АКАДЕМИЈА ТЕХНИЧКО-ВАСПИТАЧКИХ СТРУКОВНИХ СТУДИЈА
ОДСЕК НИШ

PROGRAMSKI ALATI ZA RAZVOJ SOFTVERA

Vežba 6

Testiranje problema sa dizajn paternima

Ova vežba pruža još jedan detaljan pregled najvažnijih dizajn paterna, objašnjava u kojim situacijama ih treba koristiti, zatim kako oni rešavaju uobičajene probleme u razvoju softvera, i kako ih testirati koristeći **JUnit** testove. Fokus je na praktičnoj primeni datih paterna uz primere iz svakodnevnog života. Na kraju vežbe se nalazi i jedan zadatak za samostalni rad, koji se direktno nadovezuje na opisane pojmove. Neophodno je uraditi kompletnu implementaciju u Javi zajedno sa test klasom i predati ceo projekat.

Singleton

Kada koristiti?

- Kada je potrebno imati **jednu globalnu instancu** klase koja kontroliše pristup nekom resursu, poput konekcija sa bazom podataka, podešavanja aplikacije, ili logovanja.
- Kada je kreiranje više instanci nepotrebno ili može uzrokovati greške (npr. više konekcija ka istoj bazi može preopteretiti sistem).

Koji problem rešava?

- Kontrolisanje kreiranja objekata da bismo osigurali da postoji **samo jedna instanca**.
- Centralizovano upravljanje resursima i smanjenje memorijske potrošnje.

Šta se testira u ovom primeru:

1. Da li Singleton klasa kreira istu instancu pri svakom pozivu.
2. Da li promene stanja Singleton objekta utiču na istu instancu.
3. Da li inicijalno stanje Singleton objekta ima odgovarajuće podrazumevane vrednosti.

```
public class SingletonPatternTest {
    @Test
    void testSingletonSameInstance() {
        ConfigurationManager config1 = ConfigurationManager.getInstance();
        ConfigurationManager config2 = ConfigurationManager.getInstance();
        assertSame(config1, config2, "Obe instance treba da budu iste");
    }

    @Test
    void testSingletonStateChange() {
        ConfigurationManager config = ConfigurationManager.getInstance();
        config.setConfiguration("New Config");
        assertEquals("New Config", config.getConfiguration(),
                     "Konfiguracija treba da bude ažurirana");
    }

    @Test
    void testDefaultConfiguration() {
        ConfigurationManager config = ConfigurationManager.getInstance();
        assertNotNull(config.getConfiguration(),
                     "Konfiguracija ne bi trebalo da bude null");
        assertEquals("New Config", config.getConfiguration(),
                    "Podrazumevana konfiguracija nije ispravna");
    }
}
```

Factory

Kada koristiti?

- Kada klijentski kod ne treba direktno da zavisi od konkretnih klasa (npr. kada kreirate različite tipove dokumenata, proizvoda ili korisnika).
- Kada želite fleksibilnost u dodavanju novih tipova objekata.

Koji problem rešava?

- Uklanja zavisnost klijenta od konkretnih klasa.
- Jednostavno proširenje sistema za nove tipove objekata.

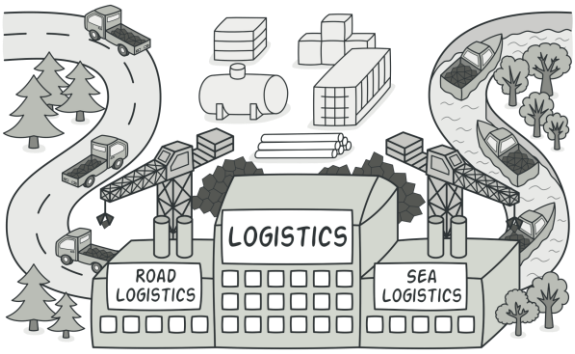
Šta se testira u ovom primeru:

1. Ispravnost kreiranja različitih tipova objekata (Truck i Ship).
2. Da li kreirani objekti vraćaju odgovarajuće opise dostave.
3. Da li fabrika ispravno obrađuje nepoznate ulaze (bacanje izuzetaka).

```
public class FactoryPatternTest {
    @Test
    void testFactoryCreatesTruck() {
        Transport transport = TransportFactory.createTransport("truck");
        assertEquals("Dostava kamionom.", transport.deliver(),
                    "Transport kamionom treba biti ispravno kreiran");
    }

    @Test
    void testFactoryCreatesShip() {
        Transport transport = TransportFactory.createTransport("ship");
        assertEquals("Dostava brodom.", transport.deliver(),
                    "Transport brodom treba biti ispravno kreiran");
    }

    @Test
    void testUnknownTransportType() {
        Exception exception = assertThrows(IllegalArgumentException.class, () -> {
            TransportFactory.createTransport("plane");
        });
        assertEquals("Nepoznat tip transporta: plane", exception.getMessage(),
                     "Poruka treba da odgovara grešci");
    }
}
```



Abstract Factory

Kada koristiti?

- Kada klijentski kod ne treba direktno da zavisi od konkretnih klasa (npr. kada kreirate različite tipove dokumenata, proizvoda ili korisnika).
- Kada želite fleksibilnost u dodavanju novih tipova objekata, bez izmene postojećeg koda.
- Kada aplikacija treba da podržava porodice objekata koji zajedno rade, ali im je potrebna mogućnost zamene ili proširenja.

Koji problem rešava?

- Uklanja zavisnost klijenta od konkretnih klasa.
- Jednostavno proširenje sistema za nove tipove objekata.

Šta se testira u ovom primeru:

1. Da li ModernFurnitureFactory kreira odgovarajuće proizvode.
2. Da li VictorianFurnitureFactory kreira odgovarajuće proizvode.
3. Da li klijent može neprimetno koristiti različite stilove nameštaja.

```
public class AbstractFactoryPatternTest {

    @Test
    void testModernFurnitureFactory() {
        FurnitureFactory modernFactory = new ModernFurnitureFactory();
        Chair modernChair = modernFactory.createChair();
        Table modernTable = modernFactory.createTable();

        assertEquals("Sedite na modernu stolicu.", modernChair.sitOn(),
                     "Moderna stolica nije pravilno kreirana.");
        assertEquals("Koristite moderan sto.", modernTable.use(),
                      "Moderan sto nije pravilno kreiran.");
    }

    @Test
    void testVictorianFurnitureFactory() {
        FurnitureFactory victorianFactory = new VictorianFurnitureFactory();
```

```
        Chair victorianChair = victorianFactory.createChair();
        Table victorianTable = victorianFactory.createTable();

        assertEquals("Sedite na viktorijansku stolicu.", victorianChair.sitOn(),
                     "Viktorijanska stolica nije pravilno kreirana.");
        assertEquals("Koristite viktorijanski sto.", victorianTable.use(),
                     "Viktorijanski sto nije pravilno kreiran.");
    }

    @Test
    void testClientWithDifferentFactories() {
        FurnitureFactory modernFactory = new ModernFurnitureFactory();
        FurnitureStore modernStore = new FurnitureStore(modernFactory);

        FurnitureFactory victorianFactory = new VictorianFurnitureFactory();
        FurnitureStore victorianStore = new FurnitureStore(victorianFactory);

        // Modern
        modernStore.describeFurniture();

        // Victorian
        victorianStore.describeFurniture();
    }
}
```

Decorator

Kada koristiti?

- Kada želite **dinamički dodavati funkcionalnosti** objektima bez menjanja njihove strukture.
- Kada želite fleksibilan način za kreiranje različitih varijacija osnovnih funkcionalnosti.

Koji problem rešava?

- Izbegava eksploziju podklasa u hijerarhijama klasa.
- Omogućava fleksibilno dodavanje ili uklanjanje funkcionalnosti tokom izvršenja programa (u runtime-u).

Šta se testira u ovom primeru:

1. Da li osnovni objekat (kafa) ispravno radi bez dodataka.
2. Da li se funkcionalnosti dekoratora (mleko i šećer) pravilno dodaju osnovnom objektu.
3. Da li su opisi i ukupne cene objekata ispravni nakon primene dekoratora.

```
public class DecoratorPatternTest {
    @Test
    void testBasicCoffee() {
        Beverage coffee = new Coffee();
        assertEquals("Osnovna kafa", coffee.getDescription(),
                    "Opis treba da bude osnovna kafa");
        assertEquals(50.0, coffee.cost(), 0.01, "Cena treba da bude 50.0");
    }

    @Test
    void testCoffeeWithMilkAndSugar() {
        Beverage coffee = new Coffee();
        coffee = new MilkDecorator(coffee);
        coffee = new SugarDecorator(coffee);

        assertEquals("Osnovna kafa, mleko, šećer", coffee.getDescription(),
                     "Opis treba da uključi sve dodatke");
        assertEquals(65.0, coffee.cost(), 0.01,"Cena treba da uključuje dodatke");
    }
}
```

Observer

Kada koristiti?

- Kada želite da jedan objekat (subjekt) obavesti jedan ili više objekata (posmatrača) o promenama svog stanja.
- Kada postoji veza između objekata gde jedan objekat zavisi od drugog, ali želite izbeći direktnu zavisnost.
- Kada je potrebno obezbediti automatsku sinhronizaciju stanja između objekata.

Koji problem rešava?

- Sprečava direktnu zavisnost između subjekta i posmatrača.
- Omogućava dinamičko dodavanje ili uklanjanje posmatrača tokom izvršenja programa.
- Olakšava rad sa sistemima gde događaji zahtevaju reakcije više komponenti.

Šta se testira u ovom primeru:

4. Da li posmatrači dobijaju obaveštenja o promenama subjekta.
5. Da li možemo dinamički dodavati i uklanjati posmatrače.
6. Da li stanje subjekta utiče na sve povezane posmatrače.

```
public class ObserverPatternTest {
    @Test
    void testObserverNotification() {
        WeatherStation station = new WeatherStation();
        SmartphoneDisplay display = new SmartphoneDisplay(station);

        station.setTemperature(25);
        assertEquals(25, display.getTemperature(),
                    "Display treba da reflektuje trenutnu temperaturu.");
    }

    @Test
    void testDynamicObserverAddition() {
        WeatherStation station = new WeatherStation();
        SmartphoneDisplay display1 = new SmartphoneDisplay(station);
        TabletDisplay display2 = new TabletDisplay(station);

        station.setTemperature(30);
         assertEquals(30, display1.getTemperature(),
                     "Prvi display treba da reflektuje temperaturu.");
        assertEquals(30, display2.getTemperature(),
                    "Drugi display treba da reflektuje temperaturu.");
    }
```

```
    @Test
    void testObserverRemoval() {
        WeatherStation station = new WeatherStation();
        SmartphoneDisplay display = new SmartphoneDisplay(station);

        station.removeObserver(display);
        station.setTemperature(20);
        assertNotEquals(20, display.getTemperature(), "Display ne treba da
                                    reflektuje promene nakon uklanjanja.");
    }
}
```

Strategy

Kada koristiti?

- Kada imate nekoliko različitih algoritama za rešavanje problema i želite da ih dinamički birate u runtime-u.
- Kada želite da izbegnete preklapanje koda u podklasama koristeći različite strategije.
- Kada želite fleksibilan način za menjanje ponašanja objekta bez menjanja njegovog koda.

Koji problem rešava?

- Uklanja potrebe za velikim *if-else* ili *switch* blokovima.
- Omogućava zamenu algoritama tokom izvršenja programa.
- Pruža jednostavan način za održavanje i proširenje novih strategija.

Šta se testira u ovoj klasi:

1. Da li strategije mogu dinamički da se promene u runtime-u.
2. Da li svaka strategija pravilno implementira svoje ponašanje.
3. Da li klijent može da koristi različite strategije bez zavisnosti od konkretnih implementacija.

```
public class StrategyPatternTest {
    @Test
    void testDynamicStrategySwitching() {
        PaymentContext context = new PaymentContext(new CreditCardPayment());
        context.pay(100);
        assertTrue(context.getPaymentMethod() instanceof CreditCardPayment,
                   "Treba koristiti CreditCardPayment strategiju.");

        context.setPaymentStrategy(new PayPalPayment());
        context.pay(200);
        assertTrue(context.getPaymentMethod() instanceof PayPalPayment,
                   "Treba koristiti PayPalPayment strategiju.");
    }

    @Test
    void testCreditCardPayment() {
        PaymentStrategy strategy = new CreditCardPayment();
        assertEquals("Paid 100 using Credit Card.", strategy.pay(100),
                     "Strategija CreditCardPayment ne funkcioniše ispravno.");
    }

    @Test
    void testPayPalPayment() {
        PaymentStrategy strategy = new PayPalPayment();
        assertEquals("Paid 50 using PayPal.", strategy.pay(50),
                     "Strategija PayPalPayment ne funkcioniše ispravno.");
    }
}
```

Composite

Kada koristiti?

- Kada želite da grupišete objekte u hijerarhijsku strukturu i tretirate ih kao jedinstvenu celinu.
- Kada postoji potreba za radom sa pojedinačnim objektima i grupama objekata na isti način.
- Kada imate složene strukture podataka koje se sastoje od delova i celina (npr. stabla, liste, grafikoni).

Koji problem rešava?

- Pojednostavljuje rad sa složenim hijerarhijskim strukturama.
- Uklanja potrebu za duplim kodom koji obrađuje pojedinačne objekte i grupe.
- Omogućava fleksibilno proširenje hijerarhija bez menjanja klijentskog koda.

Šta se testira u ovoj primeru:

1. Da li pojedinačni i kompozitni objekti pravilno implementiraju ponašanje.
2. Da li se hijerarhijska struktura pravilno tretira kao jedinstvena celina.
3. Da li se dodavanje/uklanjanje komponenti ispravno obrađuje.

```
public class CompositePatternTest {

    @Test
    void testSingleEmployeeDetails() {
        Employee worker = new Worker("John");
        assertEquals("Worker: John", worker.getDetails(),
                    "Pojedinačni zaposleni treba da daje tačne detalje.");
    }

    @Test
    void testManagerDetailsWithSubordinates() {
        Manager manager = new Manager("Alice");
        Worker worker1 = new Worker("John");
        Worker worker2 = new Worker("Jane");
        manager.addEmployee(worker1);
        manager.addEmployee(worker2);
```

```
    String expectedDetails = "Manager: Alice\nWorker: John\nWorker: Jane\n";
    assertEquals(expectedDetails, manager.getDetails(), "Menadžer sa
                podređenim zaposlenima treba da daje tačne detalje.");
}

@Test
void testCompositeStructureAsSingleUnit() {
    Manager manager = new Manager("Alice");
    Worker worker1 = new Worker("John");
    Worker worker2 = new Worker("Jane");

    manager.addEmployee(worker1);
    manager.addEmployee(worker2);

    // Provera da se menadžer i zaposleni tretiraju kao jedinstvena celina
    assertTrue(manager.getDetails().contains("Worker: John"),
                "Kompozitni objekat treba da sadrži sve komponente.");
    assertTrue(manager.getDetails().contains("Worker: Jane"),
                "Kompozitni objekat treba da sadrži sve komponente.");
    assertTrue(manager.getDetails().contains("Manager: Alice"),
                "Kompozitni objekat treba da uključuje sve članove.");
}

@Test
void testRemovingEmployeeFromManager() {
    Manager manager = new Manager("Alice");
    Worker worker1 = new Worker("John");
    Worker worker2 = new Worker("Jane");

    manager.addEmployee(worker1);
    manager.addEmployee(worker2);
    manager.removeEmployee(worker1);
```

```
        // Verifikacija nakon uklanjanja zaposlenog
        String expectedDetailsAfterRemoval = "Manager: Alice\nWorker: Jane\n";
        assertEquals(expectedDetailsAfterRemoval, manager.getDetails(),
              "Menadžer treba da bude ažuriran nakon uklanjanja zaposlenog.");
    }

    @Test
    void testEmptyManager() {
        Manager manager = new Manager("Alice");

        // Verifikacija menadžera bez podređenih
        assertEquals("Manager: Alice\n", manager.getDetails(),
              "Menadžer bez zaposlenih treba da vrati samo svoje ime.");
    }
}
```

Builder

Kada koristiti?

- Kada je potrebno kreirati složene objekte korak po korak.
- Kada želite odvojiti proces konstrukcije objekta od njegovog finalnog predstavljanja.
- Kada želite omogućiti ponovno korišćenje procesa konstrukcije za različite reprezentacije objekata.

Koji problem rešava?

- Sprečava preveliku složenost konstruktora kada imate mnogo parametara.
- Omogućava kreiranje različitih varijacija objekata pomoću istog procesa konstrukcije.
- Povećava čitljivost i modularnost koda.

Šta se testira u ovom primeru:

1. Da li Builder omogućava korak-po-korak kreiranje objekata.
2. Da li Builder podržava različite konfiguracije.
3. Da li se generisani objekti pravilno konstruišu.

```
public class BuilderPatternTest {
    @Test
    void testStepByStepBuilding() {
        Car car = new CarBuilder()
                .setMake("Tesla")
                .setModel("Model S")
                .setColor("Red")
                .build();

        assertEquals("Tesla",car.getMake(),"Marka automobila treba da je Tesla.");
        assertEquals("Model S", car.getModel(),"Model automobila treba da je S.");
        assertEquals("Red", car.getColor(),"Boja automobila treba da je crvena.");
    }

    @Test
    void testDifferentConfigurations() {
        Car car1 = new CarBuilder().setMake("BMW").setModel("X5").build();
        Car car2 = new CarBuilder().setMake("Audi").setModel("Q7")
                                                    .setColor("Black").build();

        assertEquals("BMW", car1.getMake(), "Prvi auto treba da je BMW.");
        assertEquals("Audi", car2.getMake(), "Drugi auto treba da je Audi.");
        assertEquals("Black",car2.getColor(),"Drugi auto treba da je crne boje.");
    }
}
```

Prototype

Kada koristiti?

- Kada želite kreirati objekte na osnovu postojećih primera (prototipova).
- Kada je kreiranje novog objekta skupo ili kompleksno, pa je brže klonirati postojeći objekat.
- Kada želite da izbegnete kreiranje novih klasa za svaki mogući objekat.

Koji problem rešava?

- Smanjuje troškove kreiranja objekata.
- Omogućava fleksibilnost u kreiranju sličnih objekata bez potrebe za pisanjem dodatnog koda.
- Sprečava zavisnost od konstruktora i njihove kompleksnosti.

Šta se testira u ovom primeru:

1. Da li klonirani objekti zadržavaju isto stanje kao original.
2. Da li klonirani objekti imaju različite reference u memoriji.
3. Da li je lako kreirati nove varijacije objekata kloniranjem.

```
public class PrototypePatternTest {
    @Test
    void testCloningPreservesState() {
        Document doc1 = new Document("Design Patterns", "A guide to patterns.");
        Document doc2 = doc1.clone();

        assertEquals(doc1.getTitle(), doc2.getTitle(),
                    "Naslov kloniranog dokumenta treba da je isti.");
        assertEquals(doc1.getContent(), doc2.getContent(),
                    "Sadržaj kloniranog dokumenta treba da je isti.");
    }

    @Test
    void testCloningCreatesNewInstance() {
        Document doc1 = new Document("Design Patterns", "A guide to patterns.");
        Document doc2 = doc1.clone();

        assertNotSame(doc1,doc2,"Klonirani objekat treba da je druga instanca.");
    }

    @Test
    void testModifyingClone() {
        Document doc1 = new Document("Design Patterns", "A guide to patterns.");
        Document doc2 = doc1.clone();
```

```
        doc2.setTitle("Advanced Patterns");

        assertNotEquals(doc1.getTitle(), doc2.getTitle(), "Promena naslova klona
                                                ne treba da utiče na original.");
    }
}
```

Zadatak za samostalni rad

Koristimo primer **apstraktne fabrike** i radimo njegovu nadogradnju. Dakle, potrebni su vam interfejsi za stolice i stolove (stolica ima metodu SitOn, kao da sednemo na stolicu, a sto ima metodu Use, kao da koristimo taj sto).

Što se konkretnih implementacija proizvoda tiče, možete za početak odraditi ono što je i dato u primeru - ModernChair, ModernTable, VictorianChair i VictorianTable. Zatim treba kreirati apstraktne i konkretne fabrike koje će praviti određene stolove i stolice. Klasu za klijenta možete nazvati FurnitureStore, da to bude neka vaša prodavnica nameštaja.

Nakon što utvrdite da vam inicijalna verzija dobro radi (proverite sa par JUnit testova), ispratite sledeće korake:

1. Dodajte novi stil nameštaja – *Industrial*. Kreirajte odgovarajuće implementacije Chair, Table, i novu fabriku IndustrialFurnitureFactory.
2. Proširite testove kako biste obuhvatili novi stil, napišite najmanje 3 nova testa.
3. Napravite dodatni proizvod (npr. *Sofa* ili *Closet*) i ažurirajte sve fabrike da podrže kreiranje tog proizvoda. Dodajte obavezno i testove za taj vaš novi proizvod.

Korisni resursi

- Priručnik za JUnit: https://livebook.manning.com/book/junit-recipes/chapter-14/
- Testiranje Builder paterna: https://www.javacodegeeks.com/2012/12/using-builder-pattern-in-junit-tests.html
- Još jedan koristan priručnik, pogledajte i ostala njegova poglavlja, ima dosta toga o dizajn paternima: https://softwarepatternslexicon.com/patterns-java/14/1/
- Više primera za testiranje dizajn paterna, pogledajte ovaj repository: https://github.com/stephenah-baloyi/tp2_design_patterns_tests
- Zanimljiv projekat, simulacija rok koncerta, koristi nekoliko dizajn paterna i testira preko JUnita: https://github.com/MMaier96/RockConcert-Simulation