17. JEDINIČNO TESTIRANJE

Unit-testing ili jednično testiranje je jedan od poznatijih načina testiranja. To je postupak u kom testiramo najmanje jedinice softvera koje predstavljaju jednu celinu. U objektno orijentisanom pristupu to je testiranje na nivou klase, u pojedinim slučajevima čak i na nivou metoda (procedura i funkcija). Unit testove u većini slučajeva pišu programeri dok rade na softveru kako bi se uverili da pojedine funkcije rade ispravno. Jediničnim testiranjem se ne osigurava da će svi ti pojedinačni delovi raditi ispravno kad se integririšu, već samo govori da svaki od tih testiranih delova radi ispravno. Upravo zbog toga unit testiranje se još naziva i testiranje po komponentama.

Postoje razni unit test okviri (Engl.“Framework”), pa je tako za programski jezik Javu najpoznatiji okvir JUnit, dok je za Microsoft .NET to NUnit. Pomoću njih moguće je automatsko izvršavanje unit testova koji se lako pokreću pomoću IDE-a. Proverava se da li u osnovnoj jedinici programa postoji neka greška ili neželjeno ponšanje.

Jedinica koda može biti:

- Funkcija,
- Metoda,
- Struktura podataka,
- Klasa koja sadrži atribute i metode,
- Operacija.

Celokupno testiranje klase obuhvata:

- Testiranje svih operacija povezanih sa objektima klase.
- Postavljanje i ispitivanje svih atributa objekata.
- Dovodjenje objekta u sva moguća stanja.
- Stimulisati sve dogadjaje koji izazivaju promenu stanja objekta.
- Nasledjivanje čini testiranje klase težim, jer jedinica koda koja se testira nije u potpunosti lokalizovana.
- U tom slučaju moramo testirati nasledjenu operaciju u svim kontekstima u kojima se koristi.

Svrha i strategija jediničnog testiranja - Test primeri bi trebalo da pokažu da jedinični deo koda koji se testira ima predviđenu funkcionalnost i nema neželjeno ponašanje. Postoje dva tipa testiranja osnovnih jedinica koda: prvi koji treba da pokaže da jedinica radi u skladu sa planovima i zahtevima i drugi koji je zasnovan na prethodnom iskustvu u testiranju. Prema tome, strategije testiranja osnovne jedinice koda su:

- **Partition testing** - particionisano testiranje. Identifikuju se klase ulaza koje su sličnog karaktera i koje bi trebalo da se ponašaju na sličan način. Ulazni i izlazni podaci se mogu grupisati u različite klase gde se svi članovi jedne klase ponašaju na isti nacin. Te klase se nazivaju particije ekvivalencije ili domeni, gde se program ekvivalentno ponaša sa svim članovima iste klase. Test primeri bi trebalo da budu odabrani iz svake klase. Trebalo bi odabrati test primere za granične slučajeve te klase i midpoint klase (srednju vrednost). Test primeri treba da budu takvi da se testiraju svi specifični slučajevi ulaza za svaku klasu.
- **Guideline – based testing**. U ovoj vrsti testiranja koriste se uputstva za biranje test slučajeva koja su napravljena na osnovu prethodnih iskustava. Uputstva su

napravljena po ugledu na greške koje su se često javljale u programima i na kojim mestima bi trebalo obratiti pažnju.

Kada god je moguće, poželjno je **automatizovati testiranje osnovnih jedinica koda**, tako da se testovi pokreću i njihovi rezultati proveravaju automatski. Pri automatskom testiranju može se koristiti framework za automatizaciju testiranja (npr. JUnit), tj. izrade i pokretanja testova za program. Framework za testiranje osnovnih jedinica koda pruža generičke klase testova koje treba naslediti za izradu specifičnih testova. Postoji mogućnost pokretanja svih implementiranih testova i prikazivanja njihovih rezultata. Na ovaj način, ceo skup testova može biti pokrenut u roku od nekoliko sekundi, što omogućava izvršavanje svih testova pri svakoj promeni programa.

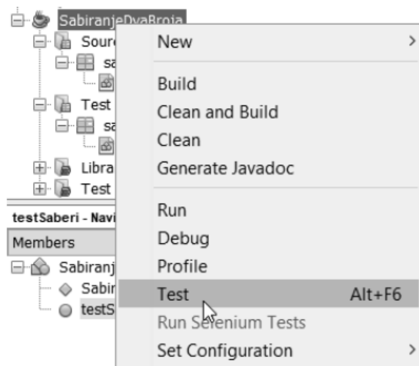
Komponente automatskih testova:

- Setup part – gde se inicijalizuje test, pre svega ulaz i očekivani izlaz.
- Call part – kreiranje objekta ili poziv metode kako bi bili testirani.
- Assertion part – poređenje rezultata poziva sa očekivanim rezultatima. Ukoliko se izlazi poklapaju, test prolazi, u suprotnom test ne prolazi.

Primer – test klasa za sabiranje dva broja:

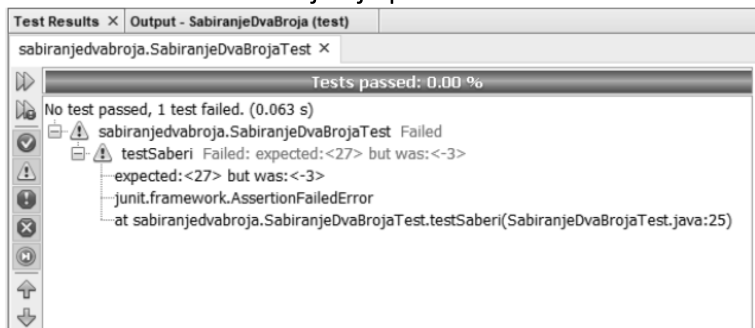
```
package sabiranjedvabroja;

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

/**
 *
 * @author mzivkovic
 */
public class SabiranjeDvaBrojaTest {

    public SabiranjeDvaBrojaTest() {
    }

    /**
     * Test of saberi method, of class SabiranjeDvaBroja.
     */
    @Test
    public void testSaberi() {
        System.out.println("saberi");
        int prvi = 0;
        int drugi = 0;
        int expResult = 0;
        int result = SabiranjeDvaBroja.saberi(prvi, drugi);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
        fail("The test case is a prototype.");
    }

}
```

Pokretanje testa:



Primer testa sa JUnit koji je prošao:

Primer testa sa JUnit koji nije prošao:



Prednosti unit testova:

- Mogu se pokretati 24/7.
- Manje ljudi potrebno u odnosu na ručno testiranje.
- Mogućnost ponovnog korišćenja, jako bitno kod regresionog testiranja u kojem se proverava da li je prethodni deo softvera nakon nadogradnje ili ispravke ostao korektan.
- Ušteda vremena.
- Rezultati testiranja lako dostupni svima.
- Pouzdani su jer se eliminiše mogućnost ljudske greške.

Nedostaci unit testova:

- Obično je razvoj automatskog testiranja dosta skuplji od manuelnog.
- Potrebno je vršiti održavanje testova, što zahteva dodatno angažovanje.