelor joacă un rol esențial în testarea aplicațiilor. Raportul trebuie să fie bine structurat și detaliat pentru a fi cât mai ușor de înțeles. Acesta trebuie să cuprindă informații despre testele executate. Chiar dacă testele sunt realizate foarte bine, un raport neglijat poate să îngreuneze găsirea defectelor din aplicație.

De aceea pentru redactarea raportului, de cele mai multe ori se aleg programe speciale. Pentru acest proiect s-a ales *framework*-ul „*Allure*”, în care raportul este prezentat sub forma unei pagini *web*.

Primul lucru pe care trebuie să îl facem, este să instalăm extensia „*Allure*” prin intermediul comenzii: „*pip install allure-behave*” introduse în linia de comandă. După aceea am creat un *folder* numit „*reports*” în directorul proiectului, unde se vor salva rapoartele sub formă de fișiere *.json*. Pentru rularea testelor, vom folosi comanda:

```
behave -f allure_behave.formatter:AllureFormatter -o reports features
```

După rularea testelor, *folder*-ul *reports* va conține fișierele cu rezultatele execuției. Pentru a vizualiza raportul vom folosi comanda:

```
allure serve reports
```



Această comandă va deschide raportul în *browser*-ul setat implicit. În urma execuției a rezultat următorul raport:



Figura 5.3 Raportul Allure

Raportul conține numărul testelor executate, statusul pașilor pentru fiecare test în parte și timpul execuție.

În secțiunea „Suites” este detaliat statusul pașilor. Statusul poate fi reprezentat în cinci moduri diferite: „failed ” (pentru testele care prezintă un defect în aplicație), „passed ” pentru testele care au trecut, „broken ” (pentru testele care au erori în scripturi), „skipped ” (pentru testele care nu au mai fost rulate) și „unknown ” (pentru un status incert). În urma acestui raport vom avea o imagine de ansamblu cu privire la statusul aplicației din punct de vedere al calității. Prin prisma faptului că putem observa testele care au picat repartizate pe categorii, implicit defectele din aplicație vor fi mai ușor de identificat.

order	name	duration	status
#1	Check cart information	49s 434ms	passed
#2	Check delivery address	35s 375ms	passed
#5	Create an account with mandatory information	39s 102ms	passed
#3	Payment methods -- @1.1 bank	30s 961ms	failed
#4	Payment methods -- @1.2 check	23s 220ms	passed
#6	Sing in with invalid email and password -- @1.1 Ruxandra11, ruxi	15s 990ms	passed
#7	Sing in with invalid email and password -- @1.2 dads, dsjds	15s 339ms	passed
#8	Sing in with valid email and invalid password	15s 236ms	passed
#9	Sing in with valid email and valid password	15s 470ms	passed
#10	Sort products -- @1.1 Price: Lowest first	6s 928ms	failed
#11	Sort products -- @1.2 Price: Highest first	7s 046ms	failed

Figura 5.6 Secțiunea „Suites ” din cadrul raportului Allure

## 6 Concluzii

În acest proiect am realizat un *framework* pentru testarea automată a unui magazin online. De ce s-a ales testarea automată? În primul rând, testele trebuie repetate adesea în timpul ciclurilor de dezvoltare a unui produs. Testarea automată salvează astfel timp, care se reflectă direct și în reducerea costurilor. Totodată elimină repetarea manuală a testelor, implicit reducerea resurselor umane. S-a ales *framework*-ul *Selenium WebDriver* deoarece este printre cele mai complexe soluții când vine vorba de testarea automată a aplicațiilor *web*. Cu ajutorul acestuia am putut realiza interacțiunea cu elementele din pagină.

Scopul principal al acestui proiect este prezentarea unei alternative a *framework*-urilor tradiționale de testare automată. Opțiunea aleasă este limbajul non-tehnic *Gherkin*, folosit în procesul de dezvoltare a software-ului agil *BDD*. Această alternativă prezintă separarea cazurilor de testare, de codul efectiv. În acest fel toate persoanele implicate în procesul de testare, în principiu persoanele non-tehnice, vor putea înțelege proiectul și totodată va exista o comunicare mai bună. În general, cea mai întâlnită problemă pe care o găsim în cadrul echipelor, este lipsa de comunicare între echipa de testare/dezvoltare, care poate să înțeleagă greșit cerințele persoanelor implicate în parte de *bussines*, iar aceștia din urmă pot înțelege greșit ceea ce *testeri/developeri* sunt într-adevăr capabili.

Implementarea celor cinci teste, exemplifică câteva dintre cele mai importante acțiuni pe care un utilizator le-ar putea realiza într-un magazin *online*. Rezultatele obținute în urma rulării sunt apoi interpretare și afișate sub formă de grafice și diagrame într-o pagină *.html* cu ajutorul *framework*-ului *Allure*. Astfel, cu o interfață foarte sugestivă, defectele pot fi ușor de identificat. Pe baza testelor implementate și a execuției efectuate, s-a obținut o serie de rezultate prin care putem spune că obiectivele inițiale au fost atinse.

Pe parcursul dezvoltării părții practice am întâmpinat mai multe obstacole care s-au reflectat în implementarea testelor, referindu-ne aici la respectarea celor mai bune practici. Deoarece nu am avut acces la baza de date, unde se salvează toate datele utilizatorilor, nu am putut să ștergem un cont după ce acesta a fost creat într-unul dintre teste. Astfel, ar fi însemnat ca la fiecare rulare a testului, acesta ar fi generat o eroare referitoare la contul deja existent, ceea ce ar fi dus la un rezultat eronat. Ca alternativă, am folosit generarea unor *email-uri* „*random*”, lucru care a făcut posibilă rularea testului în condițiile dorite.

### 6.1 Dezvoltări ulterioare

Există posibilitatea îmbunătățirii ulterioare a acestui *framework* pentru diferite funcționalități. Un prim pas în dezvoltarea ulterioară ar fi testarea *API*(eng. *Application Programming Interface*). Aceasta ar ajuta la reducerea timpului, eliminând executarea unor pre-condiții realizate prin interfața grafică pentru a ajunge la testarea funcționalității



necesare. Lucru care ar elimina erorile apărute în teste din cauza schimbărilor din interfața grafică.

O posibilă dezvoltare a aplicației ar mai putea consta în integrarea continuă(*eng. Continuous Integration*). Practică în care un *build* se creează în momentul în care se găsesc noi modificări din partea *developerilor*, testare începând automat, fără a fi nevoie de intervenția omului. Acest concept s-ar putea implementa doar în cazul în care am avea acces la *repository-ul developerilor*, putând astfel să vedem fiecare schimbare din aplicație, în momentul în care acestea au fost realizate. Există multe programe prin care se poate realiza testarea continuă: *Jenkins, Buildbot, TeamCity*, etc.

Un alt aspect important în dezvoltarea ulterioară a aplicației este testarea în paralel. Testarea acelorași teste în diferite *browsere*, sisteme de operare sau chiar setări diferite pot reduce major timpul de execuție asigurând o acoperire mai mare. Există soluții diferite care fac posibil acest lucru, precum: *Selenium Grid, Nose, Jenkins*, etc.

Prin această lucrare am reușit să demonstrez cât este de importantă testarea automată în dezvoltarea unei aplicații *web*, și totodată ce avantaje mari aduce scrierea testelor într-un limbaj natural în comunicarea dintre echipe.

Dezvoltarea acestei lucrări mi-a permis să studiez și să pot pune în practică o nouă abordare în ceea ce privește testarea automată.

## 7 Bibliografie

- [1] B. H. David Gelperin, „The Growth of Software Testing”, 1988.
- [2] „Waterfall Model in Software Testing,” Decembrie 2018. [Interactiv]. Available: <https://www.testingexcellence.com/waterfall-model/>.
- [3] „Agile Testing – Principles, methods & advantages,” Iulie 2018. [Interactiv]. Available: <https://reqtest.com/testing-blog/agile-testing-principles-methods-advantages/>.
- [4] D. C. N. P. Tonella, A Goal-Oriented Software Testing Methodology, Michael LuckLin Padgham, 2007.
- [5] „About ISTQB®,” 2015. [Interactiv]. Available: <https://www.istqb.org/>.
- [6] IEEE Standard Glossary, 1983.
- [7] ISTQB®-Certified, Tester,Basics of Software Testing, Foundation Level Curriculum, 2007.
- [8] ISTQB®- Foundation Level, 2018.
- [9] R. S. C. Itti Hooda, „Software Test Process, Testing Types and Techniques,” în *International Journal of Computer Applications* , 2015.
- [10] T. G. B. G. Elfriede Dustin, Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality, 2009.
- [11] „Automation Testing Vs. Manual Testing: What’s the Difference?,” [Interactiv]. Available: <https://www.guru99.com/difference-automated-vs-manual-testing.html>.
- [12] E. Hechtel, 9 Iulie 2015. [Interactiv]. Available: <https://saucelabs.com/blog/top-10-benefits-of-automated-testing>.
- [13] „Selenium - Web Browser Automation,” Iunie 20119. [Interactiv]. Available: <https://www.seleniumhq.org>.
- [14] „Selenium Suite,” [Interactiv]. Available: <https://seleniumwithjavapython.wordpress.com/selenium/selenium-suite/>.
- [15] B. Muthukadan, „Selenium with Python,” 2018. [Interactiv]. Available: <https://selenium-python.readthedocs.io>.

- [16] „Selenium Locators: Identify Web Elements,” 2019. [Interactiv]. Available: <https://www.softwaretestinghelp.com>.
- [17] „How to Write Effective XPath Selenium Selectors – All Tactics Explained,” Septembrie 2017. [Interactiv]. Available: <https://www.swtestacademy.com/xpath-selenium/>.
- [18] D. H. P. Nishmitha G.C, „Implementation of Page Object Model in Selenium,” 2016.
- [19] GCREDDY, „Page Object Model in Selenium,” 27 Septembrie 2016. [Interactiv]. Available: <https://www.gcreddy.com/2016/09/page-object-model-in-selenium-2.html>.
- [20] „Gherkin Reference,” [Interactiv]. Available: <https://cucumber.io/docs/gherkin/reference/>.
- [21] D. North, „Introducing BDD,” Martie 2006.
- [22] „Python Software Foundation,” [Interactiv]. Available: <https://www.python.org/>.
- [23] „Pycharm,” [Interactiv]. Available: <https://www.jetbrains.com/pycharm/>.
- [24] „ChromeDriver - WebDriver for Chrome,” [Interactiv]. Available: <http://chromedriver.chromium.org/downloads>.
- [25] 2018. [Interactiv]. Available: <https://selenium-python.readthedocs.io/navigating.html>.

## 8 Anexe

Clasa *Base()* din folder-ul „*utils*”:

```

1. import logging
2. from selenium.webdriver.common.action_chains import ActionChains
3. from selenium.webdriver.common.by import By
4. from selenium.webdriver.support.select import Select
5.
6. from utils.data_handler import DataHandler
7. from utils.driver import Driver
8.
9. try:
10.     from selenium import webdriver
11.     from selenium.webdriver.chrome.options import Options
12.     from selenium.common.exceptions import WebDriverException
13.     from selenium.common.exceptions import TimeoutException
14.     from selenium.webdriver.remote.webelement import WebElement
15. except ImportError:
16.     logging.critical("Selenium module is not installed...Exiting program.")
17.     exit(1)
18.
19.
20. ## Base class -helper functionality
21. class Base():
22.     TIMEOUT = 10
23.     instance = None
24.
25.     @classmethod
26.     def get_instance(cls):
27.         if cls.instance is None:
28.             cls.instance = Base()
29.         return cls.instance
30.
31.     def __init__(self):
32.         self.driver = Driver().get_driver()
33.
34.     def navigate_to_url(self, url):
35.         if isinstance(url, str):
36.             self.driver.get(url)
37.         else:
38.             raise TypeError("URL must be a string.")
39.
40.     def exit_browser(self):
41.         self.driver.quit()
42.
43.     def delete_cookies(self):
44.         self.driver.delete_all_cookies()
45.
46.     def move_to_element(self, how, what):
47.         element = self.driver.find_element(by=how, value=what)
48.         actions = ActionChains(self.driver)
49.         actions.move_to_element(element).perform()

```

```

1.  ##Types a string into a speciffic field
2.  def type_into_a_field(self, how, where, what):
3.      element_field = self.driver.find_element(by=how, value=where)
4.
5.      if element_field.get_attribute("value") is not None:
6.          element_field.clear()
7.          element_field.send_keys(str(what))
8.
9.      # Checks if a button is already selected
10. def check_if_button_is_selected(self, how, where):
11.     return self.driver.find_element(by=how, value=where).is_selected()
12.
13. def select_value_from_dropdown_list(self, how, where, value):
14.     select = Select(self.driver.find_element(by=how, value=where))
15.     select.select_by_visible_text(value)
16.
17. def select_radio_button(self, how, where, value):
18.     radio_buttons = self.driver.find_elements(by=how, value=where)
19.     for item in radio_buttons:
20.         if value in item.text and not item.is_selected():
21.             item.click()
22.
23.     # value format needs to be like this: 'dd/mm/yyyy'
24. def type_into_date_picker(self, how, where, value):
25.     data_list = str(value).split('/')
26.     dictionary_data = {'days': None,
27.                        'months': None,
28.                        'years': None
29.                       }
30.
31.     for index, key in enumerate(dictionary_data):
32.         dictionary_data[key] = (data_list[index])
33.
34.     data_elements = self.driver.find_elements(by=how, value=where)
35.
36.     for index, item in enumerate(data_elements):
37.         element = item.find_element(By.ID, str(list(dictionary_data.keys())[index]))
38.         Select(element).select_by_value(data_list[index])
39.
40. def get_account_info_from_csv(self, email):
41.     account_information = {'name': None,
42.                            'address': None,
43.                            'city_state_postal_code': None,
44.                            'country': None,
45.                            'phone': None
46.                           }
47.
48.     account_name = DataHandler().test_data('first_name', email) + " " + Data-
49. Handler().test_data('last_name', email)
50.     account_information['name'] = account_name
51.
52.     account_address = str(DataHandler().test_data('address_1', email))
53.     account_information['address'] = account_address
54.
55.     account_city_state_postal = DataHandler().test_data('city', email) + ", " \
56. + Data-
57. Handler().test_data('state', email) + " " + str(
58.     DataHandler().test_data('postal_code', email))
59.     account_information['city_state_postal_code'] = account_city_state_postal

```

Clasa *Home()* din folder-ul „page\_object”:

```

1. from utils.base import Base
2. from selenium.webdriver import ActionChains
3. from selenium.webdriver.common.by import By
4. from selenium.webdriver.support import expected_conditions as EC
5. from selenium.webdriver.support.ui import WebDriverWait
6.
7. from page_object.checkout import CheckOut
8. from page_object.sing_in import SingIn
9. from utils.driver import web_driver
10. from utils.element import element
11.
12.
13. class Home(Base):
14.     instance = None
15.
16.     def __init__(self):
17.         self.driver = web_driver.get_driver()
18.
19.     @classmethod
20.     def get_instance(cls):
21.         if cls.instance is None:
22.             cls.instance = Home()
23.         return cls.instance
24.
25.     # Class locators
26.     _home_page_locator = "gr__automationpractice_com"
27.     _cookies_locator = "//div[@class='cookies-info-text']"
28.     _sing_in_button_locator = "login"
29.     _sing_out_button_class = 'logout'
30.     _account_name_class = 'account'
31.     _products_container_id = 'homefeatured'
32.     _product_list_class = "product-container"
33.     _shopping_cart_class = "shopping_cart"
34.     _continue_shopping_loca-
35.     tor_xpath = '//*[@id="layer_cart"]/div[1]/div[2]/div[4]/span/span'
36.     _cart_products_class = 'products'
37.     _proceed_to_checkout_xpath = "//span[contains(., 'Proceed to checkout')]"
38.     _check_out_cart_xpath = "//span[contains(., 'Check out')]"
39.     _logo_button_css = '.logo.img-responsive'
40.     _search_field_id = "search_query_top"
41.     _search_results_css = '.product_list.grid.row'
42.     _sort_method_xpath = ".//*[id='productsSortForm']//*[id='selectProductSort']"
43.
44.     @property
45.     def already_sing_in(self):
46.         return element.is_element_present(By.CLASS_NAME, self._sing_out_button_class)
47.
48.     @property
49.     def get_account_name(self):
50.         account_name = self.driver.find_element(By.CLASS_NAME, self._ac-
51.         count_name_class)
52.         return account_name.text
53.
54.     @property
55.     def get_shopping_cart_items(self):
56.         shopping_cart = self.driver.find_element(By.CLASS_NAME, self._shop-
57.         ping_cart_class)
58.         text = shopping_cart.text
59.         for char in text:
60.             if char.isdigit():
61.                 return char

```

```

59.         else:
60.             return text
61.
62.     @property
63.     def get_number_of_items(self):
64.         cart_elements = self.driver.find_element(By.CLASS_NAME, self._cart_products_class).find_elements(By.TAG_NAME,
65.                                                         'dt')
66.         return len(cart_elements)
67.
68.     @property
69.     def get_list_of_products(self):
70.         product_containers_id = self.driver.find_element(By.ID, self._products_container_id)
71.         return product_containers_id.find_elements_by_tag_name("li")
72.
73.     @property
74.     def get_list_of_results_name(self):
75.         products_name = []
76.
77.         product_containers = self.driver.find_element(By.CSS_SELECTOR, self._search_results_css)
78.         product_list = product_containers.find_elements_by_tag_name("li")
79.         for product in product_list:
80.             name = product.text.split('\n')[0]
81.             if name is not "":
82.                 products_name.append(product.text.split('\n')[0])
83.         return products_name
84.
85.     @property
86.     def get_list_of_results_price(self):
87.         products_price = []
88.
89.         product_containers = self.driver.find_element(By.CSS_SELECTOR, self._search_results_css)
90.         product_list = product_containers.find_elements_by_tag_name("li")
91.         for product in product_list:
92.             name = product.text.split('\n')[0]
93.             if name is not "":
94.                 price_text = product.text.split('\n')[1]
95.                 products_price.append(float(price_text.split()[0].split('$')[1]))
96.         return products_price
97.
98.     @property
99.     def get_all_cart_products_details(self):
100.         products_name = []
101.         products_price = []
102.         products_quantity = []
103.

```

```

112.
113.         product_price = float(
114.             cart_info.find_ele-
115.             ment(By.XPATH, ".*[@class='price']").text.split()[0].split('$')[1])
116.         products_price.append(product_price)
117.
118.         product_quantity = int(cart_info.find_ele-
119.             ment(By.XPATH, ".*[@class='quantity']").text)
120.         products_quantity.append(product_quantity)
121.
122.         return products_name, products_price, products_quantity
123.
124.     def search_product(self, product_name):
125.         self.type_into_a_field(By.ID, self._search_field_id, product_name)
126.         self.driver.find_element(By.ID, self._search_field_id).submit()
127.
128.     def add_products_in_cart(self, quantity):
129.         wait = WebDriverWait(self.driver, self.TIMEOUT)
130.
131.         product_containers = self.get_list_of_products
132.
133.         for index, product in enumerate(product_containers):
134.             if index == int(quantity):
135.                 break
136.             i = index + 1
137.             hover = ActionChains(self.driver).move_to_element(product)
138.             hover.perform()
139.
140.             _add_to_cart_locator_xpath = '//*[@id="homefeatured"]/li[%s]/div/div[2]/div[2]/a[1]/span' % i
141.             wait.until(EC.element_to_be_clickable((By.XPATH, _add_to_cart_locator_xpath)))
142.             self.driver.implicitly_wait(2)
143.             self.driver.find_element(By.XPATH, _add_to_cart_locator_xpath).click()
144.
145.             wait.until(EC.element_to_be_clickable((By.XPATH, self._continue_shopping_locator_xpath)))
146.             self.driver.implicitly_wait(2)
147.             self.driver.find_element(By.XPATH, self._continue_shopping_locator_xpath).click()
148.
149.         # Add specific product to the cart by name
150.         # product_name - The name of the desired product
151.         def add_specific_product_in_cart(self, desired_product_name):
152.             wait = WebDriverWait(self.driver, self.TIMEOUT)
153.             product_containers = self.get_list_of_products
154.
155.             for index, product in enumerate(product_containers):
156.                 product_name = product.text.split('\n')[0]
157.                 if desired_product_name == product_name:
158.                     hover = ActionChains(self.driver).move_to_element(product)
159.                     hover.perform()
160.                     i = index + 1
161.                     _add_to_cart_locator_xpath = '//*[@id="homefeatured"]/li[%s]/div/div[2]/div[2]/a[1]/span' % i
162.                     wait.until(EC.element_to_be_clickable((By.XPATH, _add_to_cart_locator_xpath)))
163.                     self.driver.implicitly_wait(2)
164.                     self.driver.find_element(By.XPATH, _add_to_cart_locator_xpath).click()

```



```

163.         self.driver.find_element(By.XPATH, _add_to_cart_locator_xpath).click()
164.         self.driver.find_element(By.XPATH, self._continue_shopping_locator_xpath).click()
165.         break
166.     else:
167.         raise Exception("Product '%s' not found" % desired_product_name)
168.
169.     def go_to_homescreen(self):
170.         self.driver.find_element(By.CSS_SELECTOR, self._logo_button_css).click()
171.
172.     def clickSingInButton(self):
173.         sing_in_button = self.driver.find_element_by_class_name(self._sing_in_button_locator)
174.         sing_in_button.click()
175.         return SingIn()
176.
177.     def clickSingOutButton(self):
178.         self.driver.implicitly_wait(10)
179.         sing_out_button = self.driver.find_element(By.CLASS_NAME, self._sing_out_button_class)
180.         sing_out_button.click()
181.
182.     def clickCheckOutButton(self):
183.         self.move_to_cart()
184.         self.driver.find_element(By.XPATH, self._check_out_cart_xpath).click()
185.         return CheckOut()
186.
187.     def move_to_cart(self):
188.         self.move_to_element(By.XPATH, "//b[contains(., 'Cart')]")
189.
190.     def clickOnCart(self):
191.         self.driver.find_element(By.XPATH, "//b[contains(., 'Cart')]").click()
192.         return CheckOut()
193.
194.     def choose_sort_method(self, method):
195.         self.select_value_from_dropdown_list(By.XPATH, self._sort_method_xpath, method)
196.
197.
198.     home = Home.get_instance()

```

Fișierul *shopping\_steps.py* din folder-ul „features”:

```

1. from behave import *
2. from page_object.home import home
3. from page_object.checkout import checkout, address, shipping, payment
4. from nose.tools import assert_equal
5.
6.
7. @given("I add {quantity} products to the cart")
8. def step_impl(context, quantity):
9.     home.go_to_homescreen()
10.    home.add_products_in_cart(quantity)
11.
12.
13. @then("I check if the expected quantity: '{expected_quantity}' added to the cart is correct")
14. def step_impl(context, expected_quantity):
15.     current_quantity = home.get_shopping_cart_items
16.     assert_equal(current_quantity, expected_quantity, "The expected quantity added to the cart is correct")

```

```

18. @when("I click on check out button")
19. def step_impl(context):
20.     home.click_on_cart()
21.
22. @then("I check if all product details from the cart are the same with prod-
    ucts from cart summary")
23. def step_impl(context):
24.     home.move_to_cart()
25.     expected_product_details = home.get_all_cart_products_details
26.     summary_cart_product_details = checkout.get_all_cart_summary_products_details
27.
28.     assert_equal(expected_product_details[0], summary_cart_product_de-
    tails[0], 'The product name is correct')
29.     assert_equal(expected_product_details[1], summary_cart_product_de-
    tails[1], 'The product price is correct')
30.     assert_equal(expected_product_details[2], summary_cart_product_de-
    tails[2], 'The product quantity is correct')
31.
32.
33. @given("I add product '{specific_items}' to the cart")
34. def step_impl(context, specific_items):
35.     home.go_to_homescreen()
36.     items = specific_items.split(',')
37.
38.     for specific_item in items:
39.         home.add_specific_product_in_cart(specific_item)
40.
41. @when("I go to delivery address")
42. def step_impl(context):
43.     home.click_check_out_button()
44.     checkout.click_on_check_out()
45.
46. @then("I check delivery address for account: '{email}'")
47. def step_impl(context, email):
48.     address.verify_account_information(email)
49.
50. @when("I choose to pay via {payment_method}")
51. def step_impl(context, payment_method):
52.     payment.choose_payment_method(payment_method)
53.
54. @when("I proceed to checkout")
55. def step_impl(context):
56.     context.execute_steps(u"""
57.         when I go to delivery address
58.         """)
59.     checkout.click_on_check_out()
60.     shipping.agree_to_the_terms()
61.     checkout.click_on_check_out()
62.
63.
64. @then("I should be able to submit my order")
65. def step_impl(context):
66.     payment.click_confirm_order()
67.     assert_equal(True, payment.is_order_complete, 'The order is complete with cho-
    sen payment method')
68.
69. @then("The confirmation of the {payment_method} payment should be displayed")
70. def step_impl(context, payment_method):
71.     assert payment_method.upper() in payment.get_chosen_payment_method.upper()

```

```
73.  
74. @step("I check if total amount is correct")  
75. def step_impl(context):  
76.     products_total_price = checkout.calculate_products_total_price()  
77.     shipping_tax = checkout.get_shipping_tax  
78.     expected_total_price = products_total_price + shipping_tax  
79.     assert_equal(expected_total_price, checkout.get_total_amount, "The to-  
    tal amount:%f is displayed correct"  
80.                  % expected_total_price)
```