Testiranje softvera i upravljanje kvalitetom

Sadržaj

- ✧ Uvod
- ✧ Razvojno testiranje
- ✧ Testiranje proizvoda
- ✧ Korisničko testiranje

Proces razvoja sistema

Faza	Aktivnost	Izlaz
Započinjanje	Utvrđivanje poslovnih potreba	Biznis dokumenta
Analiza	Intervjuisanje stejkholdera, istraživanje sistemskog okruženja	Organizovana dokumentacija
Specifikacija	Analiza inženjerskih aspekata sistema, definisanje koncepata sistema	Logički model sistema
Implementacija	Programiranje, testiranje jedinica, integrisanje, dokumentovanje	Proverljiv sistem
Testiranje & Integracija	Integrisanje svih komponenti, verifikacija, validacija, instalacija, obuka	Resultati testiranja, funkcionalan sistem
Održavanje	Popravljanje bagova, modifikacije, adaptacija	Verzije sistema

Testiranje aplikacija

- Namena testiranja je da pokaže da aplikacija radi ono zašta je namenjena i da utvrdi aplikacione defekte pre nego što se stavi u upotrebu.
- Kada se testira softver, izvršava se program korišćenjem veštačkih podataka.
- Rezultati testova treba da pokažu greške, anomalije ili druge informacije o aplikaciji koje ukazuju na nepredviđeno ponašanje.
- Testiranje je deo generalnijih procesa validacije i verifikacije, koji uključuju i statičke tehnike validacije.

Ciljevi testiranja aplikacije

- Da se demonstrira programerima ili korisnicima da softver ispunjava definisane zahteve.
  - Za korisnika ovo znači da postoji najmanje jedan test za svaki zahtev iz dokumenta.
- Da se otkriju situacije u kojima je ponašanje softvera nekorektno ili nije u skladu sa definisanom specifikacijom.
  - Testiranje defekata se bavi utrvđivanjem uzroka neželjenog ponašanja sistema, kao što su otkazi aplikacije, neželjene interakcije sa drugim sistemima, pogrešna izračunavanja i korupcija podataka.

Validacija i testiranje defekata

- Prvi cilj vodi ka validacionom testiranju
  - Očekujemo da se aplikacija ponaša ispravno korišćenjem skupa testova koji oponašaju realnu upotrebu sistema.
- Drugi cilj vodi ka testiranju defekata
  - Testovi su napravljeni tako da otkriju defekte. Ne moraju reflektovati uobičajenu upotrebu sistema.

Ciljevi procesa testiranja

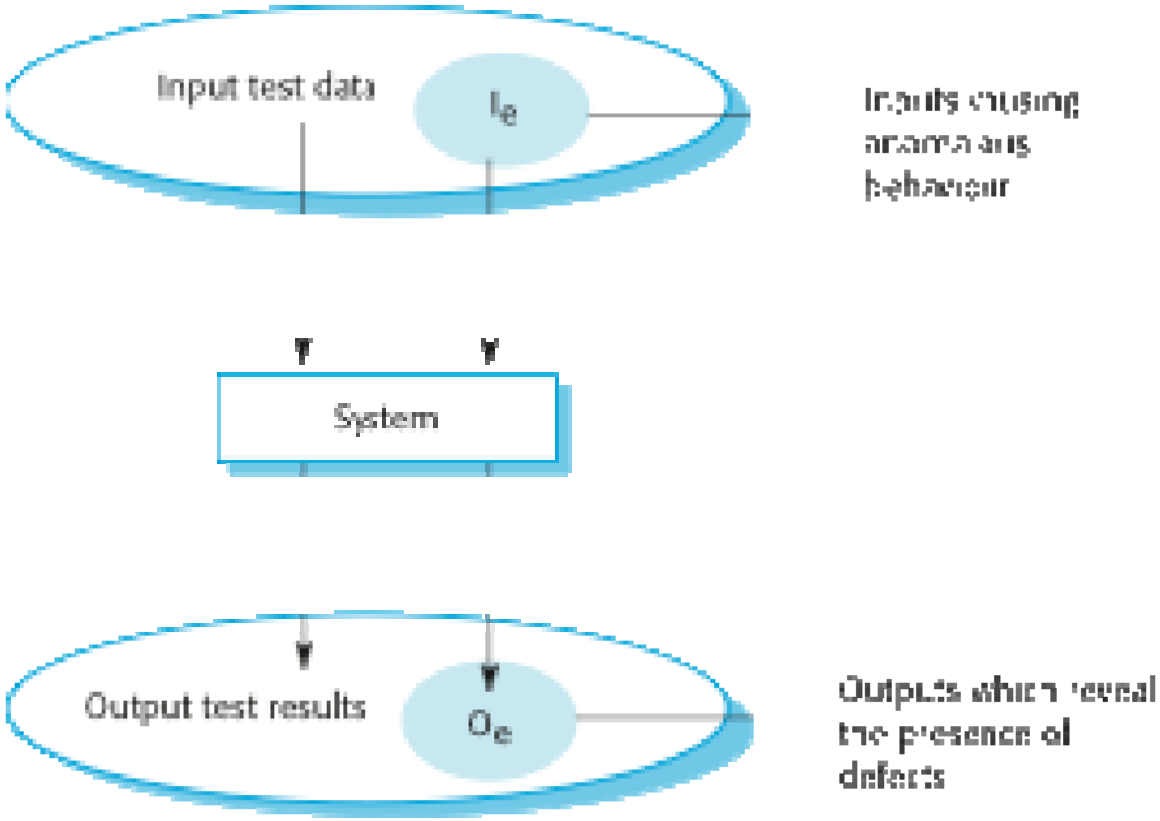
✧ Validaciono testiranje

- Da demonstrira programerima i korisnicima sistema da softver ispunjava sve definisane zahteve,
- Uspešan test pokazuje da sistem funkcioniše onako kako bi trebalo.

✧ Testiranje defekata

- Da otkrije greške ili otkaze u softveru čije je ponašanje neispravno ili nije u skladu sa definisanom specifikacijom,
- Uspešan test je test koji dovodi do nekorektnog ponašanja sistema i na taj način utvrđuje defekt u softveru.

Input-output model testiranja aplikacije



Osnovni koncepti

- **Validacija**: proces evaluacije sistema ili komponente za vreme ili na kraju procesa razvoja da bi se utvrdilo da li zadovoljava zahteve definisane od korisnika
  - Validacija: Da li kreiramo **pravi proizvod**?

- **Verifikacija**: proces evaluacije sistema ili komponente kako bi se utvrdilo da li proizvod korektno implementira određenu funkciju
  - Verifikacija: Da li kreiramo proizvod na **pravi način**?

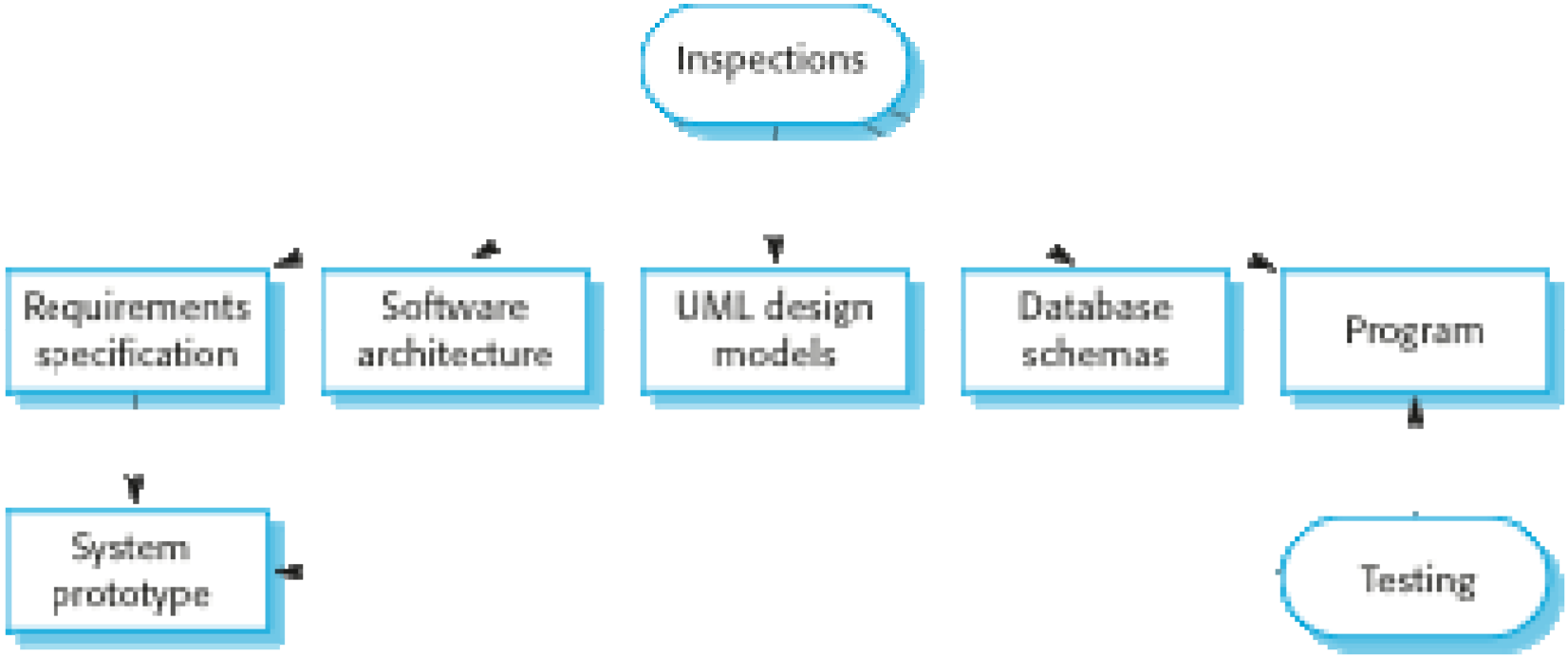
Validacija i verifikacija (V & V) - ciljevi

- Osnovni cilj V&V je uspostavljanje uverenja da sistem u potpunosti ispunjava svoju namenu.
- Zavisi od namene sistema, korisnikovih očekivanja i marketinškog okruženja.
  - Namena sistema – veoma je bitna činjenica koliko je softver kritičan za organizaciju,
  - Očekivanja korisnika – korisnici mogu imati mala očekivanja za određenu vrstu softvera,
  - Marketinško okruženje – plasiranje proizvoda na tržište ranije može biti važnije od pronalaženja defekata u softveru.

Inspekcije i testiranje

- Inspekcije softvera – Bave se analizom statičkog izgleda sistema kako bi se utvrdili problemi (statička verifikacija)
- Testiranje softvera – Bavi se proverom i posmatranjem ponašanja softvera (dinamička verifikacija)
  - Aplikacija se pokreće korišćenjem test podataka i prati se njeno ponašanje.

Inspekcije i testiranje



Inspekcija softvera

- Uključuje ljude koji proveravaju source code aplikacije sa ciljem utvrđivanja anomalija i defekata.
- Ne zahteva izvršenje aplikacije, pa se može koristiti i pre implementacije.
- Može se primeniti na bilo koji element sistema (zahteve, dizajn, konfiguracione podatke, podatke za testiranje...).
- Pokazalo se da su vrlo efikasna tehnika za utvrđivanje programskih grešaka.

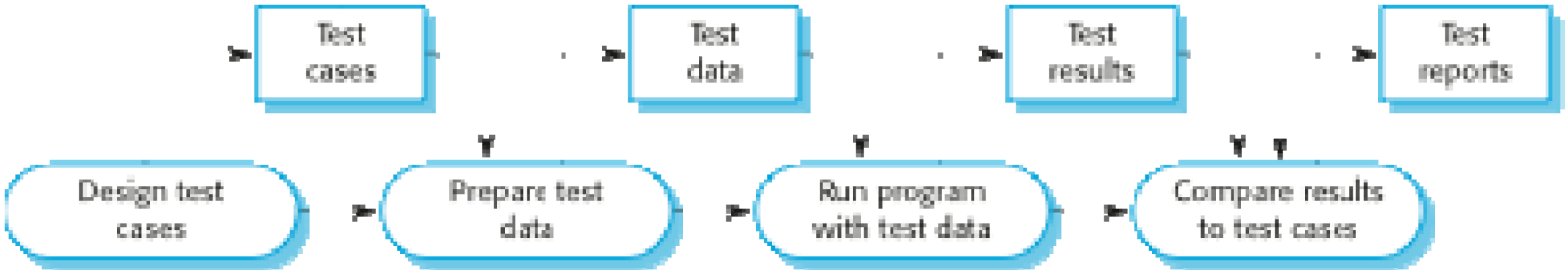
Prednosti inspekcija

- U toku testiranja, greške mogu maskirati (sakriti) druge greške. Zbog toga što je inspekcija statički proces, druge greške ne mogu uticati ili sakriti greške.
- Nekompletne verzije sistema se mogu proveravati bez dodatnih troškova. Ako je aplikacija nekompletna potrebno je razviti specijalizovan test koji će testirati dostupne delove.
- Osim pronalaženja programskih defekata, inspekcija može posmatrati i šire atribute kvaliteta aplikacije, kao što su poštovanje standarda, portabilnost i mogućnost održavanja.

Inspekcije i testiranje

- Inspekcija i testiranje su komplementarne a ne suprotne tehnike verifikacije.
- Treba ih koristiti u toku V&V procesa.
- Inspekcije mogu proveravati usklađenost sa specifikacijom ali ne i usklađenost sa korisnikovim stvarnim zahtevima.
- Inspekcije ne mogu proveriti nefunkcionalne karakteristike kao što su performanse, upotrebljivost, itd.

Model softverskog procesa testiranja



Faze testiranja

- Razvojno testiranje – sistem se testira u toku razvoja da bi se otkrili bagovi ili drugi defekti.
- Testiranje proizvoda (release testing) – poseban tim testira kompletnu verziju aplikacije pre nego što se ona isporuči korisnicima.
- Korisničko testiranje (user testing) – korisnici ili potencijalni korisnici sistema testiraju sistem u sopstvenom okruženju.

Razvojno testiranje

- Obuhvata sve aktivnosti testiranja koje izvršava tim za razvoj softvera.
  - Testiranje jedinica (Unit testing) – testiraju se individualne programske jedinice ili klase. Fokusira se na funkcionalnosti objekata ili metode.
  - Testiranje komponenti (Component testing) – nekoliko posebnih jedinica se integrišu kako bi kreirale složene komponente. Testiranje komponenti se fokusira na testiranje interfejsa komponenti.
  - Testiranje sistema – sistem kao celina se testira. Fokus je na testiranju interakcija između komponenti.

Testiranje jedinica

- Testiranje jedinica je proces testiranja individualnih delova softvera u izolaciji.
- To je proces testiranja defekata.
- Jedinice mogu biti:
  - Pojedinačne metode ili funkcije unutar objekata,
  - Objekti klasa sa nekoliko atributa i metoda,
  - Složene komponente sa definisanim interfejsima koji se koriste za pristupanje njihovim funkcionalnostima.

Testiranje klasa

- Testiranje klasa obuhvata
  - Testiranje svih operacija vezanih za klasu,
  - Postavljanje i menjanje vrednosti atributa klase,
- Nasleđivanje otežava dizajn testova klasa jer informacije koje se testiraju ne mogu biti lokalizovane.

Efikasnost testova jedinice

- Testovi treba da pokažu da komponente koje se testiraju rade ono što bi trebalo da rade.
- Ako postoje defekti u komponenti, oni bi trebalo da se pokažu u testovima slučajeva.
- Ovo vodi do 2 tipa testova jedinica:
  - Prvi koji reflektuje normalne operacije aplikacije i treba da pokaže da komponente rade onako kako se očekuje,
  - Drugi tip se zasniva na iskustvu programera da predvidi gde mogu postojati problemi. Ovaj tip treba da koristi abnormalne ulaze da bi proverio da li će biti pravilno obrađeni i da li će dovesti do otkaza komponente.

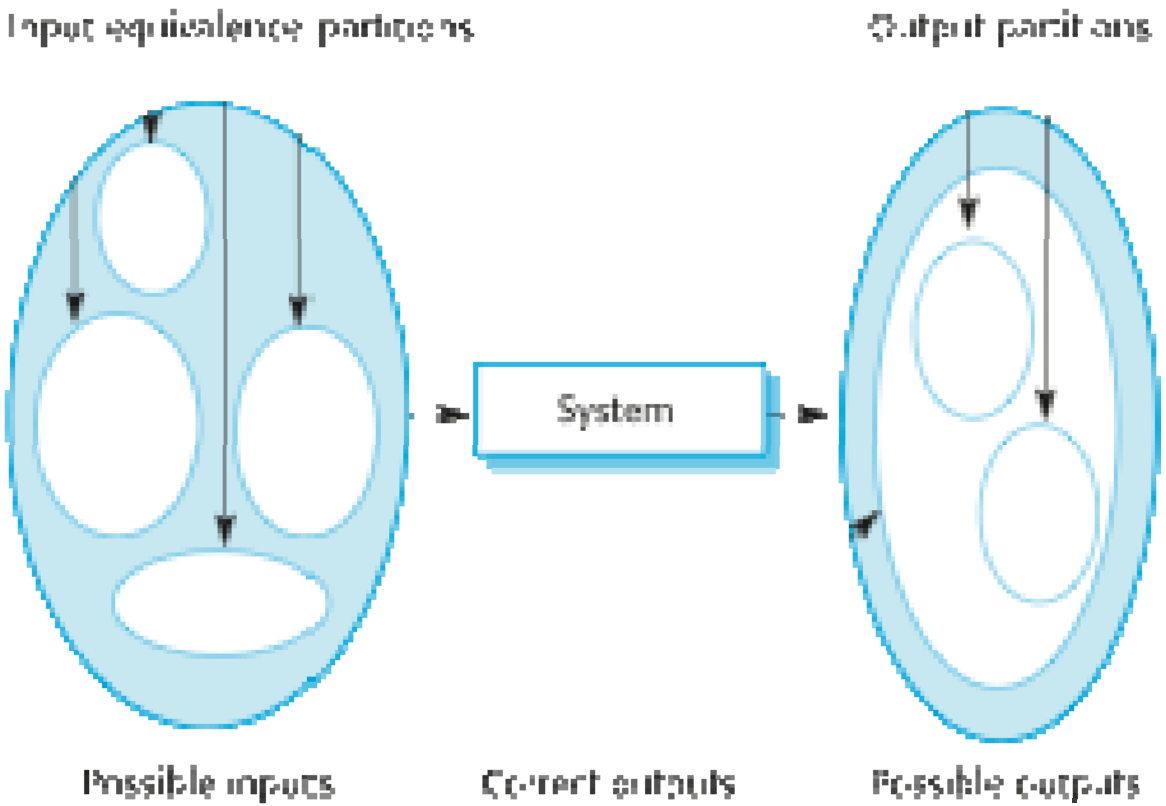
Strategije testiranja

- Testiranje po celinama, gde se identifikuju grupe ulaza koje imaju zajedničke karakteristike i koje treba obraditi na isti način.
  - Treba izabrati testove iz svake od ovih grupa
- Testiranje po smernicama, gde se koriste smernice za izbor testova.
  - Smernice predstavljaju predhodna iskustva o vrstama grešaka koje programeri prave prilikom razvoja komponenti.

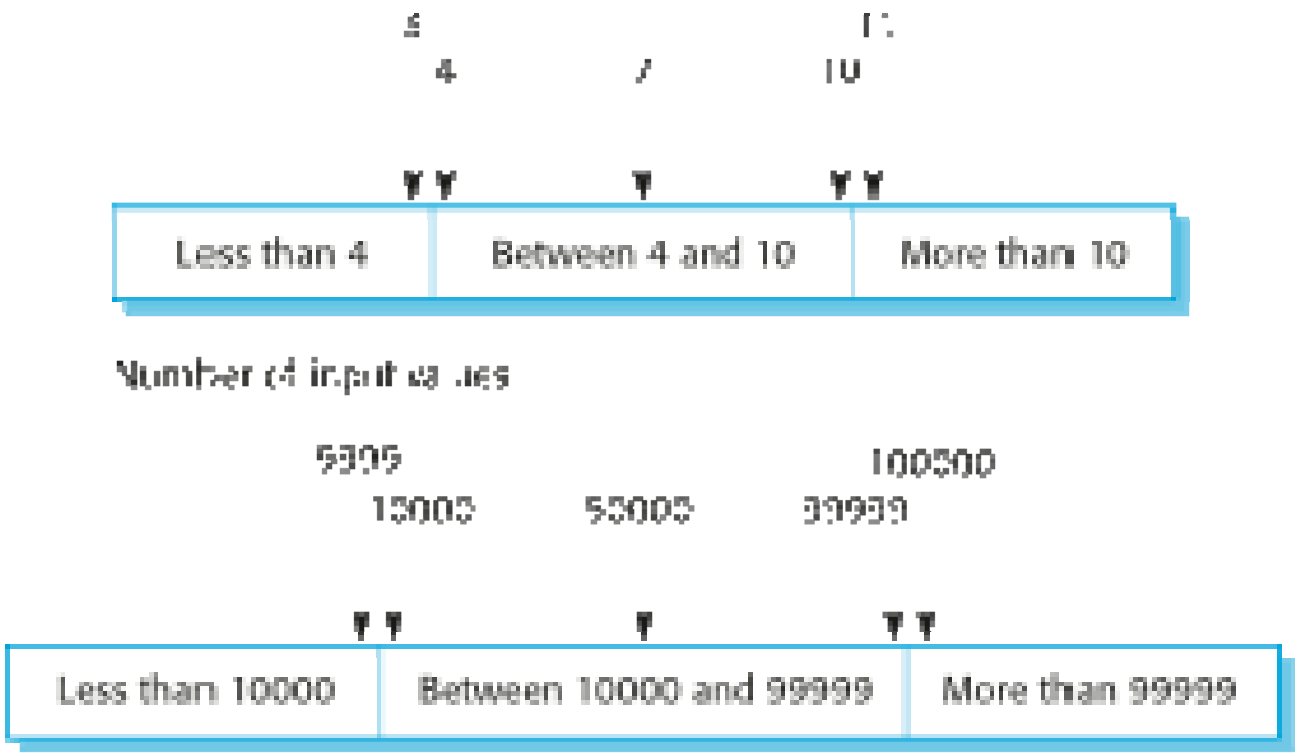
Testiranje po celinama

- Ulazni podaci i izlazni rezultati često dolaze iz različitih klasa čije su promenljive povezane.
- Za svaku od ovih različitih ulaznih parametara ili izlaznih rezultata aplikacija bi trebalo da se ponaša identično.
- Testovi bi trebalo da budu napisani za sve različite tipove parametara.

Ekvivalentna raspodela podataka



Ekvivalentna raspodela podataka



Smernice za testiranje (nizovi, liste)

- Obavezno testirajte softver sa nizovima koja imaju samo jednu vrednost ili nijednu.
  - Programeri često podrazumevaju da nizovi imaju više vrednosti, tako da često softver ne radi kada dobije samo jednu ili nijednu vrednost.
- Treba koristiti nizove različitih veličina u različitim testovima.
- Napravite testove tako da se pristupa prvim, srednjim i poslednjim elementima u nizu.

Opšte smernice za testiranje

- ✧ Izaberite ulaze koji će prinuditi sistem da generiše poruku o grešci.
- ✧ Dizajnirajte ulaze tako da prouzrokuju preopterećenje ulaznog bafera.
- ✧ Ponovite iste ulaze ili serije ulaza veliki broj puta.
- ✧ Forsirajte generisanje pogrešnih izlaza.
- ✧ Forsirajte izračunavanje rezultata koji su veoma veliki ili veoma mali.

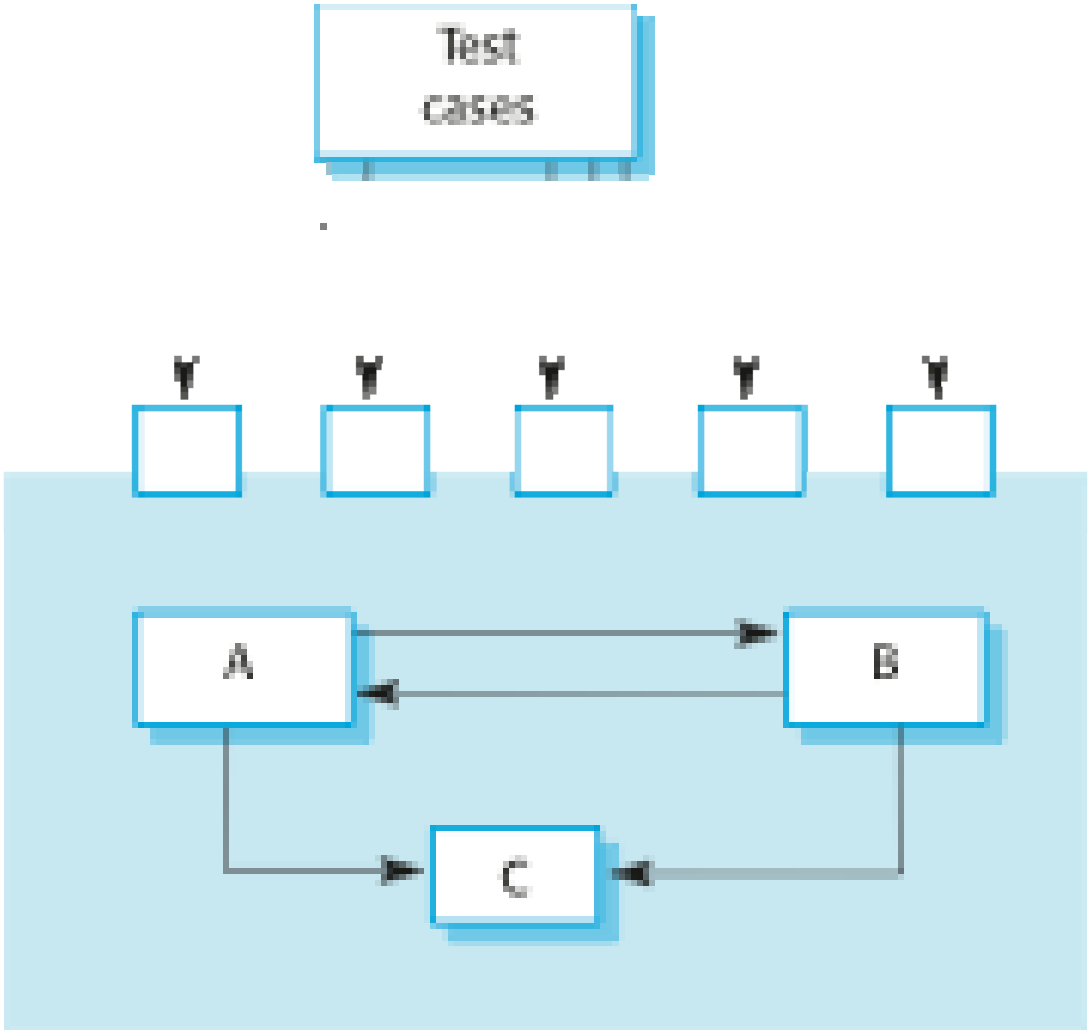
Rekapitulacija

- Testiranje može samo da pokaže prisustvo grešaka u aplikaciji. Testiranje ne može da demonstrira da nema zaostalih propusta u softveru.
- Razvojni tim je odgovoran za razvojno testiranje. Poseban tim treba da bude odgovoran za testiranje sistema pre nego što se isporuči korisnicima.
- Razvojno testiranje obuhvata testiranje jedinica, u kome treba obuhvatiti testiranje klasa i metoda, testiranje komponenti u kome treba testirati grupe klasa i testiranje sistema u kome treba testirati deo ili kompletan sistem.

Testiranje komponenti

- Softverske komponente su često složene komponente sastavljene od više objekata koji međusobno interaguju.
- Funkcionalnostima ovih objekata se pristupa kroz definisane interfejse komponenti.
- Testiranje složenih komponenti treba da se fokusira na to da pokaže da se interfejs komponente ponaša u skladu sa specifikacijom.

Testiranje interfejsa



Testiranje interfejsa

- Ciljevi su da se utvrde greške koje su posledice greške u interfejsu ili pogrešnih pretpostavki o interfejsu.
- Tipovi interfejsa
  - Parametarski interfejsi - Podaci se prosleđuju iz jedne metode ka drugoj.
  - Interfejs deljene memorije - Deli se blok memorije između funkcija. Npr. jedna podkomponenta smešta podatak u memoriju, a druga komponenta mu pristupa.
  - Proceduralni interfejsi - Podsistem enkapsulira skup procedura koje pozivaju drugi podsistemi.
  - Interfejsi za prosleđivanje poruka – Jedna komponenta zahteva servis druge komponente tako što joj prosleđuje poruku, a kao odgovor dobija rezultat izvršenja servisa.

Greške interfejsa

- Pogrešna upotreba interfejsa
  - Komponenta koja poziva drugu komponentu ali radi to na pogrešan način, npr. pogrešan redosled parametara.
- Pogrešno razumevanje interfejsa
  - Komponenta koja poziva očekuje određeno ponašanje dok interfejs ima potpuno drugačije ponašanje.
- Vremenske greške
  - Pozvana i pozivajuća komponenta funkcionišu različitim brzinama i informacije kojima se pristupa nisu ažurne.

Smernice za testiranje interfejsa

- ✧ Dizajnirajte testove tako da parametri kojima se poziva procedura idu do svojih krajnjih granica.
- ✧ Uvek testirajte parametre (objekte) sa null vrednostima.
- ✧ Dizajnirajte testove koji dovode do otkaza komponenti.
- ✧ Koristite “stres” testiranje u sistemima sa prosleđivanjem poruka.
- ✧ U sistemima sa deljenom memorijom menjajte redosled aktivacije komponenti.

Testiranje sistema

- Testiranje sistema u toku razvoja obuhvata integrisanje komponenti da bi se kreirale verzije sistema nakon čega se testira kompletan sistem.
- Fokus prilikom testiranja sistema je interakcija između komponenti.
- Testiranjem sistema se proverava da li su sve komponente kompatibilne, da li korektno interaguju i prenose prave podatke u pravo vreme preko interfejsa.
- Testiranje sistema proverava ponašanje sistema kao celine.

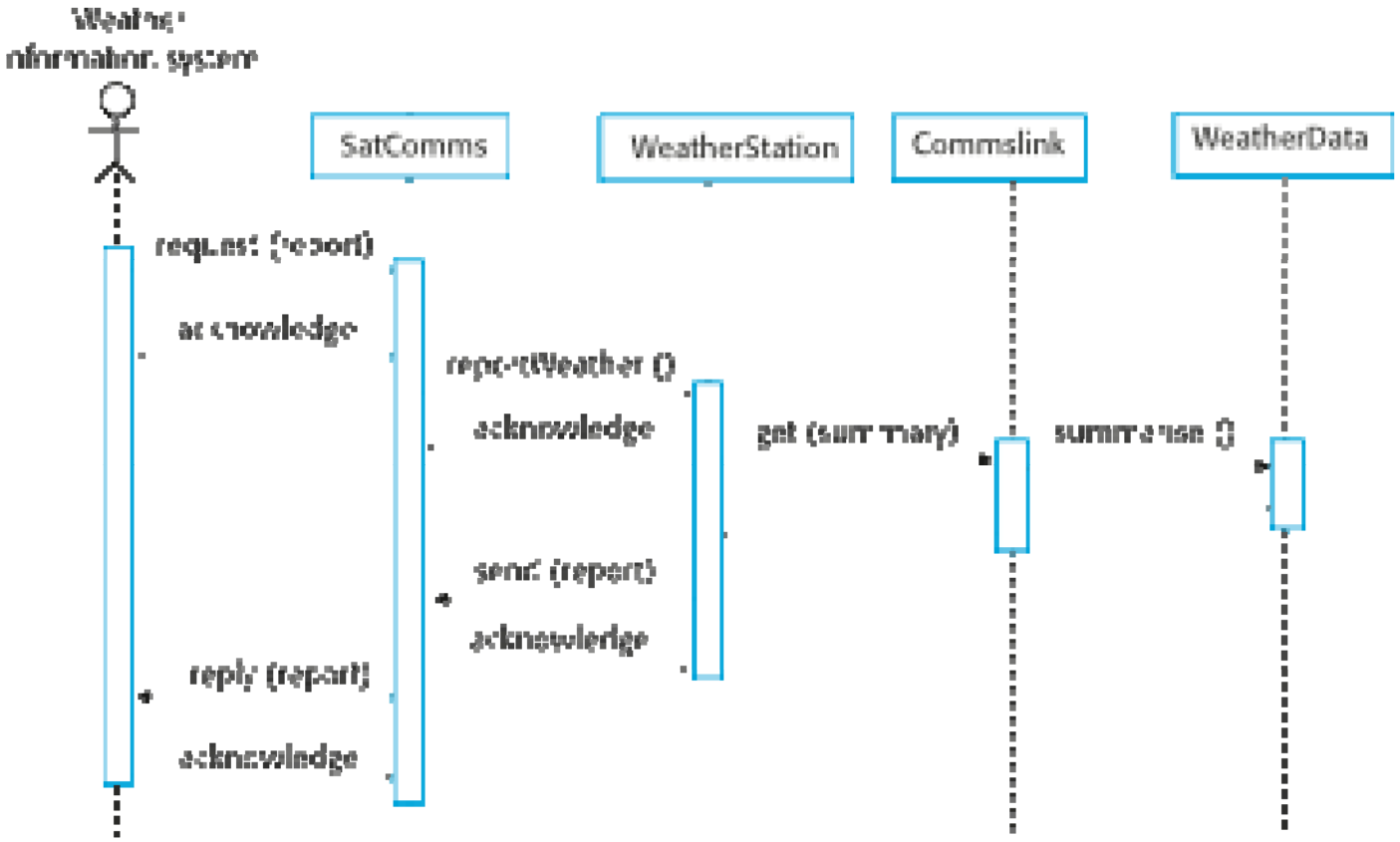
Testiranje sistema i testiranje komponenti

- Za vreme testiranja sistema, komponente koje se višestruko upotrebljavaju i koje su posebno razvijane se integrišu sa novim komponentama i testira se kompletan sistem.
- Komponente koje su razvijali različiti članovi tima ili podtimovi mogu se integrisati u ovoj fazi. Testiranje sistema je kolektivan, a ne individualan proces.
  - U nekim kompanijama, testiranje sistema može uključiti različite timove za testiranje bez uključivanje dizajnera i programera.

Testiranje slučajeva upotrebe

- Slučajevi upotrebe koji identifikuju interakcije sistema mogu se koristiti kao osnova za testiranje sistema.
- Svaki slučaj upotrebe obuhvata nekoliko sistemskih komponenti tako da testiranje slučajeva upotrebe forsira ove interakcije.
- Dijagrami sekvenci povezani sa slučajevima upotrebe treba da dokumentuju komponente i interakcije koje se testiraju.

Prikupljanje vremenskih podataka – dijagram sekvence



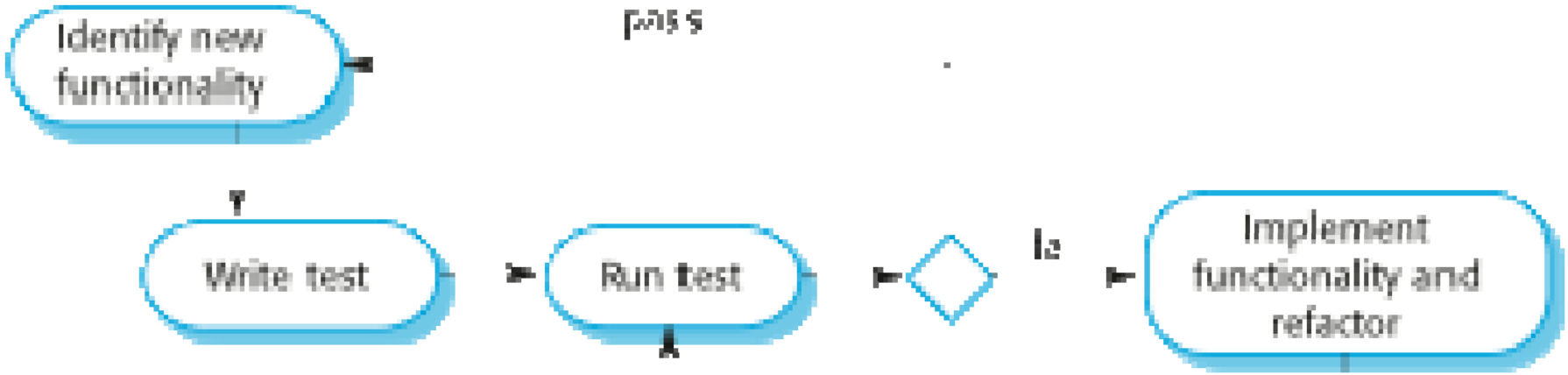
Pravila testiranja

- Potpuno testiranje sistema je nemoguće pa je potrebno poštovati pravila testiranja kojima se definiše pokrivenost sistema testovima.
- Primeri pravila testiranja:
  - Sve sistemske funkcije kojima se pristupa kroz menije moraju se testirati,
  - Kombinacije funkcija (npr. formatiranje teksta) kojima se pristupa kroz isti meni moraju biti testirane,
  - Tamo gde korisnik unosi neke podatke, sve funkcije se moraju testirati korišćenjem ispravnih i neispravnih ulaznih podataka.

Razvoj upravljan testovima (Test-driven development)

- Test-driven development (TDD) je pristup razvoju softvera u kome se preklapaju testiranje i programiranje
- Testovi se pišu pre samog koda i prolazak testova je kritična vodilja razvoja.
- Kod se razvija inkrementalno, zajedno sa testovima koji se uvećavaju. Ne prelazi se na sledeći inkrement dok kod ne prođe svoje testove.
- TDD je uveden kao deo agilnih metodologija kao što su Extreme Programming, ali se može koristiti i u drugim razvojnim procesima.

Razvoj upravljan testovima (Test-driven development)



TDD aktivnosti procesa

- Započinje se identifikovanje inkrementa funkcionalnosti koja se zahteva. Ovo bi trebalo da bude mali inkrement koji se može implementirati u nekoliko linija koda.
- Piše se test za ovu funkcionalnost i implementira se kao automatizovani test.
- Pokreće se test, zajedno sa drugim testovima koji su implementirani. U početku test neće proći s obzirom da nema implementiranih funkcionalnosti.
- Implementiraju se funkcionalnosti i ponovo pokreće test.
- Jednom kada su svi testovi izvršeni uspešno prelazi se na drugi deo funkcionalnosti.

Prednosti TDD razvoja

- ✧ Pokrivenost koda
  - Svaki segment koda koji je napisan ima bar jedan test.
- ✧ Regresivno testiranje
  - Testovi se razvijaju inkrementalno kako se program razvija. Regresivnim testiranjem se utvrđuje da izmene softvera nisu uvele nove bagove.
- ✧ Pojednostavljeno debagovanje
  - Kada test ne prođe proveru, trebalo bi da je očigledno u čemu je problem. Novokreirani kod treba da se proveri i modifikuje.
- ✧ Sistemska dokumentacija
  - Sami testovi su neka vrsta dokumentacije koja opisuje šta kod treba da radi.

Regresivno testiranje

- Regresivno testiranje je provera da li promene nisu “pokvarile” kod koji je prethodno radio.
- Kod manuelnog procesa testiranja, regresivno testiranje je skupo, dok sa automatizovanim testiranjem, je jednostavno i brzo. Svi testovi se ponovo pokreću kada se naprave izmene u programu.
- Testovi se moraju uspešno izvršiti pre nego što se “commit”-uju izmene.

Testiranje proizvoda

- Testiranje proizvoda je proces testiranja određene verzije sistema koja je namenjena za upotrebu van razvojnog tima.
- Primarni cilj procesa testiranja proizvoda da se ubedi kupac sistema da je sistem dovoljno dobar za upotrebu.
  - Testiranje proizvoda, stoga, treba da pokaže da sistem obezbeđuje određene funkcionalnosti, performanse i da je stabilan u toku normalne upotrebe.
- Testiranje proizvoda je obično black-box proces testiranja u kome se testovi izvode iz specifikacije sistema.

Testiranje proizvoda i testiranje sistema

- ✧ Testiranje proizvoda je vrsta testiranja sistema.
- ✧ Značajne razlike:
  - Poseban tim koji nije bio uključen u razvoj sistema treba da bude odgovoran za testiranje proizvoda.
  - Testiranje sistema koje izvršava razvojni tim treba da se fokusira na otkrivanje grešaka u sistemu. Cilj testiranja proizvoda je da se utvrdi da li sistem zadovoljava zahteve i da li je dovoljno dobar za eksternu upotrebu.

Testiranje zasnovano na zahtevima

- Testiranje zasnovano na zahtevima uključuje proveru svakog zahteva i razvoj testa ili testova za svaki od njih.
- Bolnički IS - zahtevi:
  - Ako je poznato da je pacijent alergičan na bilo koji određeni lek, onda prepisivanje leka treba da dovede do izbacivanja poruke upozorenja korisniku.
  - Ako korisnik izabere da ignoriše poruku o alergiji treba da da neki razlog zašto je to uradio.

Testiranje zahteva – bolnički IS

- ✧ Unesi zapis o pacijentu bez poznatih alergija. Prepiši lek za alergije za koje se zna da postoji. Proveri da sistem nije izbacio poruku upozorenja.
- ✧ Unesi zapis o pacijentu sa poznatom alergijom. Prepiši lek na koji je pacijent alergičan i proveri da li je izbačena poruka upozorenja.
- ✧ Unesi zapis o pacijentu u kome definisane alergije na 2 ili više lekova. Prepiši oba leka posebno i proveri da li se za svaki lek izbacuje upozorenje.
- ✧ Prepiši lek koji izbacuje upozorenje i ignoriši upozorenje. Proveri da li sistem zahteva da korisnik unese objašnjenje zbog čega je ignorisao upozorenje.

Testiranje performansi

- Deo testiranja proizvoda može obuhvatiti testiranje bitnih karakteristika sistema, kao što su performanse i pouzdanost.
- Testovi treba da reflektuju profil upotrebe sistema.
- Testovi performansi obično obuhvataju planiranje serija testova kod kojih se postepeno povećava opterećenje sve dok performanse sistema ne postanu neprihvatljive.
- “Stres” testiranje je oblik testiranja performansi gde se namerno testira sistem pod maksimalnim opterećenjem kako bi namerno doveli do otkaza.

Korisničko testiranje

- Korisničko testiranje je faza u procesu testiranja u kojoj korisnici obezbeđuju ulaze i savete za testiranje sistema.
- Korisničko testiranje je veoma bitno, čak i kada je detaljno testiranje sistema i proizvoda već izvršeno.
  - Razlog za ovo je taj što korisnikovo radno okruženje ima velike efekte na pouzdanost, performanse, upotrebljivost i robustnost sistema. Ovo se ne može replikovati u okruženju za testiranje.

Vrste korisničkog testiranja

- Alfa testiranje
  - Korisnici rade sa razvojnim timom da bi testirali softver na razvojnim stanicama.
- Beta testiranje
  - Korisnicima se isporučuje proizvod kako bi mogli da eksperimentišu sa njim i ustanove probleme koje razvojni tim nije pronašao.
- Testiranje prihvatljivosti
  - Korisnici testiraju sistem da bi odlučili da li je spreman da bude prihvaćen i postavljen u korisničko okruženje za upotrebu.

Faze u procesu testiranja prihvatljivosti

- ✧ Definisanje kriterijuma prihvatljivosti
- ✧ Plan testiranja prihvatljivosti
- ✧ Izvođenje testova prihvatljivosti
- ✧ Pokretanje testova prihvatljivosti
- ✧ Analiza rezultata testova
- ✧ Odbijanje/prihvatanje sistema

Agilne metode i testiranje prihvatljivosti

- U agilnim metodama, korisnik je deo razvojnog tima i odgovoran je za donošenje odluka o prihvatljivosti sistema.
- Testovi su definisani sa strane korisnika i integrisani su sa drugim testovima na taj način da se izvršavaju automatski kada se naprave promene.
- Ne postoji odvojen proces testiranja prihvatljivosti.
- Osnovna dilema je da li je uključeni korisnik “tipičan” i može zastupati interese stejkholdera.

Automatizacija testiranja

JUNIT

Automatizacija testiranja

- Softver koji omogućava automatizaciju bilo kog aspekta testiranja
  - Generišu se test inputi i očekivani rezultati
  - Pokreću se test paketi bez ručne intervencije
  - Evaluaira se prošao/nije prošao
- Testovi moraju biti automatizovani da bi bili efikasni i ponovo upotrebljivi

Prednosti automatskog testiranja

- Omogućava brzu i efikasnu verifikaciju popravljenih bagova
- Ubrzava debagovanje i smanjuje potrebu da otklanjanjem bagova
- Omogućava konzistentno prikupljanje i analizu rezultata testa
- Troškovi testiranja se vraćaju kroz povećanu produktivnost i bolji kvalitet softvera
- Bolje je uložiti više vremena u izradu kvalitetnijih testova, nego više puta izvršavati manje kvalitetne testove

Ograničenja i nedostaci

- Automatizovane testove je skupo kreirati i održavati
- Ukoliko se implementacija često menja, održavanje paketa testova može biti naporno

JUnit

- JUnit predstavlja framework za pisanje testova
  - Kreirali su ga Erich Gamma (jedan od tvoraca dizajn paterna) i Kent Beck (kreator XP metodologije)
  - JUnit pomaže programeru da:
    - Definiše i izvršava testove i pakete testova
    - Formalizuje zahteve
    - Piše i debaguje kod
    - Integriše kod i uvek je spreman da ima radnu verziju
  - BlueJ, JBuilder, i Eclipse obezbeđuju JUnit alate

Terminologija

- Fiksni test (test fixture) kreira podatke (objekte i atribute) koji su nam potrebni za svaki test
  - Npr. ukoliko testiramo kod koji menja podatke o studentu, potreban nam je podatak o studentu na kome ćemo testirati
- Test jedinice (unit test) je test jedne klase
- Test slučaj (test case) testira odgovor jedne metode na određeni skup ulaza
- Skup testova (test suite) je skup testova slučajeva
- Izvršilac testa (test runner) je softver koji pokreće testove i prikazuje rezultate

Struktura Junit test klase

- Pretpostavimo da želimo da testiramo klasu koja se zove Fraction
- Kreiraćemo odgovarajuću test klasu FractionTest

```
public class FractionTest
    extends junit.framework.TestCase {
                ...
}
```

Struktura Junit test klase

- protected void setUp()
  - Kreira fiksni test Creates inicijalizacijom objekata i vrednosti
- protected void tearDown()
  - Oslobađa bilo koji sistemski resurs koji je koristio test fixture
- public void testAdd()
  public void testToString()
  - Ove metode sadrže testove za metode klase Fraction add(), toString(), itd. Bitno je da njihovo ime počinje sa test

Šta radi JUnit test

- Za svaku test metodu t u paketu testova:
  - JUnit poziva setUp()
    - Kreira sve potrebne objekte
  - JUnit poziva t
    - Bilo koji izuzetak tokom njegovog izvršenja je logovan
  - JUnit poziva tearDown()
    - čišćenje, npr. zatvaranje fajlova, brisanje podataka e.g. close files
- Izveštaj za sve test slučajeve se predstavlja tekstualno ili grafički

Pisanje JUnit test klasa (1)

- Započnite importovanje JUnit 4 klasa:

```
import org.junit.*;
import static org.junit.Assert.*; // note static import
```

- Deklarišite svoju test klasu na uobičajen način

```
public class MyProgramTest {
```

- Deklarišite instancu klase koju testirate
- Možete deklarisati i druge promenljive ali im ne dodeljujte vrednosti

```
public class MyProgramTest {
    MyProgram program;
    int someVariable;
```

Pisanje JUnit test klasa (2)

- Definišite metodu (ili više metoda) koje će se izvršiti pre svakog testa
- Inicijalizujte svoje promenljive u ovoj metodi, tako da svaki test počinje novim svežim vrednostima

```
@Before
public void setUp() {
    program = new MyProgram();
    someVariable = 1000;
}
```

- Možete definisati jednu ili više metoda koje će se izvršavati nakon svakog testa
- Obično ovakve metode oslobađaju resurse, kao što su fajlovi
- Obično nemate potrebe da se opterećujete ovom metodom

```
@After
public void tearDown() {
}
```

Jednostavan primer

- Pretpostavimo da imamo klasu Arithmetic sa metodama int multiply(int x, int y), i boolean isPositive(int x)

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

- ```
  public class ArithmeticTest {

      @Test
      public void testMultiply() {
          assertEquals(4, Arithmetic.multiply(2, 2));
          assertEquals(-15, Arithmetic.multiply(3, -5));
      }

      @Test
      public void testIsPositive() {
          assertTrue(Arithmetic.isPositive(5));
          assertFalse(Arithmetic.isPositive(-5));
          assertFalse(Arithmetic.isPositive(0));
      }

  }
  ```

Assert metode (1)

- Unutar testa,
  - Poziva metodu koja se testira i vraća rezultat
  - **Assert** proverava šta treba da bude ispravan rezultat korišćenjem jedne od assert metoda
  - Ovi koraci se mogu ponavljati neograničen broj puta
- Assert metoda je JUnit metoda koja izvršava test i ispaljuje AssertionError ukoliko test ne prođe
  - JUnit hvata ove izuzetke i vraća vam rezultat
- static void assertTrue(boolean ***test***)
  static void assertTrue(String ***message***, boolean ***test***)
  - Ispaljuje AssertionError ukoliko test ne prođe
  - Opciona poruka je uključena u grešku
- static void assertFalse(boolean ***test***)
  static void assertFalse(String ***message***, boolean ***test***)
  - Ispaljuje AssertionError ukoliko test ne prođe

Primer: **Counter** kasa

- Pretpostavimo da kreiramo jednostavnu klasu Counter
  - Konstruktor će kreirati counter i postaviti ga na 0
  - Metoda increment će dodati 1 na vrednost counter-a i vratiti novu vrednost
  - Metoda decrement će umanjiti za 1 vrednost counter-a i vratiti novu vrednost
- Napisaćemo test metode pre nego što napišemo kod
  - Ovakav pristup ima brojne već navedene prednosti
  - Ipak, obično ćemo napisati prvo zaglavlje metode, i dozvolite IDE okruženju da generiše zaglavlje test metode.

JUnit testovi za **Counter**

```
public class CounterTest {
    Counter counter1; // declare a Counter here

    @Before
    void setUp() {
        counter1 = new Counter(); // initialize the Counter here
    }

    @Test
    public void testIncrement() {
        assertTrue(counter1.increment() == 1);
        assertTrue(counter1.increment() == 2);
     }

    @Test
    public void testDecrement() {
        assertTrue(counter1.decrement() == -1);
    }
}
```

- Svaki test počinje sa potpuno novim counter-om
- Ovo znači da ne morate da vodite računa o redosledu izvršenja testova

Counter klasa

```
public class Counter {
 int count = 0;

 public int increment() {
    return count += 1;
 }

 public int decrement() {
    return count -= 1;
 }

      public int getCount() {
         return count;
      }
}
```

- Da li je JUnit testiranje za ovako malu klasu preterivanje?
- Extreme Programming gledište je: *Ako nije testirano, onda ne radi*
- Malo je verovatno da ćete imati dosta ovako trivijalnih klasa u realnom programu
- Obično XP programeri ne pišu testove za jednostavne getter metode kao što je getCount()
- U ovom primeru se koristi samo assertTrue ali postoje i druge assert metode

Upozorenje: equals

- Primitive možete upoređivati korišćenjem ==
- Java ima metodu ***x***.equals(***y***), za upoređivanje objekata
  - Ova metoda se koristi za String klase i nekoliko drugih klasa
  - Za objekte klasa koje kreirate, morate definisati equals
- assertEquals(***expected***, ***actual***)
- Da definišete equals za sopstvene objekte, definišite sledeću metodu:
  public boolean equals(Object ***obj***) { ***...*** }
  - Argument mora biti tipa Object, koji nije ono što želite pa morate kastovati (cast) da bi bio ispravnog tipa (npr., Person):
  - ```
    public boolean equals(Object something) {
        Person p = (Person)something;
        return this.name == p.name; // test whatever you like here
    }
    ```
    ```

Assert metode (2)

- assertEquals(***expected***, ***actual***)
  assertEquals(String ***message***, ***expected***, ***actual***)
  - ***expected*** i ***actual*** moraju biti objekti ili isti tip primitivne vrednosti
  - Za objekte, koristie se vaše equals metode, ukoliko su definisane na ispravan način

- assertSame(Object ***expected***, Object ***actual***)
  assertSame(String ***message***, Object ***expected***, Object ***actual***)
  - Utvrđuje da li dva argumenta ukazuju na isti objekat

- assertNotSame(Object ***expected***, Object ***actual***)
  assertNotSame(String ***message***, Object ***expected***, Object ***actual***)
  - Utvrđuje da li dva objekta ne ukazuju na isti objekat

Assert metode (3)

- assertNull(Object ***object***)
  assertNull(String ***message***, Object ***object***)
  - Utvrđuje da li je vrednost objekta null (nedefinisana)

- assertNotNull(Object ***object***)
  assertNotNull(String ***message***, Object ***object***)
  - Utvrđuje da li objekat nije null

- fail()
  fail(String ***message***)
  - Uzrokuje da je test neuspešan i ispaljuje izuzetak AssertionFailedError
  - Korisno je kao rezultat složenog testa, kada druge assert metode nisu ono što želite

Pisanje JUnit test klasa (3)

- **Ovaj deo se odnosi samo na skupo podešavanje, kao što je povezivanje sa bazom podataka koje je neophodno za testiranje**
  - Ukoliko je to potrebno, možete deklarisati jednu metodu koja će se izvršiti samo jednom kada se klasa prvi put pokrene

```
@BeforeClass
public static void setUpClass() throws Exception {
        // one-time initialization code
}
```

  - Ukoliko želite možete deklarisati jednu metodu koja će se izvršiti samo jednom kada su svi testovi završeni

```
@AfterClass
public static void tearDownClass() throws Exception {
        // one-time cleanup code
}
```

Specijalne osobine @Test

- Možete definisati koliko dugo metoda može da se izvršava
- Ovo je dobra zaštita od beskonačnih petlji
- Vremensko ograničenje je definisano u milisekundama
- Test je neuspešan ako se metoda izvršava previše dugo

```
@Test (timeout=10)
 public void greatBig() {
    assertTrue(program.ackerman(5, 5) > 10e12);
 }
```

- Pozivi nekih metoda bi trebalo da ispaljuju izuzetak
- Možete definisati da je određeni izuzetak očekivan
- Test će proći ukoliko je izuzetak ispaljen, u suprotnom je neuspešan

```
@Test (expected=IllegalArgumentException.class)
public void factorial() {
    program.factorial(-5);
}
```

Ignorisanje testa

- @Ignore anotacija definiše da se test ne pokreće

```
@Ignore("I don't want Dave to know this doesn't work")
@Test
public void add() {
    assertEquals(4, program.sum(2, 2));
}
```

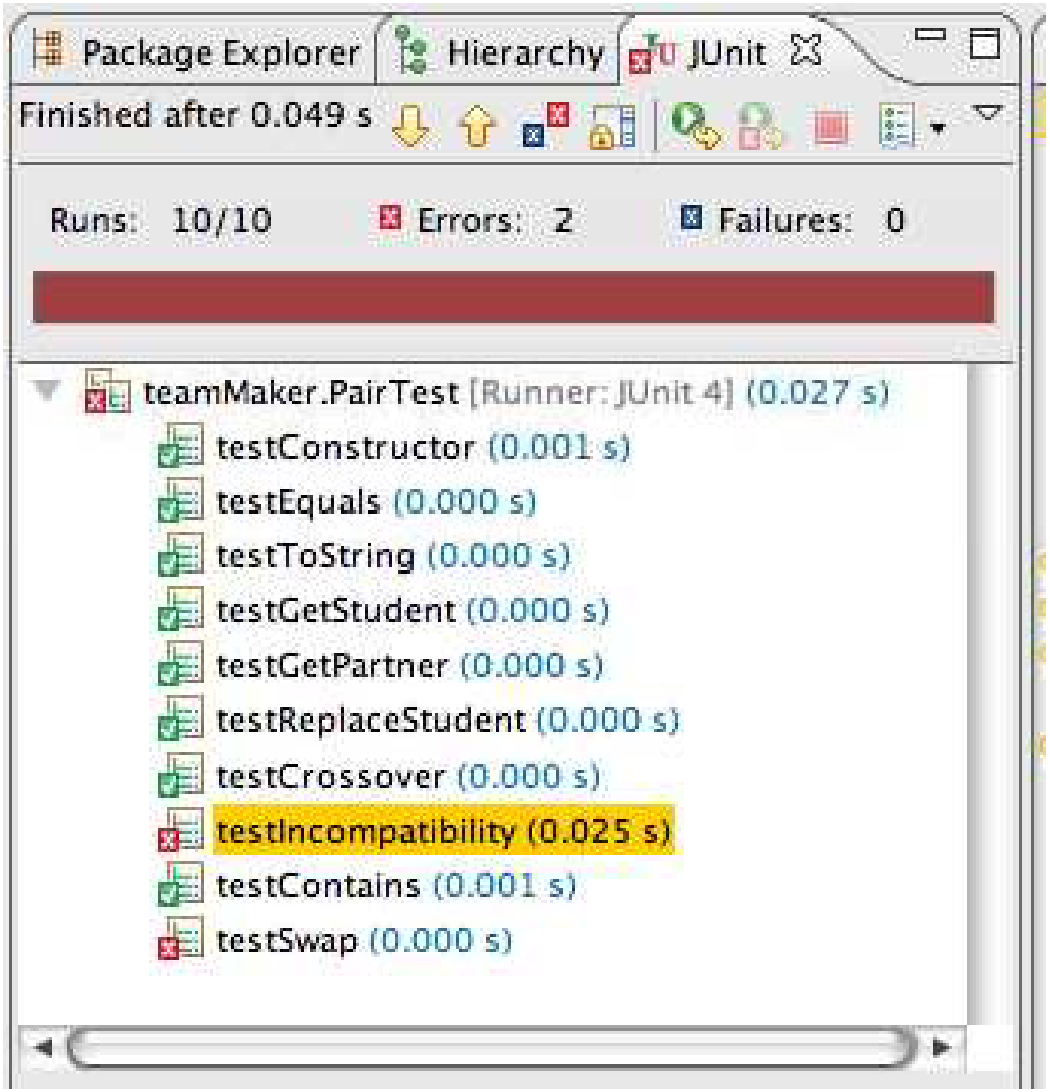
- Ne treba da koristite @Ignore bez dobrog razloga

Test paketi

- Možete definisati skup paketa

```
@RunWith(value=Suite.class)
@SuiteClasses(value={
                    MyProgramTest.class,
                    AnotherTest.class,
                    YetAnotherTest.class
                })
public class AllTests { }
```

Pregled rezultata u Ecliplse-u



Rezime

- U toku testiranja softvera, treba da probate da “razvijete” softver korišćenjem iskustva i saveta za izbor vrsta testova koji su korisni u otkrivanju grešaka u sistemu.
- Kada god je moguće treba pisati automatizovane testove. Testove treba ugnježdavati u program tako da se izvršavaju svaki put kada se napravi izmena u sistemu.
- Test-driven development je pristup razvoja kod koga se prvo prave testovi pa onda kod.
- Testiranje scenarija obuhvata pronalaženje tipičnog scenarija upotrebe i njegova upotreba za pravljenje testova.
- Testiranje prihvatljivosti je proces testiranja koji izvode korisnici sa ciljem da odluče da li je softver dovoljno dobar da bi bio korišćen u njihovom operativnom okruženju.