АКАДЕМИЈА
ТЕХНИЧКО-ВАСПИТАЧКИХ
СТРУКОВНИХ СТУДИЈА

PROGRAMSKI ALATI ZA RAZVOJ SOFTVERA

Vežba 8: Primer drugog kolokvijuma

U ovoj vežbi je dat primer drugog kolokvijuma, koji će ovog puta da se sastoji od dva zadatka. Prvi zadatak sadrži dizajn paterne, i biće potrebno nacrtati UML klasni dijagram, opisati patern koji koristite, u par rečenica objasniti zašto je on dobro rešenje za predstavljeni problem, i na kraju napisati nekoliko jUnit ili Pytestova, više kao simulacija kako se vaše rešenje može testirati. Drugi zadatak sadrži neki popularni problem iz softverskog inženjerstva, gde je neophodno skicirati izgled aplikacije, kako smatrate da je najbolje da se to odradi, zatim opisati sve klase (navesti atribute i metode) i izlistati grafičke komponente koje biste stavili (možete koristiti ili Javu ili Python za ovo). Na kraju i ovde treba odraditi simulaciju sa pisanjem par testova, čisto da se vidi kako bi to radilo u praksi, tj. kako bismo mogli da proverimo ispravnost implementacije. Pažljivo ispratite naredne delove, odradite ih i samostalno, i zaista nešto slično će biti na kolokvijumu.

Prvi zadatak: Primena dizajn paterna

Scenario: Upravljanje različitim vrstama korisnika na web platformi

U ovom zadatku, razmatraćemo sistem koji upravlja različitim vrstama korisnika na nekoj web platformi. Na osnovu tipa korisnika, oni imaju različite privilegije i pristup različitim funkcijama sistema. Na primer, postoje tri tipa korisnika:

- **AdminUser**: Ima potpuni pristup svim funkcijama sistema, uključujući upravljanje korisnicima i pristup svim panelima.
- **ModeratorUser**: Ima pristup moderacijskim funkcijama, ali ne može da menja korisničke podatke ili pristup drugim administrativnim funkcijama.
- **RegularUser**: Ima osnovni pristup, kao što su mogućnost komentarisanja i pregled sadržaja.

Naša aplikacija treba da bude sposobna da na osnovu tipa korisnika kreira odgovarajuće objekte i dodeli im odgovarajuće privilegije, koristeći **Factory Pattern**. Klijenti neće direktno kreirati objekte, već će koristiti fabriku koja će odlučiti koji tip korisnika da stvori.

Zašto koristiti Factory Pattern?

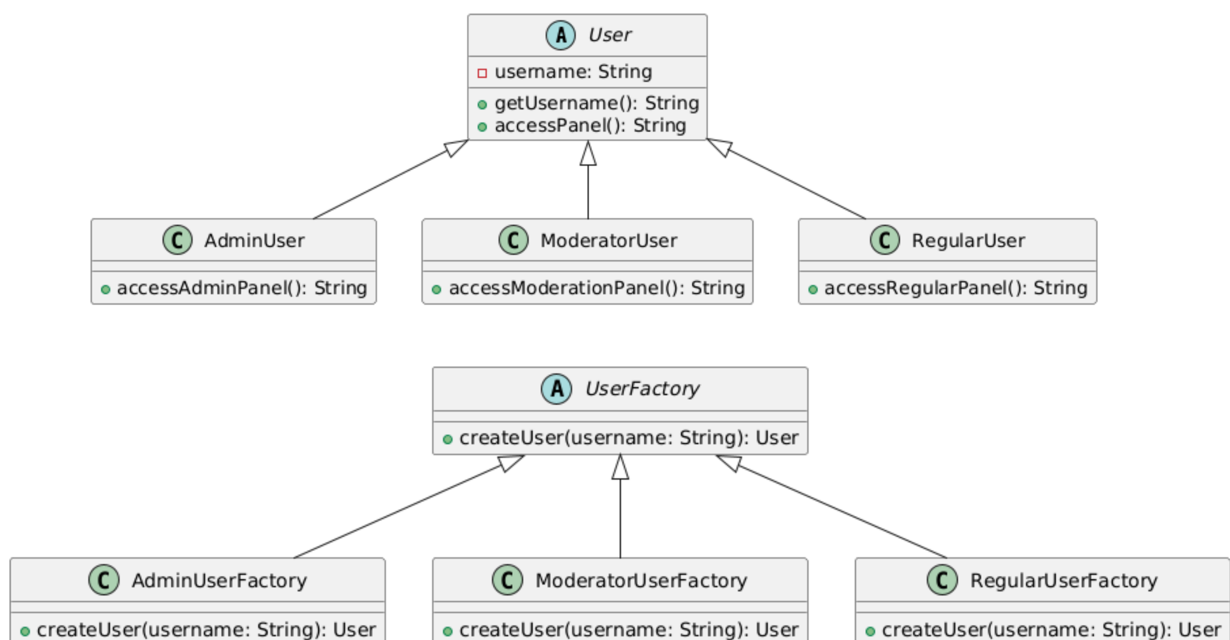
Factory Pattern je dizajniran da reši problem kreiranja objekata na fleksibilan način, bez potrebe da klijent direktno instancira konkretne klase. Umesto toga, koristi se fabrika koja odlučuje koji objekat da stvori, što omogućuje lakše proširivanje sistema.

Za naš scenario, **Factory Method** je koristan jer imamo više tipova korisnika (Admin, Moderator, Regular) sa različitim privilegijama. Korišćenjem fabrika za kreiranje tih korisnika, omogućujemo lakše proširivanje (dodavanje novih tipova korisnika) bez menjanja postojećeg koda. Na primer, dodavanje novih tipova korisnika ("GuestUser", "SuperAdmin", itd.) može se obaviti jednostavno dodavanjem novih fabrika, bez potrebe za menjanjem postojećeg koda.

Glavne prednosti:

- *Decoupling* između klijenta i konkretnih klasa.
- Lako proširenje sistema dodavanjem novih tipova korisnika.
- Centralizovana logika za kreiranje objekata.

UML klasni dijagram može izgledati ovako:



Objašnjenje UML dijagrama:

1. **User**: Apstraktna klasa koja predstavlja korisnika, sa zajedničkim atributima kao što je username i metodom accessPanel() koja je apstraktna i mora biti implementirana u konkretnim korisničkim klasama.
2. **AdminUser**, **ModeratorUser**, **RegularUser**: Konkretne klase koje implementiraju specifične metode za pristup različitim panelima. Na primer, admin može imati pristup svim panelima, dok regularni korisnik ima ograničen pristup. Ove klase omogućuju različite strategije ponašanja korisnika.
3. **UserFactory**: Apstraktna fabrika koja sadrži apstraktni metod createUser(), koji će biti implementiran u konkretnim fabrikama.
4. **AdminUserFactory**, **ModeratorUserFactory**, **RegularUserFactory**: Konkretne fabrike koje implementiraju metod createUser() i kreiraju odgovarajući tip korisnika (admin, moderator, regular).

Koje klase bismo imali u Javi? *(Ovaj deo nije potreban na kolokvijumu, vidi se iz dijagrama)*

1. **User (apstraktna klasa)**:
   - **Atributi**: username (String) – ime korisnika.
   - **Metode**:
     - getUsername() – vraća korisničko ime.
     - accessPanel() – apstraktna metoda koja mora biti implementirana u konkretnim klasama (vraća specifičan tekst u zavisnosti od tipa korisnika).
2. **AdminUser, ModeratorUser, RegularUser (konkretne klase)**:
   - Svaka od ovih klasa nasledjuje **User** i implementira accessPanel() prema specifičnim privilegijama svakog tipa korisnika. Takođe, imaju svoje specifične metode za pristup panelima:
     - accessAdminPanel() za **AdminUser**.
     - accessModerationPanel() za **ModeratorUser**.
     - accessRegularPanel() za **RegularUser**.
3. **UserFactory (apstraktna fabrika)**:
   - Apstraktna klasa koja definiše metodu createUser(username), koja mora biti implementirana u konkretnim fabrikama. Ona je odgovorna za kreiranje korisnika.
4. **AdminUserFactory, ModeratorUserFactory, RegularUserFactory (konkretne fabrike)**:
   - Svaka od ovih klasa implementira **UserFactory** i implementira metodu createUser(). Na osnovu tipa korisnika koji se traži, fabrika stvara odgovarajući objekat (npr. **AdminUser**, **ModeratorUser**, ili **RegularUser**).

Primer testova:

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

public class UserFactoryTest {
    // Testira da li fabrika za admina ispravno kreira AdminUser i kako radi
    @Test
    public void testKreirajAdminKorisnika() {
        UserFactory fabrika = new AdminUserFactory();
        User admin = fabrika.createUser("admin123");
        assertTrue(admin instanceof AdminUser);
        assertEquals("Pristupanje admin kontrolnom panelu...", ((AdminUser)
                                                admin).accessAdminPanel());
    }
```

```java
    // Testira da li fabrika moderatora ispravno kreira ModeratorUser i kako radi
    @Test
    public void testKreirajModeratorKorisnika() {
        UserFactory fabrika = new ModeratorUserFactory();
        User moderator = fabrika.createUser("moderator123");
        assertTrue(moderator instanceof ModeratorUser);
        assertEquals("Pristupanje moderatorskom kontrolnom panelu...",
                     ((ModeratorUser) moderator).accessModerationPanel());
    }

    // Testira da li fabrika korisnika ispravno kreira RegularUser i kako radi
    @Test
    public void testKreirajRegularnogKorisnika() {
        UserFactory fabrika = new RegularUserFactory();
        User regular = fabrika.createUser("regular123");
        assertTrue(regular instanceof RegularUser);
        assertEquals("Pristupanje panelu regularnog korisnika...", ((RegularUser)
                                                  regular).accessRegularPanel());
    }

    // Testira da li fabrika može da kreira korisnika sa nevalidnim imenom
    @Test
    public void testKreirajKorisnikaSaPraznimImenom() {
        UserFactory fabrika = new RegularUserFactory();
        User korisnik = fabrika.createUser("");
        assertNull(korisnik);
    }

    // Testira da li fabrika ispravno kreira korisnika kada je ime null.
    @Test
    public void testKreirajKorisnikaSaNullImenom() {
        UserFactory fabrika = new AdminUserFactory();
        User korisnik = fabrika.createUser(null);
        assertNull(korisnik);
    }
}
```

Drugi zadatak: Grafički korisnički interfejs

Cilj ovog zadatka je da razvijemo aplikaciju sa grafičkim korisničkim interfejsom (GUI) koja omogućuje korisnicima da efikasno upravljaju zadacima (taskovima). Korisnici mogu dodavati nove zadatke, ažurirati postojeće, brisati zadatke i filtrirati ih prema statusu.

---

1. Skica izgleda aplikacije *(u svom radu je neophodno priložiti i crtež rukom ili sa draw.io)*

Glavni ekran:

- **Prikaz liste zadataka**: Tabela koja prikazuje zadatke sa kolonama za naziv, opis, datum i status.
- **Opcije za filtriranje**: Padajući meni (drop-down) za filtriranje zadataka prema statusu ("u toku", "završeno", "na čekanju").
- **Opcije za pretragu**: Tekstualno polje za pretragu zadataka po nazivu.
- **Dugmad za akcije**:
    - "Dodaj zadatak"
    - "Izmeni zadatak"
    - "Obriši zadatak"

Dodavanje zadatka:

- **Forma za unos zadatka**:
    - Polja: Naziv, opis, datum, status (sa izborom iz padajućeg menija).
    - Dugme: "Sačuvaj".

Izmena zadatka:

- **Forma za izmenu zadatka**:
    - Automatsko popunjavanje trenutnih vrednosti zadatka.
    - Polja: Naziv, opis, datum, status.
    - Dugme: "Ažuriraj".

---

2. Klase i metode

Klasa Task: Predstavlja pojedinačni zadatak.

Atributi:

- name (String): Naziv zadatka.
- description (String): Kratak opis zadatka.
- status (String): Status zadatka ("u toku", "završeno", "na čekanju").

Metode:

- getName(): Vraća naziv zadatka.
- getDescription(): Vraća opis zadatka.
- getStatus(): Vraća status zadatka.
- setStatus(String newStatus): Postavlja novi status zadatka.

Klasa TaskManager: Omogućuje rad sa kolekcijom zadataka.

Atributi:

- taskList (List<Task>): Lista svih zadataka.

Metode:

- addTask(Task task): Dodaje novi zadatak u listu.
- removeTask(String taskName): Briše zadatak prema nazivu.
- updateTask(String taskName, Task updatedTask): Ažurira postojeći zadatak prema nazivu.
- getTasks(String filterStatus): Vraća listu zadataka, opcionalno filtriranu prema statusu.

Klasa TaskGUI: Implementira grafički korisnički interfejs.

Atributi:

- addButton (JButton): Dugme za dodavanje zadatka.
- editButton (JButton): Dugme za izmenu zadatka.
- deleteButton (JButton): Dugme za brisanje zadatka.
- taskTable (JTable): Tabela za prikaz zadataka.
- filterMenu (JComboBox): Padajući meni za filtriranje zadataka.

Metode:

- showTasks(List<Task> tasks): Prikazuje listu zadataka u tabeli.
- addTask(): Otvara formu za dodavanje zadatka.
- updateTask(): Omogućava korisniku da izmeni zadatak.
- removeTask(): Briše označeni zadatak.

---

3. Dodavanje zadatka:

1. Korisnik klikne na dugme "Dodaj zadatak".
2. Otvara se forma gde korisnik unosi naziv, opis, datum i status.
3. Klikom na "Sačuvaj", zadatak se dodaje u listu zadataka u klasi TaskManager, i ažurira se prikaz u GUI-u.

4. Filtriranje zadataka:

1. Korisnik bira status iz padajućeg menija.
2. TaskManager.getTasks() vraća filtriranu listu zadataka.
3. GUI ažurira tabelu da prikazuje samo zadatke sa odabranim statusom.

5. Brisanje zadatka:

1. Korisnik selektuje zadatak u tabeli i klikne na "Obriši".
2. GUI poziva TaskManager.removeTask() za odgovarajući naziv zadatka.
3. Tabela se osvežava nakon brisanja.

Primer testova:

```
class TestTaskManager(unittest.TestCase):

    def setUp(self):
        """Postavljanje osnovnih podataka za svaki test."""
        self.manager = TaskManager()
        self.task1 = Task(name="Zad 1", description="Opis 1", status="u toku")
        self.task2 = Task(name="Zad 2", description="Opis 2", status="završeno")
        self.task3 = Task(name="Zad 3", description="Opis 3", status="na čekanju")
        self.manager.addTask(self.task1)
        self.manager.addTask(self.task2)
        self.manager.addTask(self.task3)

    def test_add_task(self):
        """Provera da li dodavanje zadatka funkcioniše ispravno."""
        new_task = Task(name="Zad 4", description="Opis 4", status="u toku")
        self.manager.addTask(new_task)
        self.assertEqual(len(self.manager.taskList), 4)
        self.assertIn(new_task, self.manager.taskList)

    def test_remove_task(self):
        """Provera da li brisanje zadatka funkcioniše ispravno."""
        self.manager.removeTask("Zad 2")
        self.assertEqual(len(self.manager.taskList), 2)
        task_names = [task.getName() for task in self.manager.taskList]
        self.assertNotIn("Zadatak 2", task_names)
```

```python
def test_update_task(self):
    """Provera da li ažuriranje zadatka funkcioniše ispravno."""
    updated_task = Task(name="Zad 2", description="Opis 2", status="u toku")
    self.manager.updateTask("Zad 2", updated_task)
    self.assertEqual(self.manager.getTasks()[1].getDescription(), "Opis 2")
    self.assertEqual(self.manager.getTasks()[1].getStatus(), "u toku")

def test_get_tasks_no_filter(self):
    """Provera da li dobijanje zadataka funkcioniše ispravno bez filtera."""
    tasks = self.manager.getTasks()
    self.assertEqual(len(tasks), 3)

def test_get_tasks_with_filter(self):
    """Provera da li dobijanje zadataka po statusu funkcioniše ispravno."""
    tasks = self.manager.getTasks("u toku")
    self.assertEqual(len(tasks), 1)
    self.assertEqual(tasks[0].getName(), "Zadatak 1")

def test_empty_task_list(self):
    """Provera ponašanja kada lista zadataka postane prazna."""
    self.manager.removeTask("Zad 1")
    self.manager.removeTask("Zad 2")
    self.manager.removeTask("Zad 3")
    self.assertEqual(len(self.manager.taskList), 0)

def test_add_duplicate_task(self):
    """Provera da li aplikacija podržava dodavanje duplikata."""
    dupl_task = Task(name="Zad 1", description="Dupl", status="na čekanju")
    self.manager.addTask(duplicate_task)
    self.assertEqual(len(self.manager.taskList), 4)
    task_names = [task.getName() for task in self.manager.taskList]
    self.assertEqual(task_names.count("Zad 1"), 2)

def test_update_nonexistent_task(self):
    """Provera ponašanja kada pokušamo da ažuriramo nepostojeći zadatak."""
    updated_task = Task(name="Nepost", description="Opis", status="u toku")
    self.manager.updateTask("Nepost", updated_task)
    self.assertEqual(len(self.manager.taskList), 3)  # Broj zad ostaje isti
```