Refactorització i Proves Unitàries d'un sistema bancari bàsic

Resultats d'aprenentatge:

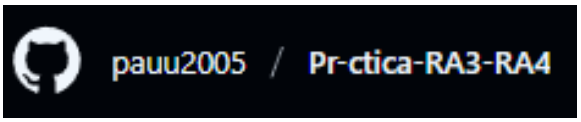
RA3. Verifica el funcionament de programes dissenyant i realitzant proves. **RA4.** Optimitza codi emprant les eines disponibles en l'entorn de desenvolupament.

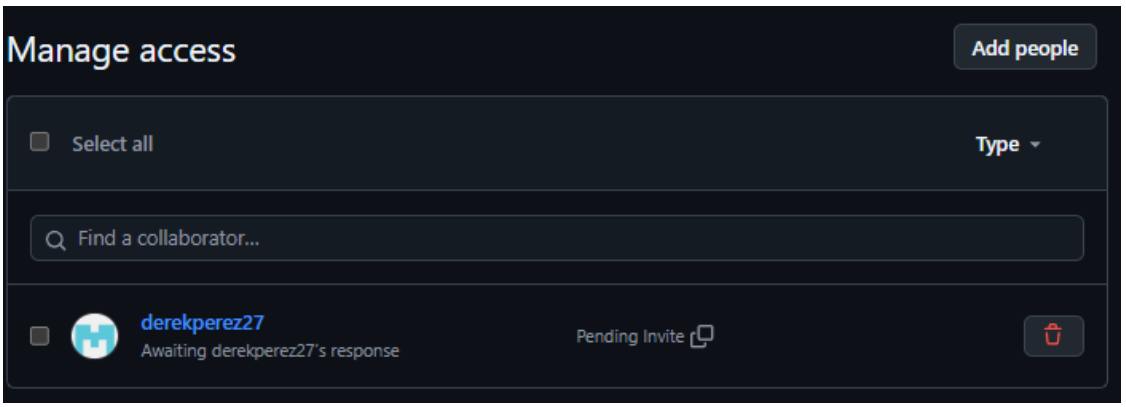
Objectiu de la pràctica:

- Aprendre a depurar i analitzar el codi per comprendre com funcionen les variables i la seva interacció en el programa.
- Refactoritzar el codi separant les responsabilitats mitjançant la tècnica de Extract Method.
- Escriure proves unitàries per assegurar el correcte funcionament dels mètodes. • Utilitzar GitHub per gestionar el control de versions mitjançant branques.

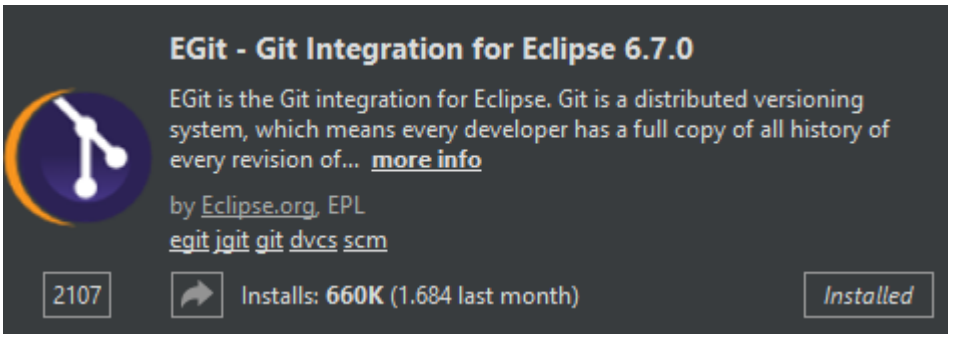
Instruccions d'ús:

1. Creeu un nou repositori pel mòdul.

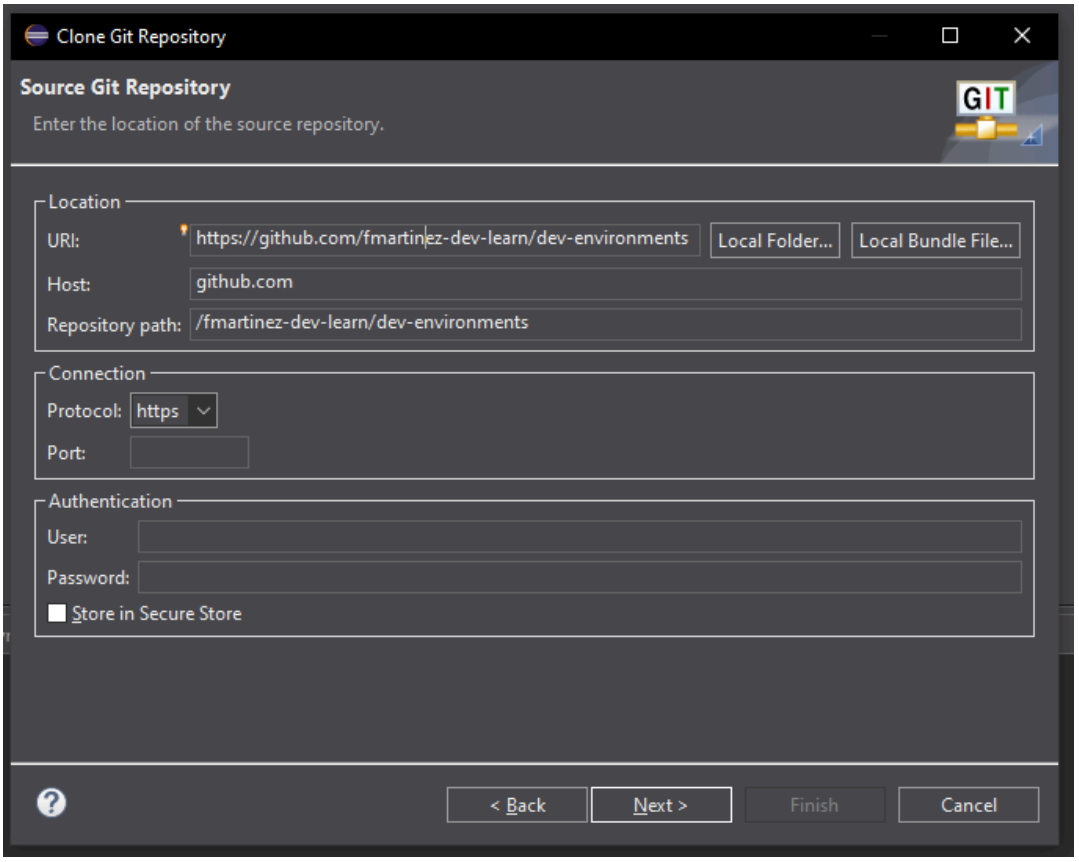




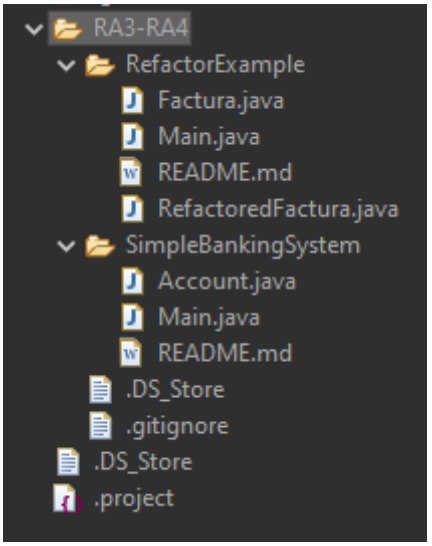
2. Configurar Eclipse per treballar amb Git:
   Abans de començar, assegura't de tenir instal·lada la funcionalitat de Git en Eclipse. Si no tens el plugin de Git, pots instal·lar-ho seguint aquests passos: - Ves a Help > Eclipse Marketplace.
   - Busca EGit (que és el plugin de Git per Eclipse) i instal·la'l si no el tens.



3. Clonar el repositori original des d'Eclipse:
https://github.com/fmartinez-dev-learn/dev-environments
 RA3-RA4/SimpleBankingSystem



4. Al mateix repositori podeu trobar un exemple de refactorització al directori
 RA3-RA4/Refactor exemple



5. Show in local terminal per treballar des de l'eclipse amb git.

Instruccions d'entrega:

- Repositori de GitHub.
- README actualitzat.
- Documentació PDF.

Objectiu: entendre com es gestionen les operacions de dipositar i retirar, com es modifiquen les variables i com es capturen les excepcions en cas d'errors.

Documenteu els següents passos:

1. Descripció inicial del codi:
   - Què fa el codi? (Explicar breument).
   - Quins són els mètodes més importants i què fan?
   - Quin és el valor inicial del saldo (`balance`) abans de realitzar qualsevol operació?

Què fa el codi?
El programa gestiona un sistema bancari bàsic que permet als usuaris realitzar operacions com **dipositar diners**, **retirar diners** i **consultar el saldo** d'un compte. També inclou mecanismes per evitar errors, com impedir que es retirin més diners dels disponibles.

Quins són els mètodes més importants i què fan?

1. `depositAmount(double amount)`

- Afegeix una quantitat de diners al saldo del compte.
- No permet dipòsits de quantitats negatives.

2. `withdrawAmount(double amount)`
   - Resta una quantitat de diners del saldo si hi ha prou fons disponibles.
   - Si l'import és major que el saldo, mostra un missatge d'error o llança una excepció.
3. `getBalance()`
   - Retorna el saldo actual del compte.
4. `main(String[] args)`
   - Gestiona la interacció amb l'usuari, executant operacions de dipòsit i retirada de diners.

Quin és el valor inicial del saldo (`balance`) abans de realitzar qualsevol operació?
El valor inicial del saldo depèn de la implementació del constructor de la classe `Account`. Si no es defineix un saldo inicial, podria ser **0 per defecte**. Si hi ha un constructor que estableix un saldo inicial, aquest valor podria variar.

2. Posar punts de control (Breakpoints): Per depurar el codi, utilitza els punts de control (breakpoints). Això permet aturar l'execució del codi en determinats punts i examinar l'estat de les variables. Per afegir un punt de control, fes clic a la barra de l'esquerra de la línia on vols aturar el codi.

   • On has col·locat els punts de control (breakpoints) i per què?
   • Inclou una captura de pantalla de Eclipse amb els breakpoints activats abans de començar la depuració.

   `depositAmount()` → Comprovar que el dipòsit es processa correctament.
   `withdrawAmount()` → Verificar que el saldo es modifica bé i captura errors.
   `main()` abans d'una operació → Veure l'estat inicial de les variables.
   Excepcions (saldo insuficient) → Comprovar que es llancen correctament.

3. Examina les variables i el flux d'execució:
   • A mesura que el codi s'atura a cada punt de control, observa el valor de les variables name, account i balance. Inclou una captura de pantalles dels valors de les variables a mesura que avancen les operacions.

4. Explora les excepcions:
   • Feu els canvis necessaris al Main per fer saltar les excepcions. Inclou la captura de pantalla d'un missatge d'error generat per una excepció i com es visualitza al terminal o a la consola de Eclipse.

Part 2: Control de versions amb GitHub

1. Crear un repositori GitHub: Heu de crear un repositori a GitHub per gestionar el codi.
2. Crear branques:
    a. Branca 1: Proves unitàries del dipòsit (tests-deposit).
        git checkout -b tests-deposit
    b. Branca 2: Proves unitàries del retir (tests-withdraw).
    c. Branca 3: Refactorització (refactor-main).
3. Fer commits separats per cada test.

Part 3: Proves unitàries

Objectiu: Escriure proves unitàries per als mètodes depositAmount() i withdrawAmount() per garantir que el codi funciona correctament i gestionar les excepcions adequadament.

1. Crear una classe de proves que es digui "AccountTest" i contingui dos mètodes: un per provar els dipòsits i un altre per provar les retirades. 2. Les proves han de verificar casos d'èxit, així com casos amb errors (quantitat negativa, saldo insuficient). Utilitzeu el mètode assertEquals.

Part 4: Refactorització del codi

La refactorització ha de seguir el patró Extract Method, que consisteix en separar les responsabilitats dins de la classe Main en mètodes independents, seguint les pràctiques de refactorització descrites a la pàgina web de Refactoring Guru https://refactoring.guru/extract-method.

En la classe Main, actualment tenim un mètode main que té moltes responsabilitats: gestionar el compte, realitzar un ingrés, retirar diners i manejar les excepcions. Segons les pràctiques de refactorització, podem dividir aquesta responsabilitat en diversos mètodes, per tal de simplificar el flux de l'aplicació.

Explica quines decisions i pasos segueixes per refactoritzar el codi i el perquè.

Annex 1: Instruccions per modificar l'arxiu README:

Descarregar Markdown editor. Seguiu les instruccions del següent enllaç:
https://davidrengifo.wordpress.com/2015/06/05/markdown-editor-plugin-for-eclipse-ide/

Annex 2: Comandes git:

git status Mostra l'estat del repositori, incloent canvis no compromesos, arxius nous…
git add <arxiu> Afegeix arxius a l'àrea de preparació (staging area) per al commit. git commit -m "<missatge>" Realitza un commit amb els canvis que estan a l'àrea de preparació.
git push Puja els canvis locals al repositori remot (per exemple, a GitHub, GitLab, etc.).
git pull Baixa i fusiona els canvis del repositori remot al repositori local.

git branch Mostra les branques locals del repositori.
git checkout <branca> Canvia a la branca especificada.
git checkout -b <branca> Crea i canvia a una nova branca en un sol pas.
git merge <branca> Fusiona la branca especificada amb la branca actual.
git branch -a Mostra totes les branques, tant locals com remotes.

```java
public class Account {
    private double balance;

    public Account() {
        this.balance = 0; // Saldo inicial
    }

    public double getBalance() {
        return balance;
    }

    public void depositAmount(double amount) {
        if (amount <= 0) {
            throw new IllegalArgumentException("L'import ha de ser positiu.");
        }
        balance += amount;
        System.out.println("S'ha dipositat: " + amount + "€ | Nou saldo: " + balance + "€");
    }

    public void withdrawAmount(double amount) {
        if (amount <= 0) {
            throw new IllegalArgumentException("L'import ha de ser positiu.");
        }
        if (amount > balance) {
            throw new IllegalArgumentException("Saldo insuficient per retirar " + amount + "€");
        }
        balance -= amount;
        System.out.println("S'ha retirat: " + amount + "€ | Nou saldo: " + balance + "€");
    }
}
```

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Account account = new Account();
        Scanner scanner = new Scanner(System.in);

        processDeposit(account, scanner);
        processWithdraw(account, scanner);
        showBalance(account);
    }

    private static void processDeposit(Account account, Scanner scanner) {
        System.out.println("Introdueix la quantitat a ingressar:");
        double amount = scanner.nextDouble();
        try {
            account.depositAmount(amount);
        } catch (IllegalArgumentException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }

    private static void processWithdraw(Account account, Scanner scanner) {
        System.out.println("Introdueix la quantitat a retirar:");
        double amount = scanner.nextDouble();
        try {
            account.withdrawAmount(amount);
        } catch (IllegalArgumentException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }

    private static void showBalance(Account account) {
        System.out.println("Saldo actual: " + account.getBalance() + "€");
    }
}
```

```java
import static org.junit.Assert.*;
import org.junit.Test;

public class AccountTest {

    @Test
    public void testDeposit() {
        Account account = new Account();
        account.depositAmount(100);
        assertEquals(100, account.getBalance(), 0.001);
    }

    @Test(expected = IllegalArgumentException.class)
    public void testDepositNegative() {
        Account account = new Account();
        account.depositAmount(-50);
    }

    @Test
    public void testWithdraw() {
        Account account = new Account();
        account.depositAmount(200);
        account.withdrawAmount(50);
        assertEquals(150, account.getBalance(), 0.001);
    }

    @Test(expected = IllegalArgumentException.class)
    public void testWithdrawExcess() {
        Account account = new Account();
        account.withdrawAmount(500);
    }
}
```