



**Departamentul Automatică și Informatică Industrială**

**Facultatea Automatică și Calculatoare  
Universitatea POLITEHNICA din București**



# **Proiect TSAC**

## **Testare în Junit5**

**Saioc Marius Andrei**

**2024**

## Contents

Introducere.....	2
Tipuri de testare software .....	3
Descrierea aplicației.....	4
JUnit 5 .....	6
Planul de testare .....	7
Testarea aplicației .....	9
Concluzii .....	15
Bibliografie .....	16

## Introducere

Testarea software-ului este o fază critică în ciclul de dezvoltare a software-ului, având ca scop asigurarea faptului că aplicațiile sunt fiabile, sigure și funcționează conform așteptărilor. Aceasta implică verificarea și validarea software-ului pentru a detecta erorile, a asigura conformitatea cu cerințele specificate și a îmbunătăți calitatea generală a produsului. Acest proces este esențial nu numai pentru a identifica defectele, ci și pentru a îmbunătăți eficiența, acuratețea și capacitatea de utilizare a software-ului.

Istoria testării software-ului datează încă din primele zile ale informaticii. Prima lucrare software recunoscută a fost scrisă de Tom Kilburn la 21 iunie 1948, la Universitatea din Manchester, Anglia. Acest program efectua calcule matematice folosind instrucțiuni de cod mașină. În această perioadă, depanarea era principala metodă de testare, concentrându-se pe identificarea și remedierea erorilor din cod.

Pe măsură ce complexitatea software-ului a crescut, a apărut necesitatea unor metodologii de testare mai structurate. Până în anii 1980, testarea a evoluat dincolo de simpla depanare pentru a cuprinde un proces mai amplu de asigurare a calității integrat în ciclul de producție al software-ului. Această schimbare a confirmat necesitatea de a evalua software-ul în contexte reale pentru a se asigura că acesta îndeplinește așteptările utilizatorilor și funcționează corect în diferite condiții [1].

Testarea software-ului este indispensabilă din mai multe motive:

1. **Detectarea timpurie a erorilor:** Identificarea defectelor la începutul procesului de dezvoltare permite remedierea în timp util, reducând costul și efortul necesar pentru rezolvarea ulterioară a problemelor.
2. **Îmbunătățirea calității:** Testarea dezvăluie imperfecțiunile din software, permițând dezvoltatorilor să îmbunătățească calitatea produsului înainte de lansare.
3. **Satisfacția clienților:** Un software fiabil, sigur și performant îndeplinește în mod eficient cerințele utilizatorilor, ceea ce duce la creșterea satisfacției clienților.
4. **Scalabilitate:** Testarea, în special testarea nefuncțională, ajută la identificarea problemelor de scalabilitate și asigură faptul că software-ul poate face față sarcinilor așteptate.
5. **Eficiența costurilor și a timpului:** Testarea regulată pe tot parcursul dezvoltării previne reparațiile costisitoare după lansare și asigură un proces de dezvoltare mai ușor.

Consecințele unei testări inadecvate pot fi grave, după cum o demonstrează incidente istorice, cum ar fi defecțiunea de radioterapie Therac-25 din 1985, care a dus la moartea pacienților, și accidentul Airbus A300 al companiei China Airlines din 1994, în urma căruia 264 de persoane și-au pierdut viața din cauza unei erori software [2].

## Tipuri de testare software

Testarea software-ului este în general clasificată în trei tipuri principale:

- **Testarea funcțională:** Acest tip verifică dacă software-ul își îndeplinește corect funcțiile prevăzute. Acesta include diverse teste, cum ar fi testarea unitară, testarea de integrare, testarea sistemului și testarea de fum.
- **Testarea nefuncțională:** Acest tip evaluează aspecte precum performanța, scalabilitatea și utilizabilitatea. Acesta include teste de performanță, teste de stres și teste de utilizare.
- **Testarea de întreținere:** Aceasta implică testarea actualizărilor și modificărilor software pentru a se asigura că noile schimbări nu afectează negativ funcționalitatea existentă. Testarea de regresie este o practică obișnuită aici.

În plus, testarea software poate fi împărțită în testare manuală și testare automată:

- **Testarea manuală:** Testerii execută manual cazurile de testare fără a utiliza instrumente de automatizare. Această abordare este potrivită pentru testarea exploratorie, în cazul în care percepția umană este crucială.
- **Testarea automatizată:** Testerii utilizează scripturi și instrumente pentru a automatiza sarcinile repetitive de testare, sporind eficiența și acoperirea. Automatizarea este deosebit de utilă pentru testarea de regresie, de încărcare și de performanță.

Testarea software-ului se desfășoară la diferite niveluri pentru a asigura o acoperire cuprinzătoare:

- **Testarea unitară:** Testarea componentelor sau a unităților individuale ale software-ului pentru a se asigura că acestea funcționează conform destinației.
- **Testarea de integrare:** Testarea interacțiunilor dintre unitățile sau componentele integrate pentru a identifica defectele de interfață.
- **Testarea sistemului:** Testarea sistemului software complet și integrat pentru a verifica dacă acesta îndeplinește cerințele specificate.
- **Testarea de acceptanță:** Testarea acceptanței sistemului pentru a se asigura că acesta îndeplinește cerințele de afaceri și că este pregătit pentru implementare.

**White Box Testing** implică testarea structurilor interne sau a mecanismelor de lucru ale unei aplicații. În cadrul testării de tip „white box”, testerul are cunoștințe despre logica internă a sistemului. Această metodă este utilă pentru verificarea fluxului de intrare-ieșire prin intermediul aplicației și verificarea structurilor interne, cum ar fi condițiile, buclele și fluxul de date.

**Black Box Testing** diferă de white box testing prin faptul că se concentrează pe comportamentul extern al software-ului. Testerul nu trebuie să cunoască funcționarea internă a aplicației. Testarea de tip black box are ca scop testarea software-ului în raport cu specificațiile sale. Acest tip de testare este de obicei efectuat de către testerii QA.

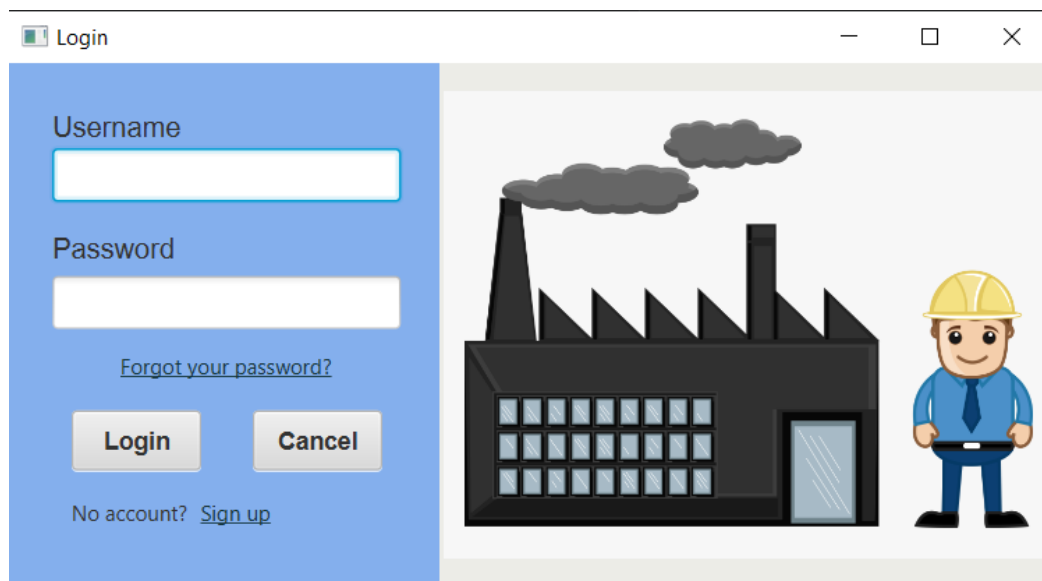
În cadrul proiectului, accentul a căzut pe testarea unitară de tip funcțional. Având la îndemână structura codului sursă al aplicației, s-au implementat teste unitare pentru verificarea funcționalității diferitelor elemente.

**Testarea unitară** este un aspect fundamental al dezvoltării de software care se concentrează pe validarea celor mai mici părți testabile ale unei aplicații, denumite adesea unități [3]. Aceste unități pot fi funcții, metode sau caracteristici individuale care constituie elementele de bază ale unui program software. Scopul principal al testării unităților este de a se asigura că fiecare unitate de cod funcționează așa cum a fost prevăzut, respectând proiectul și cerințele specificate.

## Descrierea aplicației

Aplicația testată este un parser XML conceput pentru a gestiona și analiza datele referitoare la accidente din fabrici. Funcția principală a acestei aplicații este de a oferi utilizatorilor o vizualizare detaliată a accidentelor care au avut loc într-o fabrică, permițând o analiză și o înțelegere cuprinzătoare a incidentelor. În plus, aplicația include o funcție de autentificare a utilizatorilor, permițându-le acestora să se conecteze și să acceseze datele în siguranță.




La inițializare, aplicația afișează o fereastră de autentificare, oferind utilizatorilor opțiunea de a se autentifica sau de a crea un cont nou. Procesul de conectare implică validarea numelui de utilizator și a parolei introduse în raport cu datele stocate într-un fișier XML numit users.xml. În cazul în care datele de identificare ale utilizatorului sunt corecte, aplicația trece la pagina principală, unde pot fi accesate datele privind accidentele. În cazul în care credențialele sunt incorecte, se afișează un mesaj.



Aplicația acceptă înregistrarea utilizatorilor, permițând noilor utilizatori să își creeze un cont prin furnizarea unui nume de utilizator, a unei parole și a unei adrese de e-mail. Procesul de înregistrare include verificări de validare pentru a se asigura că numele de utilizator este unic, că parola are o lungime adecvată și că ambele câmpuri de parolă se potrivesc. În cazul în care informațiile furnizate trec toate verificările, datele noului utilizator sunt adăugate la fișierul users.xml, iar utilizatorul este notificat cu privire la crearea cu succes a contului.

Odată conectați, utilizatorii pot interoga aplicația pentru a vizualiza datele privind accidentele. Datele sunt stocate într-un alt fișier XML numit fabrici.XML, care conține informații detaliate despre fabrici, accidente și angajații implicați. Utilizatorii pot căuta accidente după numele fabricii sau pot vizualiza toate accidentele dacă nu este furnizat un nume specific al fabricii. Aplicația analizează apoi fișierul XML pentru a prelua și afișa datele relevante privind accidentele, inclusiv numărul accidentului, victimele, costurile pagubelor și angajații implicați.

Baza de date accidente



Lista angajaților din fabrica

Fabrica: Arctic  
Oprea Alex 5000 lei  
Buti Carina 4000 lei  
Dinescu Nicolae 3800 lei  
Tunaru Melania 7800 lei

Fabrica: Dacia  
Tabacu Ionut 6600 lei  
Gherban Gabriel 4200 lei  
Vasilica Florentina 5280 lei

Fabrica: Fratii Schiel  
Calin Madalina 5500 lei  
Dirjan Zoe 4900 lei

Lista accidentelor din fabrica

Fabrica: Arctic  
Numar accident: 5  
Victime: da  
Valoare daune: 10000 lei  
Angajati implicati:  
Buti Carina 1 saptamani refacere  
Dinescu Nicolae 10 saptamani refacere

Gravitatea accidentelor

Numar: 5  
Victime: da  
Daune: 10000 lei

Numar: 4  
Victime: da  
Daune: 9999 lei

Aplicația dispune, de asemenea, de un meniu dropdown care le permite utilizatorilor să filtreze accidentele în funcție de gravitate - clasificate ca „usor”, „mediu” și „grav”. Gravitatea este determinată pe baza unor reguli predefinite în cadrul fișierului XML, care evaluează costurile pagubelor și timpul de recuperare pentru angajații implicați în accidente. În funcție de gravitatea selectată, aplicația afișează detaliile corespunzătoare ale accidentului, ajutând și mai mult utilizatorii în analiza lor.

## JUnit 5

Aplicația a fost dezvoltată în Java cu ajutorul IntelliJ IDEA, un mediu de dezvoltare integrat (IDE) popular, cunoscut pentru caracteristicile sale robuste care simplifică dezvoltarea Java. Pentru testarea unitară, s-a folosit JUnit, un cadru de testare utilizat pe scară largă în ecosistemul Java. Având în vedere capacitățile și îmbunătățirile aduse de JUnit 5, acesta a fost alegerea logică pentru a asigura calitatea testării.

JUnit 5 este cea mai recentă versiune a cadrului de testare JUnit, care oferă îmbunătățiri semnificative față de JUnit 4. Acesta este format din trei module principale [4]:

- Platforma JUnit
- JUnit Jupiter
- JUnit Vintage

### JUnit Platform

În testarea software, cazurile de testare sunt concepute pentru a verifica calitatea și comportamentul aplicațiilor în diferite condiții. Platforma JUnit oferă mediul necesar pentru rularea acestor cazuri de testare pe mașina virtuală Java (JVM). Acest lucru permite un mecanism de lansare standardizat între diferite IDE-uri și instrumente de compilare.

### JUnit Jupiter

JUnit Jupiter este modulul de bază pentru scrierea testelor în JUnit 5. Acesta introduce noi tehnici de programare și adnotări care facilitează dezvoltarea mai robustă și mai flexibilă a cazurilor de testare. Caracteristicile cheie includ:

- Adnotări: Adnotări precum `@Test`, `@BeforeEach`, `@AfterEach`, `@BeforeAll` și `@AfterAll` sunt utilizate pentru a defini metode de testare și callback-uri de ciclu de viață.
- Metode de testare a ciclului de viață: Aceste metode sunt executate în anumite momente din timpul ciclului de viață al testului, cum ar fi înainte sau după fiecare test, sau o dată înainte sau după toate testele dintr-o clasă.

### JUnit Vintage

JUnit Vintage permite ca testele scrise în JUnit 3 și JUnit 4 să fie executate pe platforma JUnit 5. Această compatibilitate retroactivă este crucială pentru proiectele care migrează la JUnit 5, păstrând în același timp cazurile de testare vechi.

### Caracteristici cheie ale JUnit 5

- Adnotări:
  - `@Test`: Marchează o metodă ca metodă de testare.

- @BeforeEach: Se execută înainte de fiecare metodă de testare.
- @AfterEach: Se execută după fiecare metodă de testare.
- @BeforeAll: Se execută o dată înaintea tuturor metodelor de testare din clasă.
- @AfterAll: Se execută o dată după toate metodele de testare din clasă.
- Assertions:
  - assertEquals(expected, actual): Verifică dacă două valori sunt egale.
  - assertTrue(condition): Verifică dacă o condiție este adevărată.
  - assertNotNull(object): Verifică dacă un obiect nu este nul.
- Assumptions:
  - assumeTrue(condition): Ignoră un test dacă condiția nu este adevărată.
  - assumingThat(condiție, executabil): Execută un bloc de cod în cazul în care condiția este adevărată.
- Teste parametrizate:
  - @ParameterizedTest: Execută același test cu parametri diferiți.
  - @ValueSource: Furnizează o sursă de valori pentru testul parametrizat.
- Teste dinamice:
  - @TestFactory: Generează teste dinamice în timpul execuției.
- Etichetare și filtrare:
  - @Tag: Etichetează testele pentru clasificare.
  - Filtrarea testelor permite rularea unor teste etichetate specifice.

## Planul de testare

Înainte de inițierea fazei de testare, a fost creat un plan detaliat pentru a asigura o acoperire completă a funcționalităților aplicației. Planul s-a axat pe validarea unor aspecte critice, cum ar fi funcționalitatea de conectare și caracteristicile de analiză și afișare XML. Fiecare funcționalitate a fost asociată cu cazuri de testare specifice pentru a verifica sistematic comportamentele așteptate.

## Funcționalitatea de autentificare

- **TC01: Autentificarea cu credențiale valide.**
  - Testează capacitatea aplicației de a autentifica utilizatorii cu numele de utilizator și parola corecte.
- **TC02: Autentificarea cu credențiale invalide.**
  - Se asigură că aplicația refuză accesul și afișează un mesaj de eroare atunci când sunt introduse credențiale incorecte.
- **TC03: Câmpuri de nume de utilizator sau parolă goale.**
  - o Verifică dacă aplicația afișează o eroare atunci când câmpurile de nume de utilizator sau parolă sunt lăsate goale.
- **TC04: Crearea unui utilizator nou.**
  - Validează faptul că un nou utilizator poate fi creat cu succes cu informații corecte și complete.
- **TC05: Password mismatch.**



- Asigură afișarea unui mesaj de eroare în cazul în care câmpurile de parolă și de confirmare a parolei nu se potrivesc în timpul creării utilizatorului.
- **TC06: Nume de utilizator indisponibil.**
  - Verifică dacă aplicația împiedică crearea unui nou utilizator cu un nume de utilizator care există deja.
- **TC07: Parolă slabă.**
  - Se validează faptul că aplicația afișează un mesaj de eroare în cazul în care este furnizată o parolă slabă în timpul creării utilizatorului.

### **Afișarea angajaților:**

- **TC08: Nume valid al fabricii - angajați.**
  - Se asigură că aplicația afișează corect toți angajații atunci când se introduce un nume de fabrică valid.
- **TC09: Fabrică inexistentă - angajați.**
  - Se verifică dacă aplicația afișează un mesaj relevant atunci când se introduce un nume de fabrică care nu există.
- **TC10: Afișarea tuturor angajaților.**
  - Validarea faptului că introducerea unui șir gol afișează angajații din toate fabricile.

### **Afișarea accidentelor:**

- **TC11: Nume valid al fabricii - accidente.**
  - Se asigură că toate accidentele sunt afișate pentru un anumit nume de fabrică valid.
- **TC12: Fabrică inexistentă - accidente.**
  - Se verifică dacă se afișează un mesaj corespunzător atunci când se introduce un nume de fabrică inexistent.
- **TC13: Fabrică fără accidente.**
  - Se validează faptul că se afișează un mesaj care indică lipsa accidentelor pentru o fabrică fără incidente înregistrate.
- **TC14: Afișarea tuturor accidentelor.**
  - Se asigură că introducerea unui șir gol afișează accidentele din toate fabricile.
- **TC15: Verificarea filtrării accidentelor în funcție de gravitate (3 cazuri: ușor, mediu, grav).**
  - Testează capacitatea aplicației de a filtra și afișa accidentele în funcție de gravitatea lor.

Pentru a evita perturbarea datelor existente și pentru a ne asigura că testele nu lasă în urmă date irelevante sau corupte, a fost creat un set de date special pentru testare. Acest set de date a fost conceput pentru a fi temporar, ceea ce înseamnă că va fi șters după fiecare testare.

## Testarea aplicației

Fiind construită folosind JavaFX, clasele aplicației, numite controlleri, corespund ferestrelor UI. Astfel, pentru testare s-au implementat două clase de test: clasa `LoginControllerTest` pentru testarea funcționalităților ferestrei de autentificare, respectiv clasa `Page1ControllerTest` pentru testarea funcționalităților ferestrei principale.

În IntelliJ IDEA, folosind JUnit, se poate rula fiecare caz de test în parte (cazul de test este metoda marcată prin adnotarea `@Test`, având un buton de rulare în dreptul acesteia), sau se poate rula întreaga clasă. La rularea întregii clase, fiecare caz este rulat secvențial în ordinea apariției în cod. Se vor rula toate testele, dar testarea clasei va eșua dacă oricare dintre cazurile de test eșuează.

Am folosit o structură de clase nested pentru organizare și claritate, extinzând `ApplicationTest` pentru a integra capabilitățile de testare JavaFX. Fiecare test a fost conceput pentru a gestiona interacțiunile cu utilizatorul și a valida rezultatele în raport cu rezultatele așteptate.

```
@Nested
class LoginControllerTest extends ApplicationTest {

    private loginController loginController; 50 usages
    private static final String USERS_FILE_PATH = "users.xml"; 4 usages
    private static final String USERS_BACKUP_FILE_PATH = "users_backup.xml"; 3 usages
```

- **@Nested:** Această adnotare este utilizată pentru a crea clase de test nested, ceea ce ajută la organizarea logică a testelor.
- **ApplicationTest:** O clasă din TestFX care permite testarea aplicațiilor JavaFX.

Pentru a asigura că testele nu se afectează reciproc și pentru a menține integritatea datelor, s-a efectuat o copie de rezervă a datelor înainte de fiecare test, pentru a fi restaurate după testare.

```
@BeforeEach
public void setUp() throws Exception {
    // Backup the original users.xml file
    Files.copy(Paths.get(USERS_FILE_PATH),
        Paths.get(USERS_BACKUP_FILE_PATH));

    // Initialize the controller and the scene
    loginController = new loginController();
    interact(() -> {
        try {
            FXMLLoader loader = new
FXMLLoader(getClass().getResource("/sample/login.fxml"));
            Parent root = loader.load();
```

```

        // Get the controller
        loginController = loader.getController();

        Scene scene = new Scene(root, 600, 300);
        Stage FirstPageStage = new Stage();
        FirstPageStage.setTitle("Baza de date accidente");
        FirstPageStage.setScene(scene);
        FirstPageStage.show();

    } catch (IOException e) {
        e.printStackTrace();
    }
});

// Prepare a test XML file for user data
prepareTestUserData();
}

@AfterEach
public void tearDown() throws Exception {
    // Restore the original users.xml file
    Files.copy(Paths.get(USERS_BACKUP_FILE_PATH), Paths.get(USERS_FILE_PATH),
        java.nio.file.StandardCopyOption.REPLACE_EXISTING);
    Files.delete(Paths.get(USERS_BACKUP_FILE_PATH));
}

```

- ❑ **@BeforeEach:** Această adnotare indică faptul că metoda adnotată trebuie să fie executată înaintea fiecărei metode de testare din clasa de testare curentă.
- ❑ **@AfterEach:** Această adnotare indică faptul că metoda adnotată trebuie să fie executată după fiecare metodă de testare din clasa de testare curentă.
- ❑ **interact:** O metodă din TestFX utilizată pentru a rula acțiuni pe JavaFX Application Thread (JFXAT). Acest lucru este esențial pentru modificarea componentelor UI în teste.

## Cazuri de testare

### 1. Testarea autentificării valide

Acest test verifică dacă un utilizator se poate autentifica cu credențiale valide. Specific unei abordări white box, acest test profită de o variabilă din cadrul aplicației, care devine true în urma validării datelor de autentificare.

```

@Test
void testValidLogin() throws SQLException, NoSuchAlgorithmException,
IOException {
    interact(() -> {
        loginController.login_usernameTextField.setText("validUser");
        loginController.login_enterPasswordField.setText("validPass");
        try {
            loginController.loginButtonOnAction(null);
        } catch (SQLException | NoSuchAlgorithmException | IOException e) {

```

```

        throw new RuntimeException(e);
    }
});
// Check if the userFound variable is true
assertTrue(loginController.userFound, "User should be found with valid
credentials");
}

```

- ☐ **@Test:** Această adnotare marchează o metodă ca metodă de testare.
- ☐ **assertTrue:** Această afirmație verifică dacă o condiție este adevărată.

## 2. Test nume de utilizator corect, parolă incorectă

Acest test verifică funcționalitatea de conectare atunci când este furnizat un nume de utilizator corect, dar o parolă incorectă.

```

@Test
void testUsernameCorrectPasswordIncorrect() throws SQLException,
NoSuchAlgorithmException, IOException {
    interact(() -> {
        loginController.login_usernameTextField.setText("validUser");
        loginController.login_enterPasswordField.setText("invalidPass");
        try {
            loginController.loginButtonOnAction(null);
        } catch (SQLException | NoSuchAlgorithmException | IOException e) {
            throw new RuntimeException(e);
        }
    });
    // Check if the login message label indicates incorrect credentials
    assertEquals("Incorrect username or password",
loginController.loginMessageLabel.getText());
}

```

- **assertEquals:** Această afirmație verifică dacă două valori sunt egale.

## 3. Testul de creare a unui utilizator cu detalii valide

Acest test validează crearea unui nou utilizator cu detalii corecte și verifică autentificarea ulterioară cu noile credențiale.

```

@Test
void testCreateUserWithValidDetails() throws SQLException,
NoSuchAlgorithmException {
    interact(() -> {
        loginController.create_usernameTextField.setText("newUser");
        loginController.create_passwordTextField.setText("strongPassword");
        loginController.repeat_passwordTextField.setText("strongPassword");
        loginController.create_emailTextField.setText("newuser@example.com");

        // Simulate clicking the create button
        try {
            loginController.createButtonOnAction(null);
        } catch (SQLException | NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    });
}

```

```

    });
    // Check if the account creation was successful
    assertEquals("Account created successfully.",
loginController.createMessageLabel.getText());

    // Verify the new user data in the XML file
    boolean userFound = false;
    try {
        DocumentBuilderFactory dbFactory =
DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(new File(USERS_FILE_PATH));
        doc.getDocumentElement().normalize();

        NodeList userList = doc.getElementsByTagName("user");

        for (int i = 0; i < userList.getLength(); i++) {
            Node userNode = userList.item(i);
            if (userNode.getNodeType() == Node.ELEMENT_NODE) {
                Element userElement = (Element) userNode;
                String username =
userElement.getElementsByTagName("username").item(0).getTextContent();
                String password =
userElement.getElementsByTagName("password").item(0).getTextContent();
                if ("newUser".equals(username) &&
"strongPassword".equals(password)) {
                    userFound = true;
                    break;
                }
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    assertTrue(userFound, "New user should be found in the XML file");

    // Test login with the new credentials
    interact(() -> {
        loginController.login_usernameTextField.setText("newUser");
        loginController.login_enterPasswordField.setText("strongPassword");
        try {
            loginController.loginButtonOnAction(null);
        } catch (SQLException | NoSuchAlgorithmException | IOException e) {
            throw new RuntimeException(e);
        }
    });
    // Check if the new user login is successful
    assertTrue(loginController.userFound, "New user should be able to log in
with valid credentials");
}

```

Pe langa cele prezentate, au fost implementate si urmatoarele cazuri de test pentru functionalitatile de autentificare si creare de utilizator nou:

- testPasswordCorrectUsernameIncorrect()

- testInvalidLogin() – username și parolă incorecte
- testEmptyFields()
- testPasswordMismatchDuringUserCreation()
- testExistingUsernameDuringUserCreation()
- testWeakPasswordDuringUserCreation()

În cazul clasei de test Page1ControllerTest s-a folosit o abordare identică, folosind @BeforeEach și @AfterEach, fiecare test rulează pe un set de date special creat pentru testare, astfel încât la finalul testării datele aplicației nu vor fi alterate și nu vor rămâne date reziduale în sistem.

#### 4. Afișare angajați pentru o fabrică existentă

Acest test verifică dacă, atunci când se introduce un nume de fabrică valid, aplicația afișează corect toți angajații asociați cu acea fabrică.

```
@Test
void testDisplayAllEmployeesForValidFactory() {
    interact() -> {
        page1Controller.listaAngajatiFabricaTextField.setText("Arctic");
        page1Controller.listaAngajatiFabricaTextFieldOnAction(null);
        String expectedOutput = ""
            Fabrica: Arctic
            Oprea Alex 5000 lei
            Buti Carina 4000 lei
            Dinescu Nicolae 3800 lei
            Tunaru Melania 7800 lei
            "";
        assertEquals(expectedOutput.trim(),
            page1Controller.listaAngajatiFabricaTextArea.getText().trim());
    });
}
```

#### 5. Testul mesajului de afișare pentru fabrica inexistentă (angajați)

Acest test verifică dacă se afișează un mesaj de eroare corespunzător atunci când se introduce un nume de fabrică inexistent în funcționalitatea de afișare a angajaților.

```
@Test
void testDisplayMessageForNonExistingFactoryAngajati() {
    interact() -> {
        page1Controller.listaAngajatiFabricaTextField.setText("Another");

        page1Controller.listaAngajatiFabricaTextFieldOnAction(null);

        String expectedOutput = "Nu exista o fabrica numita Another";
        assertEquals(expectedOutput.trim(),
            page1Controller.listaAngajatiFabricaTextArea.getText().trim());
    });
}
```

## 6. Testul mesajului de afișare pentru fabrica fără accidente

Acest test garantează că, atunci când se interoghează o fabrică fără accidente înregistrate, aplicația afișează un mesaj corect care indică absența accidentelor.

```
@Test
void testDisplayMessageForFactoryWithNoAccidents() {
    Platform.runLater(() -> {
        page1Controller.listaAccidenteFabricaTextField.setText("Fratii
        Schiel");
        page1Controller.listaAccidenteFabricaTextFieldOnAction(null);
        String expectedOutput = "Fabrica Fratii Schiel nu are inregistrate
        accidente";
        assertEquals(expectedOutput.trim(),
        page1Controller.listaAccidenteFabricaTextArea.getText().trim());
    });
}
```

Pe lângă testele prezentate, au fost implementate și restul cazurilor de testare din plan:

- testDisplayAllEmployeesForEmptyFactoryName() – afișarea tuturor angajaților
- testDisplayAllAccidentsForValidFactory() – afișarea accidentelor dintr-o anumită fabrică
- testDisplayMessageForNonExistingFactoryAccidente() – mesaj când fabrica nu există
- testDisplayAllAccidentsForEmptyFactoryName() – afișarea tuturor accidentelor

Afișarea accidentelor în funcție de gradul de severitate:

- testComboBoxUsor()
- testComboBoxMediu()
- testComboBoxGrav()

Datele de test au fost scrise pentru a testa toate funcționalitățile aplicației, dar în special pentru a testa valorile limită în cazul determinării gradului de severitate al accidentelor.

În cadrul aplicației, gravitatea accidentelor este determinată după următoarele reguli:

- în funcție de daunele materiale sau cea mai mare perioadă de concediu medical primită de un angajat implicat:

Valoare daune materiale [lei]	Concediu medical [săptămâni]	Gravitate accident
[0-4999]	[0-2]	ușor
[5000-9999]	[3-9]	mediu
>= 10 000	>= 10	grav

Gravitate unui accident se determină în funcție de gravitatea maximă. (de exemplu: un accident cu daune de 2500 lei, dar care a provocat 5 săptămâni de concediu medical pentru un angajat implicat va fi un accident mediu). Astfel datele de testare au fost scrise folosind aceste valori limită pentru a asigura funcționarea corectă a aplicației. Pentru valori negative, nu se face atribuirea gravității accidentului, datele fiind greșite. Pentru că aplicația nu se ocupă cu introducerea datelor, ci cu parsarea și afișarea lor, existența unor date eronate în baza de date este posibilă și acceptată, afișarea lor făcându-se normal.

## Concluzii

Proiectul a abordat testarea unei aplicații monolitice dezvoltate cu JavaFX pentru gestionarea unor datelor critice precum angajații și accidente din fabrici. Prin planul de testare am asigurat acoperirea completă a funcționalităților esențiale, validând diverse scenarii de utilizare.

Funcționalitatea de autentificare a fost riguros testată, acoperind cazuri de utilizare valide și invalide, asigurând astfel securitatea și fiabilitatea procesului de login. De asemenea, afișarea datelor despre angajați și accidente a fost verificată în multiple scenarii, garantând corectitudinea și relevanța informațiilor afișate utilizatorilor finali.

Fiind o aplicație monolitică, orice modificare adusă unei componente poate avea implicații asupra întregului sistem. De aceea, este esențial ca toate testele să fie rulate din nou după fiecare modificare pentru a ne asigura că funcționalitățile existente nu sunt afectate negativ. Această practică de rerulare completă a testelor previne apariția bug-urilor și menține integritatea aplicației, garantând o experiență consistentă și fără erori pentru utilizatori.

Proiectul a beneficiat de un set de date special creat pentru testare, eliminând riscul ca datele de test să afecteze datele reale. Metodologia de testare, incluzând backup-uri și restaurări ale datelor înainte și după fiecare test, a fost crucială pentru a păstra integritatea datelor și a asigura izolarea testelor.

În concluzie, abordarea sistematică de testare și măsurile riguroase de asigurare a calității implementate au demonstrat că aplicația este robustă și pregătită pentru utilizare reală, oferind încredere în capacitatea sa de a gestiona corect și eficient datele utilizatorilor.



## Bibliografie

- [1] *What is software testing?* | IBM. (n.d.). <https://www.ibm.com/topics/software-testing>
- [2] GeeksforGeeks. (2024, May 24). *What is Software Testing?* GeeksforGeeks. <https://www.geeksforgeeks.org/software-testing-basics/>
- [3] Sahu, N. (2022, January 4). Major types of software testing - Niket sahu - medium. *Medium*. <https://medium.com/@niketsahu/types-of-software-testing-37d7cfeae3fb>
- [4] GeeksforGeeks. (2024, January 3). *Introduction to JUnit 5*. GeeksforGeeks. <https://www.geeksforgeeks.org/introduction-to-junit-5/>