



Funded by the
Erasmus+ Programme
of the European Union



This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



KI206 - PROCES I METODOLOGIJE RAZVOJA SOFTVERA

Testiranje softvera

Lekcija 07

PRIRUČNIK ZA STUDENTE

KI206 - PROCES I METODOLOGIJE RAZVOJA SOFTVERA

Lekcija 07

TESTIRANJE SOFTVERA

- ✓ Testiranje softvera
- ✓ Poglavlje 1: Šta je testiranje softvera?
- ✓ Poglavlje 2: Testiranje u razvoju
- ✓ Poglavlje 3: Testiranje jedinice
- ✓ Poglavlje 4: Testiranje komponenata
- ✓ Poglavlje 5: Testiranje sistema
- ✓ Poglavlje 6: Testovima vođen razvoj sistema
- ✓ Poglavlje 7: Testiranje pri primopredaji sistema
- ✓ Poglavlje 8: Korisničko testiranje
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

✓ Uvod

UVOD

Ova lekcija vam omogućava da:

- razumete faze testiranja softvreskog sistema, počev od testiranja za vreme razvoja, pa do testiranja sistema kod kupca;
- se upoznate sa tehnikama koje će vam pomoći da izaberete testova koja omogućavaju otkrivanje programskih defekata;
- razumete razvoj koji počinje sa testom, tj. u kome vi projektujete test pre nego što pišete kod i sprovodite testove automatski;
- znate važne razlike između testiranja komponenata, sistema, i testiranja verzije za isporuku, kao i da razumeju procese i tehnike testiranja.

✓ Poglavlje 1

Šta je testiranje softvera?

DVA CILJA PROCESA TESTIRANJA

Testiranje služi da se utvrdi da li program radi ono bi trebalo da radi i da se otkriju greške u programu pre nego što se pusti u upotrebu

Testiranje služi da se utvrdi da li program radi ono bi trebalo da radi i da se otkriju greške u programu pre nego što se pusti u upotrebu. Testiranje se vrši puštanjem da softver radi, pri čemu se koriste „veštački“ podaci. Onda se proveravaju rezultati testiranja da bi se videlo da li oni ukazuju na neku grešku, nepravilnost i da bi se došlo do informacija o nefunkcionalnim svojstvima softvera.

Proces testiranja ima dva cilja:

1. *Da pokaže inženjeru razvoja i kupcu da softver ostvaruje svoje zahteve.* Za softver koji se radi po posebnom zahtevu, sprovodi se po najmanje jedan test za svaki od zahteva iz dokumenta sa zahtevima. U slučaju opštih softverskih proizvoda, testovi treba da provere sva svojstva softvera, kao i kombinacije ti svojstava, onako kako su realizovana u proizvodu koji se pušta u prodaju.
2. *Da otkrije situacije u kojima ponašanje softvera nije ispravno,* nepoželjno ili da ne odgovara specifikaciji softvera. To su posledice defekata u softveru (pad sistema, neželjene interakcije sistema sa drugim sistemima, netačni proračuni, i nekontrolisana promena podataka), čiji razlozi treba da se testiranjem pronađu.

Prvi cilj se ostvaruje sprovođenje testova za proveru ispravnosti softvera, tj. testovima validacije (engl., **validation testing**). Drugi cilj se ostvaruje testiranjem na defekte (engl., **defect testing**). U praksi, ove dve vrste testiranja se mešaju, jer i pri sprovođenju testova za proveru ispravnosti, otkrivaju se defekti (greške) u softveru, a i pri testiranju na defekte, vidi se da li program zadovoljava zahteve.

MODEL TESTIRANJA PROGRAMA

Testiranje je samo deo šireg procesa provere (verifikacije) i potvrđivanja (validacije) ispravnosti softvera

Dijagram na slici 1 pokazuje razlike između testiranja ispravnosti (validacije) i testiranja defekata. Sistem prihvata skup ulaznih podataka U i generiše skup izlaznih rezultata I . Neki od rezultata su pogrešni, i označeni su sa I_g . Ti rezultati sa greškom su dobijeni pri ulaznim podacima U_g . Cilj testiranje na defekte je da se otkriju ti skupovi ulaznih podataka U_g koji

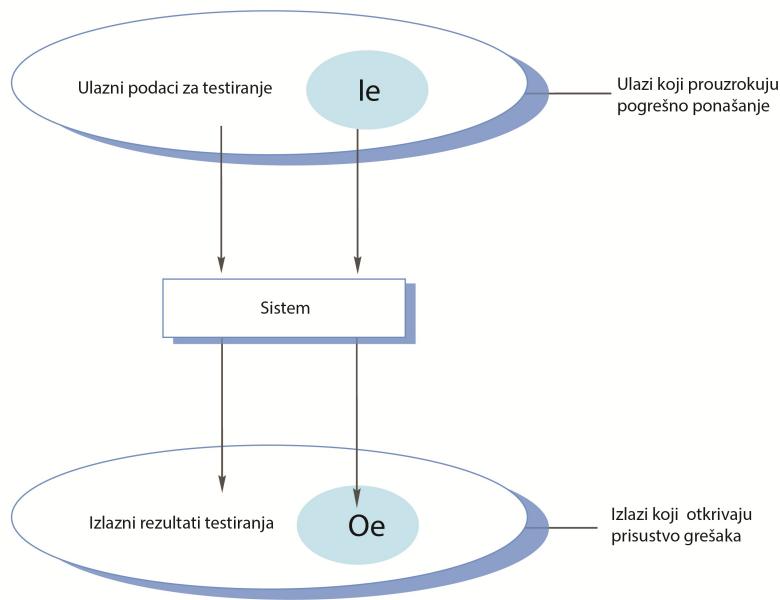
dovode da neispravnog rada sistema. S druge strane testovi na ispravnost (testovi validacije) se sprovode sa ispravnim skupom podataka (koji nisu u skupu U_g) jer se stimuliše da sistem da samo tačne rezultate.

Testiranje ne može da pokaže da softver nema defekata ili da će se ponašati u skladu sa specifikacijom pri svim uslovima. Može se destiti da testiranje nije obuhvatilo neki test koji bi pokazao na neku grešku. Testiranje može samo da pokaže prisustvo grešaka, a ne i njihovo nepostajanje.

Testiranje je samo deo šireg procesa provere (verifikacije) i potvrđivanja (validacije) ispravnosti softvera. Koja je razlika između provere (verifikacije) i potvrđivanja ispravnosti (validacije)?

1. *Potvrđivanje ispravnosti - validacija* (engl. validation): Da smi razvili ispravan proizvod?

2. *Provera - verifikacija* (engl. verification): Da razvijamo proizvod na ispravan način?



Slika 1.1 Model testiranja programa

CILJEVI VERIFIKACIJE I VALIDACIJE SOFTVERA

Krajnji cilj procesa verifikacije i validacije je utvrđivanje da li softver zadovoljava svoju svrhu, tj. da je sistem dobar za planiranu upotrebu.

Proces verifikacije i validacije služi za proveru da li softver u razvoju zadovoljava svoju specifikaciju i da li obezbeđuje funkcionalnost koju očekuju njegovi kupci. Ovaj proces počinje odmah po utvrđivanju zahteva, i sprovodi se u svim fazama procesa razvoja softvera.

Cilj verifikacije je provera da li softver zadovoljava postavljene funkcionalne i nefunkcionalne zahteve. Validacija je opštiji proces. **Cilj validacije** je da potvrdi da softver zadovoljava očekivanja kupca. Ako specifikacija zahteva nije dobro napravljena, može se

desiti da verifikacija bude pozitivna (softver zadovoljava sve zahteve) a da validacija bude negativna (softver ne zadovoljava očekivanja kupca). Zato je validacija i najbitnija.

Krajnji cilj procesa verifikacije i validacije je utvrđivanje da li softver zadovoljava svoju svrhu, tj. da je sistem dobar za planiranu upotrebu. Nivo poverenja koji se zahteva od softvera, zavisi od svrhe softvera, od očekivnja korisnika sistema, i od trenutnog tržišnog okruženja sistema:

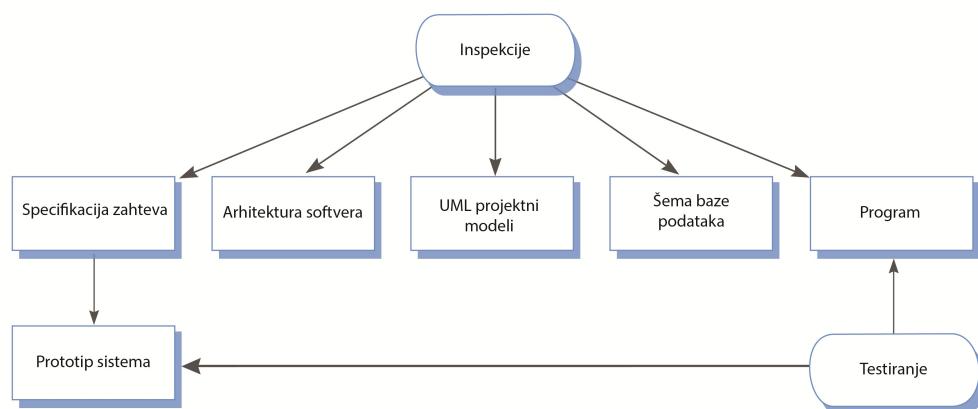
1. **Svrha softvera:** Što je softver važniji (kritičan), to je važnije da bude pouzdan. Na primer, sistemi kod kojih je bezbednost kritična, imaju veći nivo zahtevanog poverenja nego softver razvijen kao prototip koji treba da pokaže ideje o novom proizvodu.
2. **Očekivanja korisnika:** Korisnici najčešće tolerišu greške u novom softveru, ali kasnije, očekuju da on postane pouzdan, tj. da se bitno smanji broj grešaka.
3. **Tržišno okruženje:** Vrši se upoređenje sa konkurenčkim proizvodima, utvrđuje se koliko je kupac spremjan da plati za proizvod, kao i koji su zahtevani rokovi za isporuku softvera. U želji da se novi proizvod što pre pojavi na tržištu, kompanije često puste proizvod koji još nije dovoljno testiran. Ako je proizvod jeftin, onda i kupci nisu vrlo zahtevni, te tolerišu i greške.

INSPEKCIJA SOFTVERA

Inspekcije softvera podržava proces verifikacije i validacije u različitim fazama procesa razvoja softvera.

Proces verifikacije i validacije sadrži i inspekcije (preglede) i recenzije softvera. Ove aktivnosti analiziraju i proveravaju sistemske zahteve, projektne modele, izvorni kod programa, i predložene testove sistema. To su tzv. „statičke“ tehnike verifikacije i validacije, jer ne zahtevaju da softver radi, da bi bio proveren.

Slika 2 pokazuje kako inspekcije softvera podržava proces verifikacije i validacije u različitim fazama procesa razvoja softvera. Strelice pokazuju faze procesa u kojima se može vršiti inspekcija softvera.



Slika 1.2 Inspekcija i testiranje softvera

PREDNOSTI INSPEKCIJE U ODNOSU NA TESTIRANJE

Inspekcija bolje otkriva greške nego testiranje programa.

Inspekcije se najčešće usmeravaju na izvorni kod sistema, ali i na ostala dokumenta, kao što je specifikacija zahteva i projektni model. Pri inspekciji, koristi se znanje o sistemu, njegovom domenu primene, a i znanje o programiranju ili modeliranju, da bi se otkrile greške.

Inspekcija pruža tri prednosti u odnosu na testiranje:

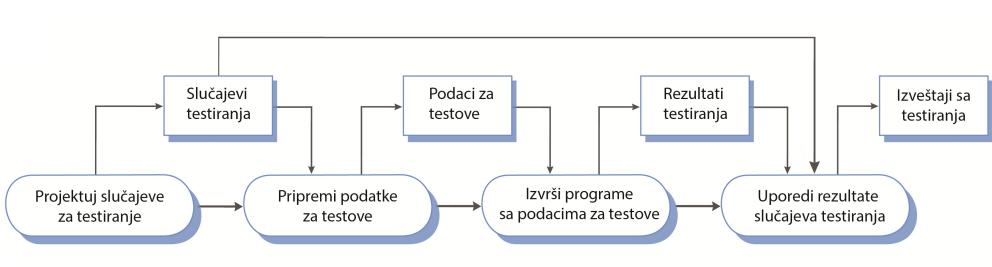
1. *Za vreme testiranja, greške mogu da prikriju druge greške.* Kada se dobije pogrešan rezultata, nikad se ne zna pouzdano da li je to posledica jedne ili neke druge greške. Kako je inspekcija statički proces, on ne zavisi od interakcije među grešaka. Jedna inspekcija, zbog toga, može da otkrije mnoge greške u sistemu.
2. *Nekompletne verzije softverskog sistema se mogu proveravati bez dodatnih troškova.* Iako je program nekompletan, vi koristite prednost specijalizovanih testova za testiranje samo delova softvera koji je razvijen. Naravno, na ovaj način se uvećavaju troškovi razvoja softvera.
3. *Pored traženja defekata, inspekcija može uzeti u obzir i šire atribute kvaliteta programa,* kao što je usaglašenost sa standardima, prenošljivost (na različite sisteme) kao i lakoća održavanja. Analizirate neefikasnost u radu, neodgovarajuće algoritme ili loš način programiranja koji vodi ka dobijanju sistema koji će biti težak za održavanje i menjanje

Istraživanja su pokazala da je inspekcija bolje otkriva greške nego testiranje programa. Naravno, inspekcija ne može da zameni testiranje, ali omogućava da se veći broj grešaka otkrije i pre testiranja. Inspekcije nisu dobre u otkrivanju defekata koji su rezultata neočekivanih interakcija između različitih delova softverskog sistema, usled problema sa vremenom, ili problema sa performansama sistema. Pored toga, u malim kompanijama je teško da se formira poseban tim za inspekciju softvera, zbog nedostatka ili ljudi ili novca.

MODEL PROCESA TESTIRANJA SOFTVERA

Slučajevi za testiranje i specifikacije su ulaz u testiranje. Izvršenje testova se može automatizovati i dobijeni rezultati automatski upoređivati sa očekivan rezultatima.

Na slici 3 je prikazan apstraktni model „tradicionalnog“ procesa testiranja softvera. Slučajevi za testiranje i specifikacije su ulaz u testiranje. Priprema testa se ne može automatizovati, ali izvršenje testova može. Dobijeni rezultati se automatski upoređuju sa očekivanim rezultatima.



Slika 1.3 Model “tradicionalnog” procesa testiranja softvera

TRI FAZE TESTIRANJA

Tri faze testiranja su: 1) Razvojno testiranje, 2) Testiranje softvera za isporuku i 3) Korisničko tetsiranje

Komercijalni softveri najčešće prolaze kroz tri faze testiranja:

1. **Razvojno testiranje**, kada se softver testira za vreme razvoja, a da bi se otkrile greške i defekti. To najčešće rade projektanti sistema i programeri.
2. **Testiranje softvera za isporuku**, kada poseban tim testira kompletну verziju softvera pre nego što se isporuči korisnicima. Cilj je da se proveri da li sistem zadovoljava zahteve svih aktera.
3. **Korisničko testiranje**, kada sadašnji ili budući korisnici softvera vrše testiranje u svom radnom okruženju. U slučaju softverskih proizvoda, ovo može da radi unutrašnja grupa iz marketinga koja odlučuje da li proizvod spremан и добар да ide na tržiste. Test prihvatanja (engl., acceptance testing) je tip korisničkog testiranja kada kupca formalno testira sistem da bi odlučio da li da ga prihvati ili da traži dodatni razvoj.

U praksi, najčešće se kombinuje ručno i automatizovano testiranje. Kod ručnog testiranja, tester koristi program sa odabranim podacima za testiranje i onda upoređuje dobijene i očekivanje rezultate. Kod automatskog testiranja, testiranje se programira u samom kodu softvera. To je brži način testiranja, naročiti pri regresionom (ponovljenom) testiranju, kada se vrši provera efekata promena koje su unete u program.

STANDARDI ZA TESTIRANJE SOFTVERA

Primena standarda u ratvoju softvera obezbeđuje njihov kvalitet

Ovde se neki od standarda IEEE i Britanskih standarda koji pokrivaju obezbeđenje kvaliteta i testiranje softvera. Standardi se dobijaju pretplatom. Videti: <http://www.standards.ieee.org/software/index.html> za IEEE standarde i [bsonline.techindex.co.uk](http://www.bsonline.techindex.co.uk) za Britanske standarde.

- ISO 9126, Software Product Quality Characteristics
- IEEE Standard 730, Software Quality Assurance Plans
- IEEE Standard 829, Software Test Documentation
- IEEE Standard 1012, Software Verification and Validation

- IEEE Standard 1028, Software Reviews
- British Standard 7925, Software Testing

PRIMER PLANIRANJA TESTIRANJA SISTEMA ZA UPRAVLJANJE INSULIN PUMPOM

Primer planiranja testiranja softvera i određivanja potrebnih resursa po fazama.

Na slici 4 prikazan je proces planiranja testiranja po fazama i trajanje faza u danima. Prva faza obuhvata izradu plana testiranja, odabir tehnika testiranja i odabir okruženja u kome će se izvršiti testiranje. Sledeća faza je definisanje test slučajeva gde je potrebno za svaki slučaj korišćenja prethodno definisan u korisničkim zahtevima osmisliti test slučaj. U ovoj fazi potrebno je obuhvatiti glavne funkcionalnosti sistema kao što su merenje nivoa šećera u krvi, davanje doze insulina itd.

Na osnovu definisanih test slučaja prelazi se na sledeću fazu koja obuhvata dizajniranje testova na osnovu test slučajeva (koji su opisani i postavljeni u prethodnoj fazi). Faze četiri, pet i šest (jedinično, integraciono i sistemsko testiranje) obuhvataju testiranje kompletног sistema na osnovu test slučajeva i dobijanje prvih rezultata testiranja. Rezultati omogućavaju dorade i izmene u samom sistemu pre faza alfa i beta testiranja u kojima se sistem testira pre davanja na korišćenje krajnjim korisnicima. Krajnji korisnici mogu izvršiti testiranje prihvatljivosti koje će biti predstavljeno u narednim primerima.

Faza	Trajanje u danima
Izrada plana testiranja	15 dana
Definisanje test slučajeva	7 dana
Dizajniranje testova	10 dana
Jedinično testiranje	30 dana
Integraciono testiranje	20 dana
Sistemsko testiranje	50 dana
Alfa testiranje	10 dana
Beta testiranje	25 dana

Slika 1.4 Faze testiranja softvera

U toku planiranja testiranja softvera potrebno je izvršiti i planiranje okruženja u kome će biti izvršeno testiranje koje zavisi od potreba i mogućnosti softvera koji se testira. Primer definisanja hardvera potrebnog za testiranja: CPU : Intel i6 i i7 Processor, HDD: 500GB, GPU: Integrisana kartica, RAM: 6GB+ , dok softver zavisi od tipa testiranja (da li se izvršava u fazi razvoja ili pre prihvatanja od strane korisnika).

ZADACI ZA SAMOSTALAN RAD

Dati su zadaci za definisanje procesa testiranja softvera.

- 1. Zadatak:** Napraviti plan testiranja softvera za proizvoljnu aplikaciju. Definisati faze testiranja, opisati svaku fazu i procese testiranja koji će se odvijati u svakoj fazi.
- 2. Zadatak:** Osmisliti verifikaciju i validaciju softvera proizvoljne aplikacije. Predložiti aktivnosti na osnovu tri faze testiranja (razvojno testiranje, testiranje softvera za isporuku, korisničko testiranje).

✓ Poglavlje 2

Testiranje u razvoju

VRSTE TESTIRANJA PRILIKOM RAZVOJA SOFTVERA

Razvojno testiranja uključuju sve aktivnosti testiranja koje provode članovi razvojnog tima.

Razvojno testiranje uključuju sve aktivnosti testiranja koje provode članovi razvojnog tima. Tester softvera je najčešće programer koji je razvio softver. Ponekad, tester i programer rade i u paru. Kod kritičnih sistema, koriste se formalni procesi koji odvajaju posebnu grupu koja se testiranjem, a u okviru razvojnog tima.

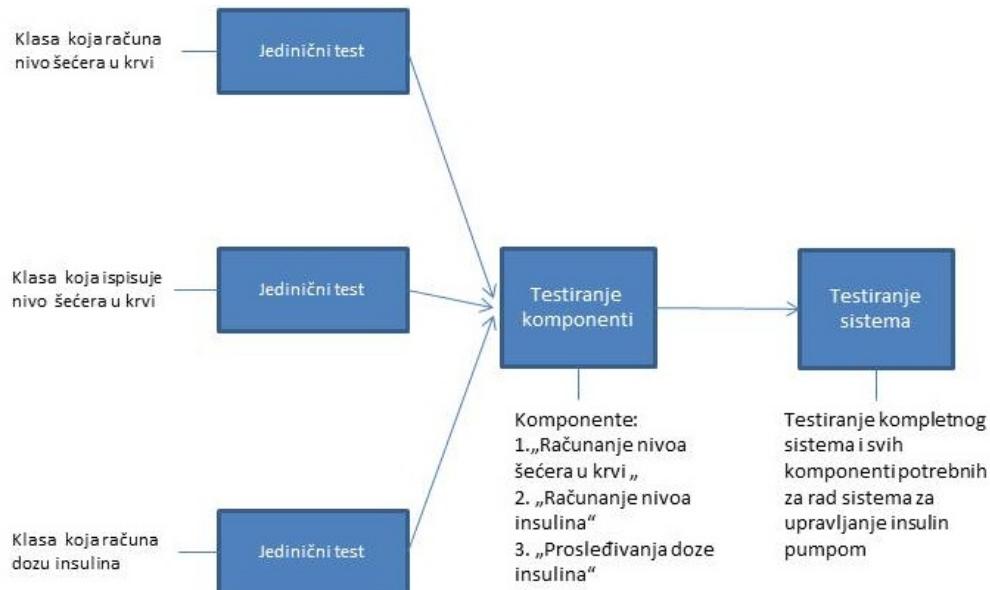
Za vreme razvoja, testovi se sprovode na tri nivoa granulacije:

1. **Jedinično testiranje**, kad a se testiraju jedinice programa ili klase objekata. Testovi jedinica testiraju funkcionalnost objekata ili metoda.
2. **Testiranje komponenata**, kada se po nekoliko jedinica se integrišu i čine komponente. Testiranje komponenti se koriste radi testiranja interfejsa.
3. **Testiranje sistema**, kada se neke ili sve komponente integrišu u sistem koji se onda u celini testira, Oni služe za testiranje interakcija komponenata.

Razvojno testiranje prvenstveno služi za otkrivanje grešaka u softveru. Zato se najšešče sprovodi paralelno sa procesom debagiranja, tj. procesa lociranja problema sa kodom, kada se vrše promene u softveru radi otklanjanja odvih problema.

PRIMER TESTIRANJA U RAZVOJU SISTEMA ZA UPRAVLJANJE INSULIN PUMPOM

Dat je primer testiranja u toku razvoja.



Slika 2.1 Primer testiranja u razvoju sistema za upravljanje insulin pumpom

Na slici 1 dat je primer testiranja u fazi razvoja. Jedinični test podrazumeva testiranje klase (u ovom slučaju klase koja računa nivo šećera u krvi, klase koja ispisuje nivo šećera u krvi i klase koja računa dozu insulina). Kada je testiranje klase kroz jedinične testove završeno, sledeći korak je testiranje komponenti u okviru koje se testiraju komponente: računanje nivoa šećera u krvi, računanje nivoa insulina i prosleđivanje doze insulina" koje sadrže prethodno testirane klase kroz jedinične testove. Na taj način eliminisane su greške u klasama. Kada je i testiranje komponenti uspešno završeno prelazi se na testiranje celokupnog sistema gde se zajedno testiraju sve komponente sistema.

ZADACI ZA SAMOSTALAN RAD

Dati su zadaci za testiranje softvera u toku razvoja.

1. Zadatak: Za sistem za merenje metereoloških podataka defisati jedinične testove, testiranje komponenti i testiranje celokupnog sistema. U jediničnim testovima opisati svaku klasu koja će biti testirana, u testiranju komponenti opisati koje komponente će biti testirane i koje klase sadrže.

2. Zadatak: Navesti prednosti testiranja u toku razvoja na osnovu prethodno urađenog zadatka. Da li jedinično testiranje olakšava proces testiranja komponenti?

▼ Poglavlje 3

Testiranje jedinice

TESTIRANJE OSNOVNIH JEDINICA SOFTVERA

Testiranje jedinice je proces testiranja osnovnih softverskih jedinica, kao što su metodi ili klase

Testiranje jedinice je proces testiranja osnovnih softverskih jedinica, kao što su metodi ili klase. Metod se testira tako što se poziva u izvršenje pri čemu mu se menjaju ulazni parametri.

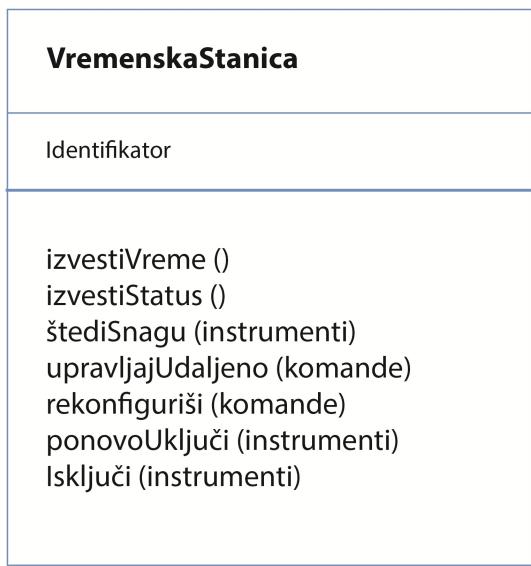
Kada se vrši testiranje klase, onda se testovi projektuju da pokriju sva svojstva klase, te sadrže:

- Testiranje svih operacija klase;
- Ubacivanje i proveru vrednosti svih atributa klase;
- Stavljanje objekta u sva moguća stanja. To se radi tako što se vrši simulacija svih događaja koja dovode do promene stanja.

Radi ilustracije, uzmimo primer stанице za prikupljanje podataka o vremenskim prilikama (ili kraće, vremenske stанице). Interfejs klase koji predstavlja stanicu je prikazan na slici 1.

Interfejs ima samo jedan atribut, a to je „identifikator“. To je konstanta čija se vrednost unosi kada se pri instalaciji stанице. Treba da se definišu slučajevi testiranja za sve metode navedene u interfejsu. Idealno, metode treba pojedinačno testirati. Međutim, često to nije mogućno jer oni zahtevaju određeno serijsko izvršenje, te se onda testiraju zajedno.

Testiranje klasa se komplikuje usled generalizacije i nasleđivanja klasa. Nije dovoljno testirati operaciju u klasi u kojoj se ona definiše, i očekivati da će u svim pod-klasama on raditi ispravno. Neophodno je testirati istu operaciju u svim pod-klasama klase u kojoj se definiše operacija.



Slika 3.1.1 Intrefejs klase stanice za praćenje vremenskog stanja

AUTOMATIZACIJA TESTIRANJA

Alati za testiranje (kao što je JUnit) poseduju opšte klase za testiranje koje vi možete da proširite da bi kreirali specifične slučajeve testiranja.

Uvek kada možete, automatizujte testiranje jedinica, primenom alata za testiranje, kao što je JUnit. Oni poseduje opšte klase za testiranje koje vi možete da proširite da bi kreirali specifične slučajeve testiranja. Na taj način možete da automatizovano izvršavate testiranja radne verzije vašeg metoda, da dobijate izveštaj sa testiranja i da koristiti odgovarajući GUI (engl. Graphic User Interface) za komunikaciju sa sistemom. Kako ti testovi obično traju samo nekoliko sekunda, možete ugraditi opciju da se oni automatski izvode uvek kada uvedete neku promenu u vaš metod (operaciju).

Automatizovani test ima tri dela:

1. Deo početnog podešavanja (initialization part), kada vršite inicijalizaciju sistema zajedno sa vašim slučajem testiranja (test case), tj. vršite unos podataka i očekivanih rezultata.
2. Deo pozivanja, kada pozivate metod ili objekt koji testirate.
3. Deo utvrđivanja rezultata, kada poredite rezultat rada pozvanog metoda ili objekta sa očekivanim rezultatom. Ako je dobijen pozitivan rezultat, onda je testiranje uspešno (nema greške); ako je pogrešno, onda test nije prošao.

Ponekad, objekt ili metod koji testirate, ima veze sa drugim objektima ili metodima, koji još nisu razvijeni (napisani) a a ko postoje, onda oni usporavaju proces testiranja. Na primer, ako vaš metod poziva bazu podataka, onda to obuhvata spori proces početnog uključivanja i podešavanje sistema baze podataka. U takvim slučajevima, možete koristiti lažne objekte. Lažni objekti su objekti sa istim interfejsom kao pravi spoljni objekti koje upotrebljavamo, i koji simuliraju njihovu funkcionalnost. Prema tome, lažni objekt može da simulira bazu podataka

i da ima samo nekoliko podataka koji su organizovani u nizu. Na taj način, njima s epristupa brzo, bez pozivanja prave baze podataka. Slično, lažni objekti se mogu da upotrebe i za simulaciju nenormalnih operacija ili vrlo retkih događaja.

IZBOR SLUČAJEVA ZA TESTIRANJE

Koriste se dve vrste slučajeva testiranja. Prvi se odnosi na normalan način rada programa, a drugi - na česte probleme zasnovane na iskustvu.

Zbog smanjivanje troškova testiranje, izbor slučajeva za testiranje jedinica se mora izvršiti tako da obezbedi efektivno testiranje, a to znači:

1. Slučajevi testiranja trebalo bi da pokažu da jedinica koja se testira, da radi ono što se od nje očekuje.
2. Ako ima grešaka u jedinici, test slučajevi bi trebalo da ih otkriju.

Vi bi trebalo da pišete dve vrste slučajeva testiranja. Prvi slučaj bi trebalo da se odnosi na normalni način rada programa, i trebalo bi da pokaže da komponenta radi. Drugi slučaj testiranja bi trebalo da se odnosi na iskustvo vezano za česte probleme. Koriste se nenormalni ulazi, da bi se proverilo da se oni obrađuju na propisan način i da ne dovode do ispada komponente iz rada jedinice.

Moguće se dve strategije koje su efektivne za izbor slučajeva za testiranje:

1. *Testiranje particija*, kada utvrđujete grupe ulaza koje imaju zajednička svojstva i trebalo bi da se obrađuju na isti način. Za svaku od ovih grupa treba planirati posebne testove.
2. *Testiranje zasnovano na smernicama*, kada se slučajevi za testiranje utvrđuju na osnovu smernica, koje su rezultat prethodnog iskustva sa greškama koje programeri često prave kada razvijaju

ULAZNI PODACI I IZLAZNI REZULTATI PROGRAMA

Kategorije ulaznih podataka se često nazivaju particijama ekvivalencije ili domenima. Testiranje particija se koristi kod slučajeva testiranja sistema.

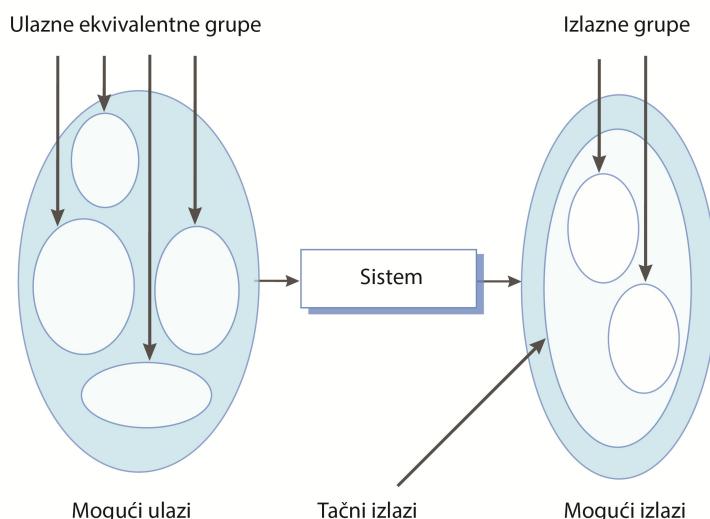
Ulagani podaci i izlazni rezultati programa, često se mogu podeliti na određeni broj kategorija sa zajedničkim karakteristikama. Na primer, kategorije pozitivnih brojeva i negativnih brojeva. To znači, ako testirate program koji vrši proračun i zahteva dva pozitivna broja, onda očekujete da se program na isti način i pri unosu bilo kojih pozitivnih brojeva.

Ove kategorije ulaznih podataka se često nazivaju particijama ekvivalencije ili domenima. Ako se vrše sistematsko testiranje sistema ili komponenata, onda se testiranje vrši korišćenjem svih ulaznih i izlaznih particija. Slučajevi testiranja se kreiraju tako da se ulazi i izlazi

nalaze unutar ovih particija. Testiranje particija se koristi kod slučajeva testiranja sistema i komponenata (slika 2)..

Male elipse na slici 2 prikazuju ulazne i izlazne particije. Sprovode se svi testovi za sve ulazne particije, tj. za sve njihove članove (podatke). Izlazne particije su particije u okviru kojih se dobijaju rezultati koji dele neke zajedničke karakteristike. Tamno zatamljene elipse, na obe strane, pokazuju pogrešne ulaze, odn. izuzetke koje su se javili zbog pogrešnih ulaznih podataka.

Kada utvrdite particije skup particija, onda birate slučajeve testiranja za svaku od ovih particija. Preporuka je da to budu slučajevi koji se nalaze na granicama particija, kao i za srednje tačke particija. Greške se javljaju najčešće kada se koriste netipični podaci

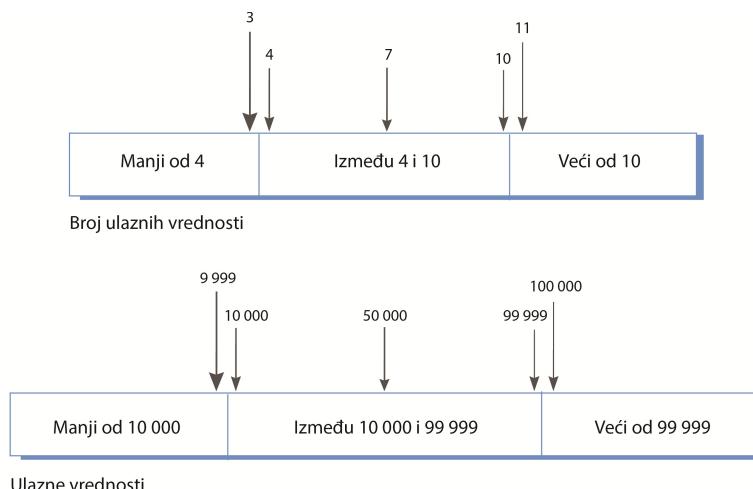


Slika 3.1.2 Ulagne i izlagne particije

PRIMENA PARTICIJA EKVIVALENCIJE

Primena particija ekvivalencije je efektivan pristup testiranja jer pomaže u utvrđivanju grešaka programera kada se za ulaze koriste podaci na ivicama particija.

Izbor slučajeva za testiranje pokazaćemo na sledećem primeru. Specifikacija programa definiše da program prihvata 4 do 8 ulaza koji predstavljanju cele brojeve sa pt brojeva, koji su veći od 10.000. Na osnovu ovoga, mi određujemo ulazne particije i moguće ulazne vrednosti za testiranje (slika 3).



Slika 3.1.3 Primer korišćenja participacija ekvivalencije

Ovde se koristi „testiranje crne kutije“. To znači da ne morate da znate sistem radi. Poželjno je i dodatno testiranje sa „testiranjem bele kutije“, kada gledate kod programa da bi odredili druge

moguće testove. Na primer, vaš kod može da sadrži izuzetke radi reakcije na netačne ulaze. Na osnovu uvida u ovo, utvrđujete „particije izuzetaka“ koje čine različiti opsezi ulaznih podataka za koje se očekuje isti izuzetak u programu.

Primena particija ekvivalencije je efektivan pristup testiranja jer pomaže u utvrđivanju grešaka programera kada se za ulaze koriste podaci na ivicama particija.

SMERNICE ZA IZBOR SLUČAJEVA ZA TESTIRANJE

Smernice su urađene na osnovu znanja o vrstama slučajeva testiranja koje su efektivne u otkrivanju grešaka

Pri izboru slučajeva za testiranja, mogu se koristiti i odgovarajuće smernice. One su urađene na osnovu znanja o vrstama slučajeva testiranja koje su efektivne u otkrivanju grešaka. Na primer, kada testirate programe sa sekvencama, nizovima ili listama, smernice obuhvataju:

1. Testiranje softvera sa sekvencama koje imaju samo jednu vrednost. Programeri kreiraju sekvene sa nekoliko vrednosti. Ako pri tome koriste sekvene sa istom vrednošću, program može da radi nepravilno.
2. Upotrebite različite sekvencije sa različitim veličinama u različitim testovima.
3. Radi testove tako, da se koriste prvi, srednji i poslednji elementi u sekvenci.

U knjizi: Whittaker, J. W. (2002). How to Break Software: A Practical Guide to Testing. Boston: Addison-

Wesley. daju se se uopštene smernice, ka na primer:

- Birajte ulaze koje dovode do toga da sistem proizvodi sve poruke o greškama;
- Planirajte ulaze kaoji dovode do prekoračenja prostora za njihov smeštaj;

- Ponovite isti ulaz ili serije ulaza više puta;
- Dovodite to dobijanja pogrešnih rezultata.
- Dovedite do toga da rezultati budu ili isuviše veliki ili isuviše mali,

PRIMER TESTIRANJE STANJA VREMENSKE STANICE

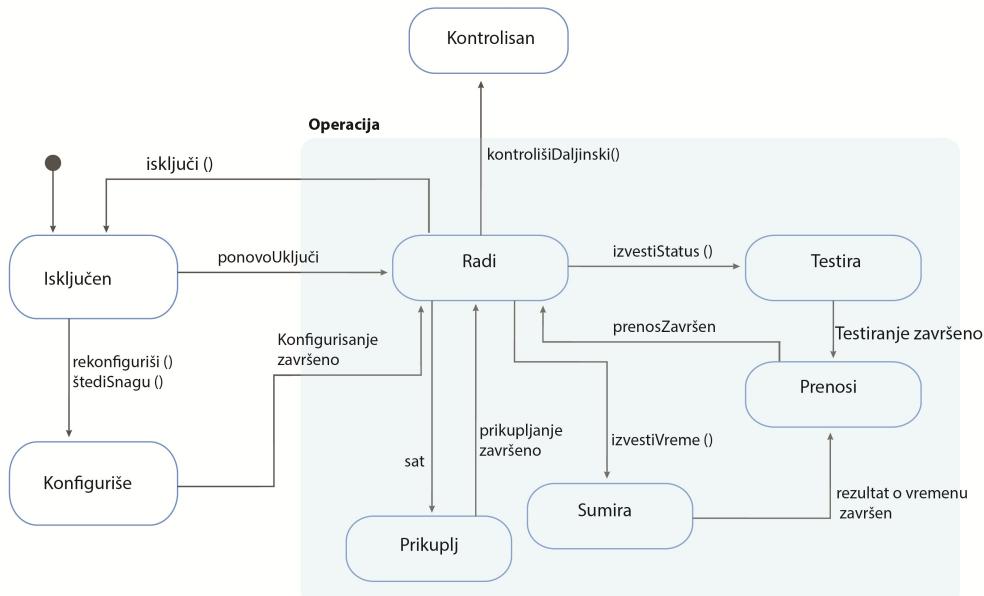
Modelom se može definisati redosled promene stanja objekta koja se moraju testirati, kao i redosled dođađaja koji dovode do promene ovih stanja.

Za testiranje stanja vremenske stanice, koristi se njen dijagram stanja (slika 4). Upotrebom ovog modela, možete da utvrdite redosled promena stanja koja se moraju testirati i da definišete redosled događaja koje izazivaju javljanje ovih stanja. U principu, trebalo bi da testirate svaki mogući redosled tranzicija (promena) stanja, iako to može biti dosta skupo.

Evo nekih primera redosleda stanja koja bi trebalo da se testiraju:

1. Isključenje → Rad → Isključenje
2. Konfigurisanje → Rad → Testiranje → Prenos → Rad
3. Rad → Prikupljanje → Rad → ...

... → Sumiranje → Prenos → Rad



Slika 3.1.4 Dijagram stanja stanice za prikupljanje podataka o vremenskim prilikama

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci za testiranje jedinica.

1. Zadatak: Za proizvoljnu aplikaciju definisati promenu stanja (na osnovu modelovanog dijagrama stanja) a zatim testirati svaku moguću promenu stanja unutar aplikacije. Napraviti redosled stanja i objasniti svako stanje u kome se može nalaziti aplikacija.

2. Zadatak: Navesti prednosti automatizacije testiranja i konkretan primer gde je moguće izvršiti primenu automatizacije testiranja.

3.1 JUnit

JEDINIČNO TESTIRANJE

Šta je testiranje softvera

Kada smo govorili o klasama i objektima, rekli smo da klasu pišemo uvek tako da ona bude proizvod za sebe, da možemo da tu klasu iskoristimo u bilo kom drugom proizvodu (projektu), i da bez ikakvih problema možemo da je implementiramo i primenimo, da klasa bude jedan jedinstveni proizvod. Ova osobina Objektno orijentisanog programiranja je izuzetno bitna za jedinično testiranje. Jedinično testiranje je testiranje pojedinačnih objekata. Dakle, imamo test koji testira odgovarajuću klasu, odnosno metode koje nešto rade u okviru te klase. Najbolji alat za testiranje pojedinačnih klasa je **JUnit**. To je **Framework**, skup alata pomoću kojih možemo da vršimo testiranje. Naravno, testove moramo sami da napišemo. **JUnit** se isporučuje kao Jar biblioteka, sa svim potrebnim alatima za jedinično testiranje, mi tu biblioteku uvrstavamo u naš projekat, i pozivamo odgovarajuće klase iz te biblioteke da bi vršili testiranje. Jedinično testiranje možemo da izvršimo i bez **JUnit Framework**-a tako što bismo napisali novu klasu, koja bi pozivala instance klase koju testiramo, i proveravala da li ispravno radi na određenim test primerima.

- Objekat je proizvod za sebe
- Test testira pojedinačni objekat
- JUnit je Framework
- Jar biblioteka sa svim potrebnim alatima za jedinično testiranje

PREDNOSTI TESTIRANJA SOFTVERA

Značaj jediničnog testiranja

Savremeni projekti obično insistiraju na ovim jediničnim testovima. Jedinični testovi značajno povećavaju kvalitet koda, zato što je svaka pojedinačna klasa iztestirana. Ako imamo testove za svaku pojedinačnu klasu lakše je čitanje koda, ako nešto ne razumemo u kodu, možemo da pogledamo testove i da vidimo šta taj deo koda radi. Ako imamo testove, onda možemo bez problema da radimo naknadne **refactoring**-e, izmene koda, a da se ne bojimo da ćemo narušiti neku funkciju na odnos koji postoji. Nakon izvršenog **refactoring**-a pustimo ponovo test i, ako je sve u redu, znači da nismo narušili prvočitnu funkciju na odnos programa. Testovi

se koriste i za pravljenje testova prihvaćanja. Dakle, budućem korisniku pokazujemo da naša aplikacija zaista radi ono što on očekuje od nje. Dakle, testovi mogu da pokažu da određene metode izračunavaju tačno ono što je korisnik tražio. Jedinični testovi se odlično uklapaju u procedure izrade softwera-a, u odgovarajući Project Plan, u odgovarajuću kontrolu kvaliteta koda. Ako svaka klasa koju napišemo ima odgovarajuće **Junit** testove, to sve povećava pouzdanost celog sistema, možemo sa većom pouzdanošću reći da system radi ono što treba da radi. Samim tim je smanjena i količina **bug-ova** koji mogu da se pojave u kodu. **JUnit** testovi, za razliku od drugih tipova testova, proveravaju sve metode koje se nalaze u okviru jedne klase koja nešto konkretno radi, bez obzira kolika je verovatnoća pokretanja te metode u realnom radu. Problem sa drugim oblicima testiranja je u tome što se neke situacije nikada ne dese prilikom testa, već tek kod klijenta. **JUnit** testovi proveravaju svaki deo koda, bez obzira na njegovu verovatnoću pojavljivanja u kodu.

- Veći kvalitet koda
- Lakše čitanje koda
- Moguće raditi naknadne refaktoringe
- Mogućnost pravljenja testova prihvaćanja
- Lako se uklapa u procedure izrade softvera
- Povećava pouzdanost celog sistema
- Smanjuje količinu grešaka u kodu

MANE TESTIRANJA SOFTVERA

Mane jediničnog testiranja

Pored velikih prednosti, jedinično testiranje ima i izvesnih mana. Ove mane su obično vezane za kod koji je već ranije napravljen po neobjektno orijentisanim principima, pa je nemoguće implementirati na njega jedinično testiranje koje je specijalno optimizovano i prilagođeno za objektno orijentisano programiranje. Pored toga, dosta je teško primeniti jedinično testiranje na GUI klasi. GUI klasa obično interaguje sa korisnikom pa je teško napisati **JUnit** testove koji bi izvršili takva testiranja. Testiraju se samo klase, a ne i okruženje u kome se radi. Npr. preko **JUnit** testiranja se testira kod Javin, ali ne i SQL naredbe baze. Takođe, neke situacije u mreži ne mogu da se istestiraju, jer se odnosi na konkretni kod u aplikaciji. Pomoću ove vrste testiranja ne možemo da istestiramo neku situaciju koja može da se desi u okruženju (nasilan nestanak struje ili sl.). Kada imamo **JUnit** testiranje, susrećemo se sa sporijim razvojnim ciklusom. Naime, prilikom pisanja testova, mi moramo u prosek da napišemo oko tri puta više koda u okviru testa, nego što pišemo u samom kodu koji treba da se izvršava. Ovo, naravno, zbog količine posla koji treba uraditi, dosta usporava razvojni ciklus projekta.

- Ne može da se primeni na GUI klase
- Testira se samo klase ne i okruženje u kom kod radi (baze podataka, mreža, ...)
- Kod mora da bude rađen po strogim Objektno Orijentisanim principima
- Sporiji razvojni ciklus
- do 3x više koda u testovima nego u samom projektu

NAPOMENE PRILIKOM IZRADE TESTOVA

Korisni saveti i napomene prilikom izrade JUnit testova

Prilikom jediničnog testiranja najviše se obraća pažnja na metode koje nešto rade. Metode koje nešta rade obično implementiraju nekakav algoritam, su metode koje određuju funkcionalnost naše aplikacije, tako da je to upravo mesto koje trebamo najviše da testiramo. Pored metoda, možemo da testiramo i konstruktore, ako se u okviru konstruktora generiše nekakav funkcionalni kod. Obično se ne testiraju **get**-eri i **set**-eri, zato što **get**-eri i **set**-eri samo postavljaju, odnosno prosleđuju vrednost iz polja klase, tako da oni nisu zanimljivi za detaljno testiranje (oni skoro uvek rade ono što treba da rade). **Set**-ere često i automatski generišemo, tako da oni nisu interesantni za **JUnit** testiranje. Ne testiraju se metode koje služe samo za adaptiranje, odnosno prosleđivanje naredbe drugim metodama. Takve metode takođenisu interesantne za testiranje.

- metode koje nešta rade
 - mogu se testirati konstruktori
1. ne testira se geteri i seteri
 2. ne testiraju se metode koje samo pozivaju druge metode

GREŠKE

Postupak pronalaženja grešaka u JUnit testovima

Pogrešno je misliti da, ako imamo **JUnit** testove, da je nemoguće da se pojavi **bug**. Naravno da mi testovima ne možemo da predvidimo sve situacije, i da se ponekad **bug** ipak podkrade i pojavi u aplikaciji. U tom slučaju trebamo da poštujemo sledeći postupak za otkrivanje **bug**-a u aplikacijama koje imaju **JUnit** testove: Prvo i osnovno što treba da znamo je da ne smemo da diramo kod. Prvo što radimo je pronalaženje testa koji pokazuje da ta greška postoji, i taj test onda pišemo. Takav test treba da padne, treba da pokaže da je **failed**, odnosno da projekat ne daje zadovoljavajući rezultat. Kada smo definisali **bug**, odnosno tačno ga locirali, onda menjamo kod tako da ispunji zahtev testa, odnosno da prođe. Pokrene se ponovo test, i ustanovi se da greške više nema. Na taj način smo ispravili grešku u aplikaciji. Prilikom pokretanja testova obično pokrećemo sve testove u aplikaciji. Ovo je jako dobro prilikom otklanjanja **bug**-ova, jer smo sigurni da otklanjanje ove greške nije prouzrokovalo druge greške. U slučaju da nemamo **Junit** testove, i da pokušavamo da ispravimo **bug**-ove, to radimo obično tako što prepostavimo da je na nekom mestu **bug**, pa probamo da ispravimo, onda pokrenemo aplikaciju, pa shvatimo da nije bio tu **bug**, pa onda izmenimo neko drugo mesto, pa neko treće mesto... Tim menjanjima smo možda u potpunosti uništili neke stvari koje su radile kako treba. Ništa od ovoga mi ne vidimo u tom trenutku, već dajemo takav kod klijentu, klijent to startuje i ustanavljava da sada ima više grešaka nego što je imao prvi put. Ako koristimo **Junit** testove, i postupak za pronalaženje **bug**-ova koji je definisan na ovom primeru, takva situacija se nikada neće desiti.

1. Kada se pojavi BUG u aplikaciji postupak je sledeći.

2. kod se ne dira
3. napiše se test koji pokazuje da greška postoji (test padne)
4. promeni se kod da ispunji zahtev testa
5. pokrenuti se testovi i ustanovi da greške nema.
6. prilikom pokretanja pokreću se svi testovi kako bi ustanovili da ispravka bug-a nije nešto poremetila

ŠTA TEST TREBA DA PROVERI

Jediničan test treba da proveri da li je rezultat tačan za realne vrednosti koje će se dešavati u aplikaciji

Zatim treba da proveri kako se ponaša kod za potencijalno problematične vrednosti (npr. ako mu pošaljemo nulu sa kojom nešto deli). Takođe treba da vidimo kako se naš kod ponaša na graničnim vrednostima. Interesantno je videti i kako se kod ponaša na vrednostima koje su sigurno pogrešne, šta se dešava kada je vrednost u pogrešnom formatu, šta se dešava kada fali podatak. Test treba da proveri da li je vraćeni rezultat u pravilnom formatu, da li pripada opsegu kojem treba. Ovo je sve jako bitno prilikom pisanja testova jer, ako predvidimo sve varijante u okviru testa i znamo tačno kako se aplikacija ponaša za svaku od ovih varijanti, onda tačno znamo šta ta klasa može i kako će se ponašati, i nećemo doći u situaciju da joj prosleđujemo podatke ili da je stavljamo u situacije koje ona ne ume da razreši.

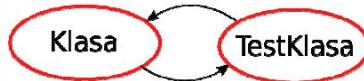
- Da li je rezultat tačan za realne vrednosti
- Potencijalno problematične vrednosti (np. deljenje sa nulom)
- Granične vrednosti
- Vrednosti koje su sigurno pogrešne
- Vrednosti u pogrešnom formatu
- Kad fali podatak
- Provera formata rezultata
- Opsege

DELOVI JUNIT TESTA

Odnos klase i Test klase, prikaz iz kojih delova se sastoji Junit test kao i njegov životni ciklus (Life Cycle)

Rekli smo da su **JUnit** testovi testovi nad klasama, odnosno nad pojedinačnim klasama. To znači da uvek imamo par: Klasa koju nešto radi, i njena odgovarajuća TestKlasa koja testira ispravnost rada ove klase. Oni uvek idu u paru, i, kada govorimo o **JUnit** testovima, uvek govorimo o paru klase i klase koja je obrađuje, odnosno testira. Ne može da se desi da jedna

ista TestKlase testira više klasa konkretnih, ili da jedna konkretna klasa bude testirana od strane nekoliko TestKlase. Uvek je to odnos jedan na jedan, klasa i odgovarajuća TestKlase.



Slika 3.2.1 Odnos klase i test klase

Osnovne klase koje se koriste u okviru **JUnit** testova su:

- **TestCase**, koji predstavlja pojedinačni slučaj testiranja (dakle, naša klasa nasleđuje klasu **TestCase**, i pravi pojedinačni slučaj testiranja)
- **TestSuite**, je skup klasa koje se testiraju (imamo klasu koja nasleđuje **TestSuite**, i ona u sebi može da sadrži više **TestCase**-ova),
- **TestRunner (BaseTestRunner)**, je klasa koja obezbeđuje funkcionalnost, odnosno pokretanje testova, i
- **TestResults**, je klasa u koju se smeštaju rezultati testa.

S' obzirom da je **JUnit** testiranje manje – više automatizovano, nama će, u većini slučajeva, trebati smo **TestCase** i **TestSuite** klase, odnosno njih ćemo nasleđivati da bi napravili svoje **JUnit** testove.

Postoje tri metode u okviru **TestCase**-a, koje mi nasleđujemo i implementiramo u okviru svoje TestKlase. To su metode:

- **setUp()** , koja nam služi da u nju postavimo osnovne postavke (da se konektujemo na bazu, ili da se povežemo na mrežu, ili da učitamo neki dokument sa diska),
- **testXXX()** , metode koje testiraju odgovarajuće metode iz konkretne klase (njih može da bude više, i svaka **test** metoda se odnosi na neku metodu u konkretnoj klasi),
- **tearDown()** , je metoda koja se pokreće nakon završetka testova (u nju možemo da zatvorimo vezu ka bazi, ili vezu ka drugom serveru, ili bilo šta što je potrebno uraditi nakon izvršenih testova).

JUNIT ASSERT

Assert metode služe da se proveri ispravnost neke vrednosti

U slučaju **assertEquals** metode proveravamo da li neki broj odgovara očekivanom broju, sa određenom tolerancijom. Prvi parametar je **message**, odnosno poruka koju možemo da pošaljemo u slučaju da taj test ne prođe. **AssertTrue** je test pomoću koga proveravamo da li je vrednost tačna. Dakle, imamo nekakav **expression**, izraz, i proveravamo da li taj izraz vraća **true**. Izraz može da bude promenljiva, a može da bude poziv metode, opcionalno možemo da pošaljemo i poruku. **AssertFalse** je kontra od **AssertTrue**-a, dskle proveravamo da li nam vraća **false**, da li nam vraća netačno.

- `assertEquals([String message], expected, current, tolerance);`

- assertTrue([String message], boolean);
- assertFalse([String message], boolean expression);

Pored toga što možemo da proveravamo vrednosti prostih tipova, moramo da imamo mehanizam i za proveru objekata. To su metode **assertNull** i **assertNotNull**, da bi videli da li objekat postoji ili ne postoji. Prosleđujemo kao parametar, naravno, objekat za koji se pitamo da li postoji ili ne postoji, i opcionalno poruku koja će se pojaviti u slučaju da kod test padne. Onda imamo metode **assertSame** i **assertNotSame**, odnosno da li su dva objekta isti objekat ili ta dva objekta nisu isti objekat. Takođe, pored ta dva objekta prosleđujemo i onaj opcionalni **message**, odnosno tekst. Ovde treba obratiti pažnju na to da postoji i vrednost i kontravrednost, dakle **assertNull** i **assertNotNull**, **assertSame** i **assertNotSame**.

Kada testiramo, mi se ponekad pitamo da li će naš program vratiti **null object** zato što nije umeo da razreši neku situaciju, i to postavljamo kao odgovarajući test **assertNull**, dakle da li će da vrati **null**. Iako **null** obično predstavlja grešku u programu, mi mu često dajemo pogrešne vrednosti prilikom testiranja, i od njega očekujemo takve rezultate.

- 1.assertNull([String message], Object object);
- 2.assertNotNull([String message], Object object);
- 3.assertSame([String message], Object o1, Object o2);
- 4.assertNotSame([String message], Object o1, Object o2);

PRIMERI JUNIT TESTOVA

Primer testiranja stanja naloga kao i datuma

Ovde se vidi primer jedne test klase (koja se u ovom slučaju zove AccountTest) i odgovarajućeg metoda testGetBalanceOk, u kome se proverava stanje te metode u odgovarajućoj originalnoj klasi. Iz samog naziva test klase vidimo da se originalna klasa zove Account, a da se metoda koju testiramo zove GetBalanceOk.

```
import junit.framework.TestCase;
public class AccountTest extends TestCase{
    public void testGetBalanceOk () {
        long balance = 1000;
        Account account = new Account(balance);
        long result = account.getBalance();
        assertEquals(balance, result);
    }
}
```

Datumi su izuzetno nezgodni za testiranje, zbog toga što se oni pomeraju, dakle u trenutku kada smo pravili test bio je jedan datum, u trenutku kada pokrećemo test može da bude drugi datum, tako da ne smemo nigde u okviru testa da se referenciramo na današnji datum, uvek moramo da pravimo nekakve absolutne datume i da pomoću njih proveravamo. Problem je još komplikovaniji ako se sam kod koji testiramo bazira na trenutnom datumu. Onda u okviru test klase moramo da pravimo pomeranje u odnosu na trenutni datum, vodeći računa da ta

pomeranja odgovaraju željenim testovima. U ovom primeru se vidi kako se prave testovi kada su datumi u pitanju. Datumi mogu da budu najkomplikovaniji delovi testa.

```
public class MyTaskTest extends TestCase
{
    SimpleDateFormat df = new SimpleDateFormat("dd/MM/yyyy");
    private MyTask ob1;
    Calendar cal0b2;
    private MyTask ob2;
    Calendar cal0b3;
    private MyTask ob3;
    public static Test suite()
    {
        return new TestSuite(MyTaskTest.class);
    }
    public MyTaskTest(String name)
    {
        super(name);
    }
    protected void setUp()
    {
        ob1 = new MyTask();
        ob2 = new MyTask();
        ob3 = new MyTask();
        cal0b2 = Calendar.getInstance();
        setCalendarWithoutTime(cal0b2);
        cal0b2.roll(Calendar.DAY_OF_MONTH, 2);
        cal0b3 = Calendar.getInstance();
        setCalendarWithoutTime (cal0b3);
        cal0b3.roll(Calendar.DAY_OF_YEAR, -41);
        ob2.setDone(new Boolean(true));
        ob2.setMyTask ("Task Test");
        ob2.setContact("331984");
        ob2.setFinalDate(df.format(cal0b2.getTime()));
        ob2.setPriority("Height");
        ob3.setField(0, new Boolean(false));
        ob3.setField(1, "Other Task");
        ob3.setField(2, "43904");
        ob3.setField(3, df.format(cal0b3.getTime()));
        ob3.setField(4, "Low");
    }
    private void setCalendarWithoutTime(Calendar c)
    {
        c.set(Calendar.HOUR, 0);
        c.set(Calendar.MINUTE, 0);
        c.set(Calendar.SECOND, 0);
        c.set(Calendar.MILLISECOND, 0);
    }
    public void testGetField()
    {
        assertEquals(new Boolean(false), ob1.getField(0));
        assertEquals(new String(), ob1.getField(1));
        assertEquals(new String(), ob1.getField(2));
    }
}
```

```
assertEquals(new String(), ob1.getField(3));
assertEquals(new String(), ob1.getField(8));
assertEquals(new Boolean(true), ob2.getField(0));
assertEquals("Task Test", ob2.getField(1));
assertEquals("331984", ob2.getField(2));
assertEquals(df.format(cal0b2.getTime()), ob2.getField(3));
assertEquals("Height", ob2.getField(8));
assertEquals(new Boolean(false), ob3.getField(0));
assertEquals("Other Task", ob3.getField(1));
assertEquals("43904", ob3.getField(2));
assertEquals(df.format(cal0b3.getTime()), ob3.getField(3));
assertEquals("Low", ob3.getField(8));
}
public void testNoDayFinalDate()
{
long diff0b2 = cal0b2.getTimeInMillis() - Calendar.getInstance().getTimeInMillis();
diff0b2 /= 86400000;
assertEquals(diff0b2, ob2.noDayFinalDate());
long diff0b3 = cal0b3.getTimeInMillis() - Calendar.getInstance().getTimeInMillis();
diff0b3 /= 86400000;
assertEquals(diff0b3, ob3.noDayFinalDate());
}
```

TESTED CLASS

Prikazana je klasa MyTask koja je testirana prethodnim testovima

Konstruktor klase MyTask prosleđuje napunjene vrednosti u druge konstruktore iste ove klase. To znači da ova klasa nikada ne može da se pojavi bez elemenata – oni su ili podešeni na prazne **Stringove**, ili se napune kroz drugi konstruktor.

```
public class MyTask extends Object implements Serializable
{
private Boolean done;
protected String obaveza;
protected String kontakt;
protected String rok;
protected String prioritet;
private Calendar danas;
protected Calendar uradjeno;
public MyTask()
{
this(new Boolean(false), new String(), new String(), new String(), new String());
}
private MyTask(Boolean done, String obaveza, String kontakt, String rok, String prioritet) {
super();
this.done = done;
this.obaveza = obaveza;
this.kontakt = kontakt;
```

```
this.rok = rok;
this.prioritet = prioritet;
this.danas = Calendar.getInstance();
this.uradjeno = null; }
public String getPrioritet()
{
return prioritet;
}
public void setPrioritet(String prioritet)
{
this.prioritet = prioritet;
}
.....
public Object getField(int i)
{
if (i == 0) return getDone();
else if (i == 1) return getObaveza();
else if (i == 2) return getKontakt();
else if (i == 3) return getRok();
else return getPrioritet();
}
public void setField(int i, Object o)
{
if (0 == i) setDone((Boolean) o);
else if (i == 1) setObaveza((String) o);
else if (i == 2) setKontakt((String) o);
else if (i == 3) setRok((String) o);
else setPrioritet((String) o);
}
public long noDayFinalDate() {
long res;
if (rok != null || rok != "") {
res = RokOdStringa().getTimeInMillis() - Calendar.getInstance().getTimeInMillis();
res /= 86400000;
return res;
}
return 0;
}
```

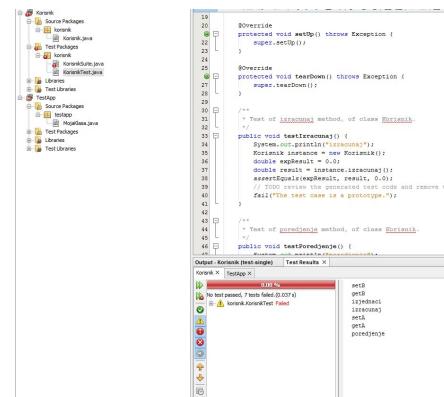
RAZVOJNO OKRUŽENJE & JUNIT

Upotreba NetBeans razvojnog alata za kreiranje Junit testova

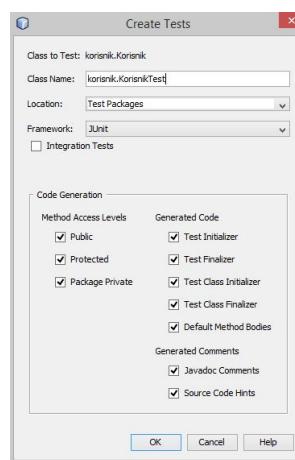
Razvojna okruženja ponekad uvode razne olakšice, da možemo jednostavnije da napravimo **Junit** testove. U je obično ugrađen **JUnit**, tako da ne moramo posebno da ga dodajemo i da ga posebno pozivamo, već to radi razvojno okruženje umesto nas. **NetBeans**-a idu čak korak dalje, da nam ponude da umesto nas kreiraju metode, sve sa početnim nultim vrednostima. Takođe nam daju mogućnost da biramo da li hoćemo da kreirami i **setUp** i **tearDown** metodu, kao i opcionalno da li želimo da se generiše Java DOC komentari. Naš zadatak je samo da upišemo odgovarajući naziv test klase.

Kada prvi put kroz **NetBeans**-ov **Wizard** napravimo testove, nakon svakog testa postoji metoda **Failed** koja se izvršava uvek, odnosno svaki test izvršava kao da je pao. To je urađeno namerno, da se mi ne bismo oslonili na te osnovno automatske generisane testove, već da bismo ih izmenili i prilagodili konkretnim situacijama, da na bi došlo do slučajnog uspešnog testa. Kada pokrenemo testove, a da ništa nismo menjali, svi testovi će da se prikažu sa **Failed**, da su pali. Mi možemo to da izmenimo tako što pravimo konkretne, prave testove koje naša aplikacija treba da radi, i nakon toga možemo da pustimo izvršavanje testova. Može da nam se desi da u okviru **testSuit**-a neki testovi prođu, a neki testovi padnu. To se vidi na ovom **ScreenShot**-u **NetBeans** a, gde se vidi da je ukupan test **Failed**, nije prošao, iako su neki od konkretnih testova prošli. Da bi celokupan test prošao, moraju svi pojedinačni testovi da prođu.

U trenutku kada uspešno propustimo test, odnosno kada su svi pojedinačni testovi koji postoje u okviru tog paketa uspešno prošli, onda je i celokupan test prošao. U donjem delu **NetBeans**-a se nalazi statistika za **JUnit** klasu **testResults**, i tu vidimo da su svi pojedinačni testovi prošli, pa je i ukupna ocena kompletног testa **Passed**, prošao.



Slika 3.2.2 Prikaz lošeg testa



Slika 3.2.3 Kreiranje JUnit

Datumi su izuzetno nezgodni za testiranje, zbog toga što se oni pomeraju, dakle u trenutku kada smo pravili test bio je jedan datum, u trenutku kada pokrećemo test može da bude drugi datum, tako da ne smemo nigde u okviru testa da se referenciramo na današnji datum, uvek moramo da pravimo nekakve absolutne datume i da pomoću njih proveravamo. Problem je

još komplikovaniji ako se sam kod koji testiramo bazira na trenutnom datumu. Onda u okviru test klase moramo da pravimo pomeranje u odnosu na trenutni datum, vodeći računa da ta pomeranja odgovaraju željenim testovima. U ovom primeru se vidi kako se prave testovi kada su datumi u pitanju. Datumi mogu da budu najkomplikovaniji delovi testa.

ZADACI ZA SAMOSTALNI RAD

Zadaci za samostalan rad korišćenjem NetBeans razvojnog okruženja.

1. Zadatak: Napraviti klasu koja predstavlja rezultat ispita. U okviru klase imamo niz od 6 celobrojnih cifara u intervalu od 0 do 20 koje predstavljaju ocene na pojedinačnim zadacima na ispitu. U klasi treba da imamo i dve metode brojLispitnihPoena() i brojPoena(). Metoda brojLispitnihPoena() treba da vrati zbir 5 najboljih zadataka dok metoda brojPoena() treba da vrati isti podatak pomnožen sa brojem 0.4 Za ovu klasu napraviti JUnit testove po pravilima datim na času. Probati da se obuhvate sve varijante opisane na vežbama (nema vrednost, negativna vrednost, granična vrednost...)

2. Zadatak: Napraviti program kalkulator. Kalkulator treba da ima ulaz na konzolnoj liniji dva broja i operaciju koju treba da izvrši (sabiranje, oduzimanje, množenje, deljenje). Napraviti odgovarajuće Junit testove koji će demonstrirati ispravnost rada kalkulatora.

Napomena: Kalkulator ne sme deliti sa nulom.

3.2 Selenium framework

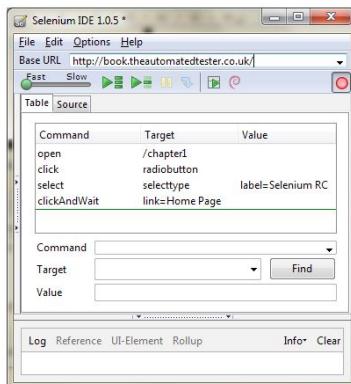
SELENIUM IDE

Kreiranje testa upotrebom Selenium framework-a

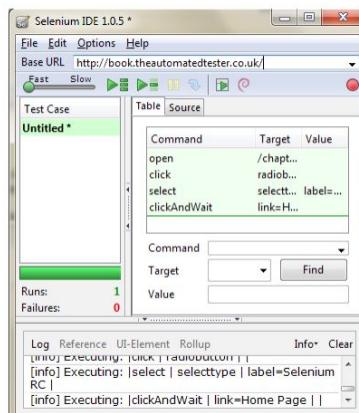
Pokazaćemo kako se snima test korišćenjem Selenium IDE. Da bi pokrenuli snimanje testova, moramo pokrenuti web browser Mozilla Firefox. Nakon toga pokrenućemo Selenium IDE, dodatak za Firefox, koji treba biti instaliran. Njega možemo pronaći u Tools meniju. Kada se prvi put pokrene Selenium, snimanje testa je već uključeno. Da bismo pokrenuli snimanje testova, potrebno je uraditi sledeće:

1. Promeniti Base URL na glavnu URL adresu aplikacije koju želimo da testiramo. U ovoj vežbi, mi ćemo koristiti stranicu <http://book.theautomatedtester.co.uk/>.
2. Kliknuti na Chapter1.
3. Kliknuti na radio dugme.
4. Promeniti vrednost padajuće liste Select na Selenium RC.
5. Kliknuti na link HomePage.
6. Naš test je sada snimljen i treba da izgleda kao na slici 1 . Kliknuti na PLAY dugme.
7. Kada se jedan test izvrši, on izgleda slično kao na slici 2 .

Uspešno smo snimili naš test i pokrenuli ga ponovo. Kao što možemo primetiti, Selenium IDE pokušava da primeni prvo pravilo automatskog testiranja tako što koristi OPEN komandu (u ovom slučaju biramo Chapter1), a zatim korak po korak snima izvršavanje komandi koje želimo da testiramo. Kada se sve akcije izvrše, videćemo da sve akcije imaju zelenu pozadinu. To označava da su sve izvršene uspešno.



Slika 3.3.1 izgled test primera nakon snimanja



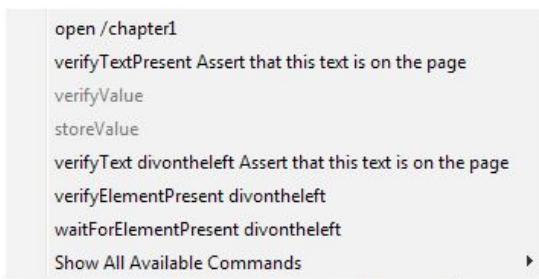
Slika 3.3.2 Primer nakon pokretanja

AŽURIRANJE TESTA

Ažuriranje testa za proveru elemenata na stranici

Koristićemo istu stranicu kao i ranije, ali čemo se uveriti da su različiti elementi na toj stranici. Postoje dva mehanizma validacije elemenata, koji su dostupni na aplikaciji koja se testira. Prvi mehanizam je assert: on omogućava da se proveri da li je element na stranici. Ako on nije dostupan, test će biti prekinut u određenom koraku koji se nije uspešno izvršio. Drugi mehanizam je verify: on takođe omogućava da proverimo da li je element na stranici, ali čak i ako nije, test će nastaviti dalje izvršavanje.

Kako bismo dodali assert ili verify u testove, moramo da koristimo sadržaj menija koji Selenium IDE dodaje u Firefox. Potrebno je da kliknemo desnim dugmetom na element, nakon čega će se pojaviti meni kao na slici 3 .



Slika 3.3.3 Padajuci meni za proveru elemenata

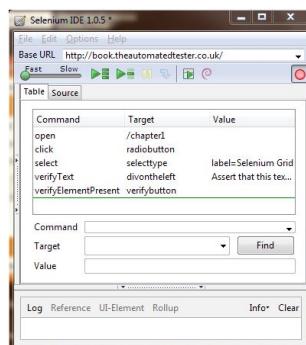
AŽURIRANJE TESTA NASTAVAK

Ažuriranje testa za verifikaciju stavki

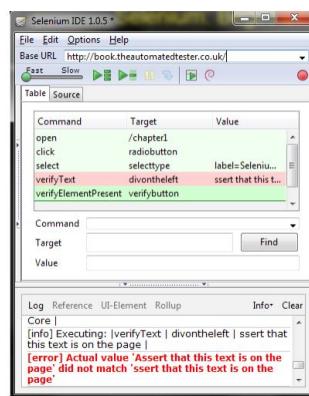
1. Otvoriti Selenium IDE, tako da započnemo snimanje testa.
2. Postaviti Base URL na <http://book.theautomatedtester.co.uk/>.
3. Kliknuti na Chapter1.
4. Kliknuti na radio dugme.
5. Izabrati u padajućoj listi Selenium Grid.
6. Proveriti da li postoji sledeći tekst desno od padajuće liste: Assert that this text is on the page. To možemo da uradimo tako što kliknemo desnim dugmetom miša na element i dobićemo meni, slično kao na slici 3.
7. Proveriti da li je dugme na stranici. I u tom slučaju koristi se padajući meni.
8. Kada završimo prethodne korake, Selenium IDE treba da izgleda kao na slici 4.

Ako sada pokrenemo izvršavanje testa, videćemo da će provere koje smo dodali da budu deo testa. Test će sada proveriti da li na stranici postoji tekst, koji smo odabrali da proverimo, i da li postoji dugme na stranici. Šta bi se dogodilo da provera nije uspešno završena? Javila bi se greška da očekivana komanda nije izvršena, ali bi test bio izvršen do kraja, što možemo videti na slici 5.

Test ne bi nastavio izvršavanje, da smo u testu koristili proveru assert kao mehanizam za proveru da li elementi postoje na stranici. Videli smo da provere elemenata moraju uvek da se dodaju kao manuelni korak (Selenium ne radi automatski assert/verify)



Slika 3.3.4 Dodavanje komandi za proveru elemenata



Slika 3.3.5 Testiranje

RAD SA VIŠE PROZORA

Upotreba Selenium frameworka za kreiranje testa iz više prozora

Rad sa više prozora može biti jedna od najtežih stvari pri pravljenju Selenium testa. To znači da pretraživač mora dozvoliti Selenium-u da programski zna koliko dece pretraživača procesa je napravio. U ovim primerima, mi ćemo kliknuti na element na stranici, koji će izazvati otvaranje novog prozora. U slučaju da je uključena blokada pop up prozora, potrebno ga je isključiti za korišćenje ovih primera.

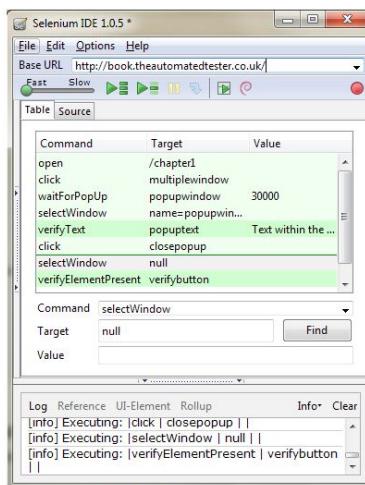
Koraci:

1. Otvorićemo Selenium IDE i odabrat stranicu Chapter1.
2. Kliknuti na jedan element na stranici sa sledećim tekstom Click this link to launch another window. To će izazvati otvaranje novog prozora.
3. Kada se prozor učita, kliknuti na Close the window tekst unutar prozora.
4. Dodati verify komandu za element na stranici. Test treba da izgleda kao na slici 6.
5. Kliknuti na Close the window link.
6. Proveriti element u originalnom prozoru.

U ovom test skriptu, možemo da primetimo da kada otvorimo novi prozor, ubacuje se

komanda `waitForPopUp`. Ovo se dešava zato što test zna da sačeka veb server da obradi zahtev i da generiše stranicu. Takođe, bilo koja komanda koja čeka da se učita veb stranica sa servera, ima komandu `waitFor`. Sledeća komanda je `selectWindow` komanda. Ta komanda govori Selenium-u da prebací kontekst na prozor koji se zove `popupwindow` i da će se izvršavati sve komande koje slede u tom prozoru, osim ako drugačije ne bude rečeno drugom komandom.

Kada test završi rad sa pop-up prozorom, on će se vratiti u roditeljski prozor sa početka testa. Da bismo to uradili, moramo da označimo `null` za prozor. Ovo će naterati `selectWindow` da pomeri kontekst testa u pozadini na roditeljski prozor.



Slika 3.3.6 Dodavanje verifikacije za prozor

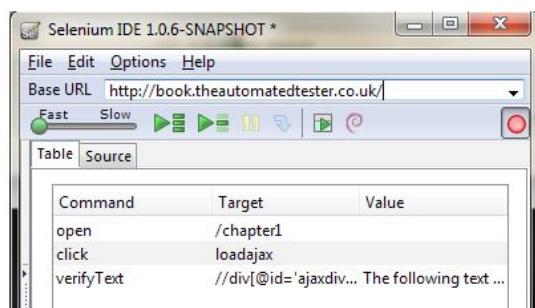
SELENIUM TESTOVI NAD AJAX APLIKACIJAMA

Primena Selenium frameowrka za testiranje AJAX web aplikacija

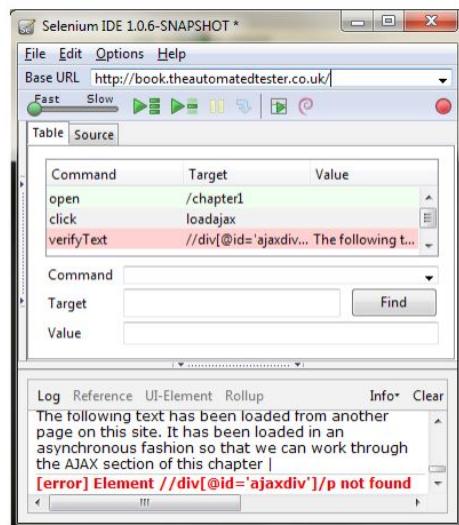
Veb aplikacije danas su dizajnirane tako da su slične desktop aplikacijama. Brzina u radu sa veb aplikacijama dobijena je korišćenjem AJAX tehnologije. AJAX je skraćenica za Asynchronous JavaScript and XML zato što se oslanja na JavaScript tehnologiju za stvaranje asinhronih poziva i vraćanje XML sa podacima koje korisnik ili aplikacija zahteva za nastavak rada. AJAX se ne oslanja na XML, zato što sve više i više ljudi prelazi na JavaScript Object Notation (JSON), koji predstavlja lak način za prenos podataka.

U našem primeru, kliknućemo na neki link i proveriti da li je neki tekst vidljiv na ekranu:

1. Pokrenuti Selenium IDE i pritisnuti dugme Record.
2. Kliknuti na tekst Click this link to load a page with AJAX.
3. Proveriti da li se tekst pojavio na ekranu. U testu treba da se pojavi sličan sadržaj kao na slici 7. Selenium će generisati sve.
4. Pokrenuti test koji smo kreirali. Kada se završi sa radom, treba da izgleda kao na slici 8. lokatore koji su potrebni u ovom testu



Slika 3.3.7 Izgled testa nakon Ajax komande



Slika 3.3.8 Greška prilikom nalaženja teksta dobijenog ajaxom

SELENIUM TESTOVI NAD AJAX APLIKACIJAMA - NASTAVAK

Primena Selenium testa u AJAX aplikacijama

Ako pogledate stranicu, koju smo pokrenuli, vidimo da se tekst nalazi na njoj, nakon klika na link. Dakle, tekst se pojavljuje, a naš test nije prošao - zašto? Test nije prošao uspešno, zato što smo stigli do te tačke izvršavanja testa, a element koji sadrži taj tekst se još uvek nije učitao u DOM (Document Object Model). To se dešava zbog toga što je element zahtevan i renderovan sa veb servera u pretraživač. Kako bismo prevazišli ovaj problem, mi ćemo morati da dodamo novu komandu u naš test, tako da naši testovi prolazi u budućnosti.

1. Desnim dugmetom miša kliknuti na korak koji nije prošao pa će se pojaviti meni.
2. Kliknuti na Insert New Command.
3. U Command select box, postaviti waitForElementPresent ili odabratи iz padajuće liste.
4. U Target box, dodati cilj koji se koristi u verifyText komandi
5. Pokrenuti test ponovo, i on treba da prođe nakon ovoga.

ČUVANJE TESTA I UDALJENI PRISTUP

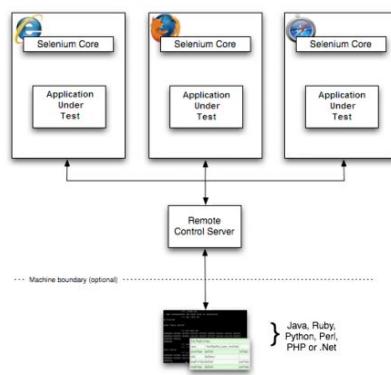
Čuvanje kreiranog testa pomoću Selenium frameworka i pokretanje testa korišćenjem Selenium remote controle

Snimanje testova se vrši na isti način kao i snimanje grupe testova. Kliknuti na File i odabratи Save Test Case. Ovo će prikazati dijalog za snimanje, tako da kasnije možete pokrenuti neki od snimljenih testova.

Selenium IDE pruža testiranje samo preko Firefox-a, što predstavlja manu jer korisnici koriste i druge pretraživače poput Internet Explorer, Google Chrome i Operu. Selenium RC (Remote Control) je alat koji nam omogućava testiranje preko različitih pretraživača koji pritom ne moraju da imaju instaliran Selenium Core na web serveru. Selenium RC se ponaša kao proxy između testirane aplikacije i samih test skripti. Selenium Core je spojen sa Selenium RC-om umesto da bude instaliran na server.

Komponente Selenium RC-a su:

- Selenium server koji pali i gasi pretraživače, interpretira i izvršava Selenium komande dobijene od test programa, ponaša se kao HTTP proxy, presreće i verificuje HTTP poruke razmenjene između pretraživača i AUT(application under test).
- Klijentske biblioteke koje sprovode interfejs između svakog programskog jezika i Selenium RC servera.
- Klijentske biblioteke komuniciraju sa serverom slanjem svake Selenium komande na izvršavanje. Zatim server prosleđuje komandu pretraživaču koristeći Selenium Core JavaScript komande. Pretraživač, koristeći JavaScript Interpreter, izvršava Selenium komandu.

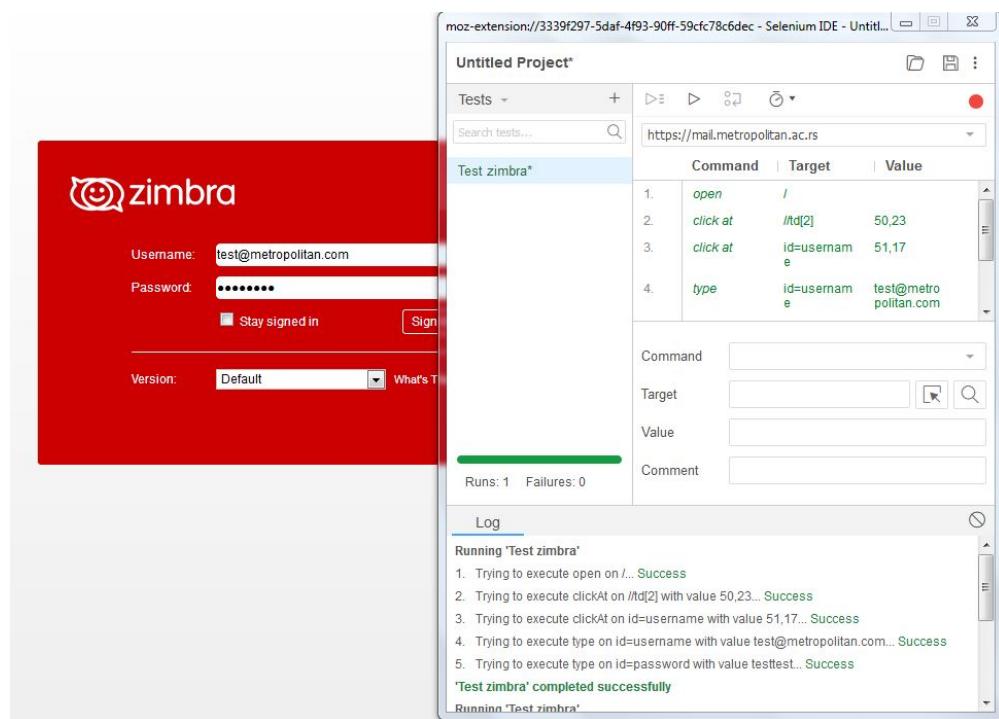


Slika 3.3.9 Prikaz Remote server arhitekture

PRIMER UPOTREBE SELENIUM TESTOVA

Dat je primer korišćenja Selenium testova.

Konkretni primer korišćenja Selenium testova:



Slika 3.3.10 0 Pokretanje Selenium testa

Na Zimbra mejl klijentu kroz Mozilla Firefox pretraživač pokrenut je Selenium IDE sa prethodno objašnjenim koracima. Sačuvan je projekat u kome se nalaze snimljeni koraci (klik na polje username, unos teksta, klik na polje password i unos teksta). Kao što je navedeno korisnik mora prvi put da uradi sve korake a zatim nakon svakog pokretanja testiranje se vrši automatski. Na slici 10 prikazan je Log gde se vidi da Selenium IDE vrši pokretanje testa i unosi prethodno definisan tekst u određena polja. Na korisniku je da odabere šta želi da testira i kojim redosledom.

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci za upotrebu Selenium testova.

- 1. Zadatak:** Testirati web aplikaciju po sopstvenom izboru korišćenjem Selenium framework-a.
- 2. Zadatak:** Predstaviti dobijene rezultate testiranja. Da li je ostvarena ušteda u vremenu potrebnom za testiranje? Koji su uočeni nedostaci u toku testiranja uz pomoć Selenium framework-a?

✓ Poglavlje 4

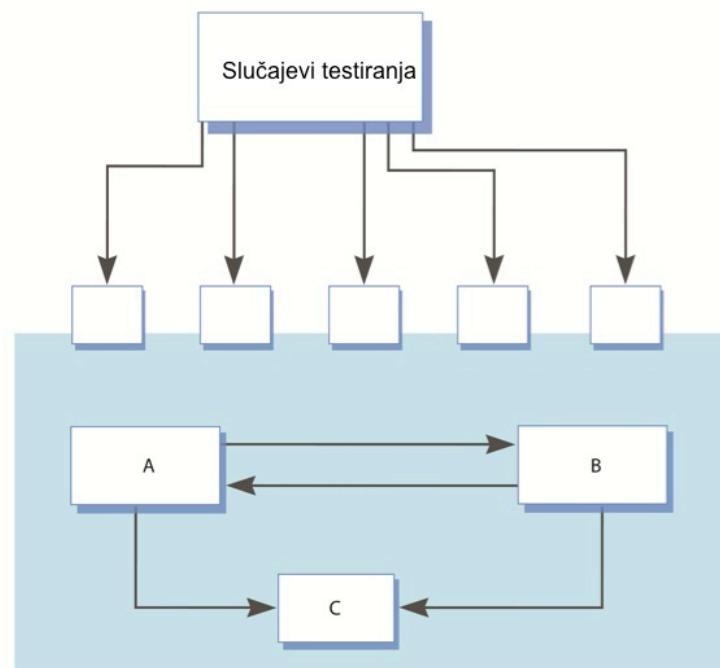
Testiranje komponenata

TESTIRANJE INTERFEJSA KOMPONENTE

Testiranje komponente se fokusira na interfejs komponente s ciljem da se proveri da li se ponaša u skladu sa specifikacijom

Softverske komponente najčešće sadrže nekoliko međusobno povezanih objekata, tj. koji su u interakciji. Testiranje komponente se fokusira na interfejs komponente s ciljem da se proveri da li se on ponaša u skladu sa specifikacijom. Prepostavka je da se pojedinačni objekti (koji realizuju operacije navedene u interfejsu) već prošle svoje (jedinične) testove.

Slika 1 prikazuje ideju testiranja interfejsa jedne komponente koja u sebi sadrži tri manje (jedinične) komponente A, B i C. Prethodno testiranje jediničnih komponenata A, B i C nije moglo da ukaže na greške koje su vezane za interfejs, jer su te greške posledica interakcija između jediničnih komponenata A, B i C.



Slika 4.1

VRSTE INTERFEJSA

Vrsta interfejsa usplovaljava i mogućnost javljanja greške određenog tipa.

Kako postoji više različitih vrsta interfejsa između komponenti programa, to se mogu javiti i više različitih vrsta grešaka interfejsa.

1. **Interfejsi sa parametrima:** To su interfejsi koji sadrže podatke, a ponekad i reference funkcija. Koje se prenose sa jedne na drugu komponentu. Metodi neke klase koriste interfejse sa parametrima.
2. **Interfejsi sa zajedničkom memorijom:** Ovo su interfejsi u kojima komponente zajednički koriste jedan blok memorije. Podatke mogu da u memoriju ubaci jedna komponenta, a druga da ga preuzme i dalje obrađuje. Ovакве interfejse najčešće koriste ugrađeni sistemi (engl. **embedded systems**) u kojima senzori kreiraju podatke koje preuzimaju i obrađuju druge komponente.
3. **Proceduralni interfejsi:** To su interfejsi u kojima jedna komponenta sadrži skup procedura koje mogu da pozivaju druge komponente. Objekti i višestruko upotrebljive komponente koriste ovaj oblik interfejsa.
4. **Interfejsi za propust poruka:** Ovo su interfejsi u kojima jedna komponenta zahteva neki servis druge komponente na taj način što joj pošalje poruku. Povratna poruka sadrži rezultat izvršenja servisa. Objektno-orientisani sistemi imaju ovaj oblik interfejsa, a takođe i klijent-server sistemi..

VRSTE GREŠAKA KOD INTERFEJSA

Greške potiču od (1) lošeg korišćenja interfejsa, (2) nerazumevanja interfejsa i (3) usled neusklađenosti brzina rada komponenti koje su u vezi posredstvom interfejsa.

Greške kod interfejsa su jedna od najčešćih grešaka kod složenih sistema. Ove greške se mogu podeliti u tri kategorije:

1. **Loše korišćenje interfejsa:** Jedna komponenta poziva drugu komponentu i pri tom pravi grešku u korišćenju interfejsa. Ova vrsta grešaka se naročito javlja kod interfejsa sa parametrima, kada se koriste pogrešni tipovi parametara, ili kada se koristi pogrešan redosled parametara, ili kada se koristi pogrešan broj parametara.
2. **Nerazumevanje interfejsa:** Komponenta koja poziva ne razume specifikaciju interfejsa komponente koju poziva i pravi prepostavke o njenom ponašanju. Pozvana komponenta, se zbog toga, ponaša neočekivano.
3. **Vremenske greške:** Ovo se javlja kod sistema u realnom vremenu koji upotrebljavaju interfejse sa zajedničkom memorijom, ili sa propustom poruka. Proizvođač podataka i korisnik podataka mogu da rade sa različitim brzinama. Ako se na ovo ne obrati pažnja pri projektovanju interfejsa, korisnik podataka može da

pristupa zastarem podacima, jer proizvođač podataka još nije postavio nove podatke koje treba zajednički da koriste.

Testiranje grešaka kod interfejsa je otežano jer neki interfejsi daju greške samo u nekim neuobičajenim uslovima. Na primer, ako neki objekat koristi red čekanja (engl. **queue**) sa fiksnom dužinom strukture podataka, a objekt koji poziva pretpostavlja da se primjenjuje red čekanja sa beskonačnom strukturu podataka i ne proverava red čekanja na prekoračenje kada unosi jednu stavku. Ovaj uslov se može detektovati za vreme testiranja projektovanjem slučaja testiranja koji treba da prouzrokuje da se red čekanja prepuni (engl., **overflow**) a to onda prouzrokuje nepravilno ponašanje objekta.

Drugi problem se može javiti zbog interakcija između različitih modula ili objekata koji sadrže greške. Greška u jednom objektu se može se može detektovati kada se neki drugi objekt ponaša na neočekivan način. Na primer, jedan objekat poziva drugi objekat da bi dobio neki servis i pretpostavlja da je dobijeni odgovor ispravan. Ako je pozvani servis sa greškom, povratna vrednost može biti važeća, ali netačna. U prvi mah se to ne vidi, ali postaje očigledno kada kasniji proračuni postaju pogrešni.

OPŠTA PRAVILA TESTIRANJA KOMPONENTA

Inspekcije i recenzije ponekad mogu biti troškovno efektivnije nego testiranje radi otkrivanja grešaka

Pri testiranju komponenta mogu se koristiti sledeća opšta pravila:

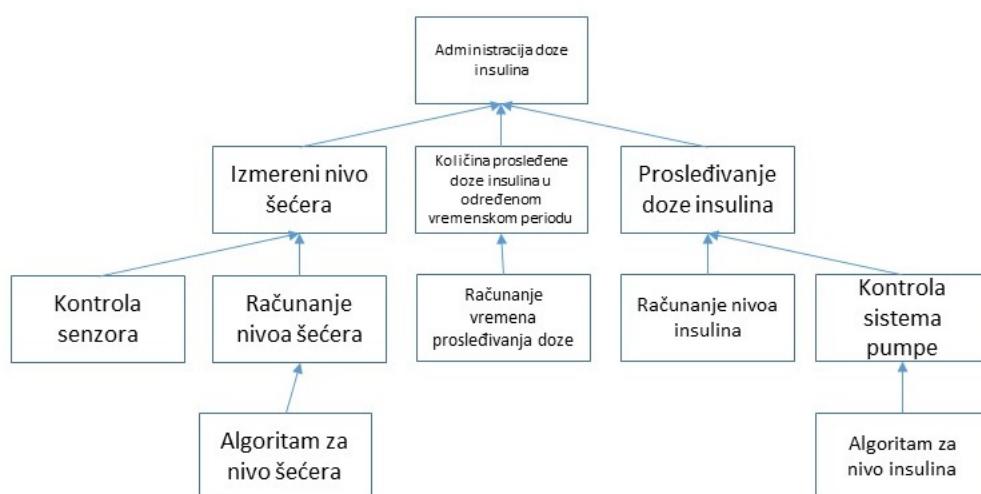
1. Ispitajte kod koji se testira i eksplisitno izlistajte svaki poziv neke spoljne komponente. Projektujte skup testova u kojima vrednosti parametara koji se šalju spoljnim komponentama imaju ekstremne vrednosti u odnosu na dozvoljeni opseg vrednosti. Ove ekstremne vrednosti dovode do javljanje nekonsistentnosti interfejsa.
2. Kada se preko interfejsa šalju pokazivači (engl. pointers), uvek testirajte interfejs sa „null“ parametrima pokazivača.
3. Kada se komponenta poziva preko proceduralnog interfejsa, projektujte testove koji prouzrokuju komponentu da „padne“. Različite pretpostavke za padove u radu su jedna od najčešćih posledica nerazumevanja specifikacije.
4. Upotrebite „stress testing“ (testovi preopterećenja) u sistemima sa prenosom poruka. Ovo znači da bi trebalo da projektujete testove koji generišu mnogo više poruka nego što se mogu javiti u praksi. Ovo je efektivan način da se izazovu problemi sa vremenom.
5. Kada nekoliko komponenti komuniciraju preko zajedničke memorije, projektuj testove koji menjaju redosled aktiviranja ovih komponenta. Ovi testovi mogu izazvati implicitne pretpostavke programera o redosledu proizvodnje i korišćenja podataka.

Inspekcije i recenzije ponekad mogu biti troškovno efektivnije nego testiranje radi otkrivanja grešaka. Inspekcije se koncentrišu na interfejs komponenta i ispituju pretpostavljeno ponašanje interfejsa. Strogo tipizirani jezici, kao što je Java, dozvoljavaju otkrivanje mnogih grešaka od strane kompjajlera. Statički analizatori mogu da detektuju širok opseg grešaka interfejsa.

PRIMER TESTIRANJA KOMPONENTA NA SISTEMU ZA UPRAVLJANJE INSULIN PUMPOM

Primer koji prikazuje način testiranja komponenta sistema, interfejsa između komponenta i svih komponenta istovremeno.

Testiranje komponenta moguće je izvršiti na nekoliko načina. Jedan od načina je pristup odozdo nagore u okviru koga se vrši jedinično testiranje svake komponente sistema polazeći od najnižeg nivoa hijerarhije sistema. Ovaj način omogućava testiranje svih komponenta u sistemu i ponavlja se sve dok svaka komponenta sistema ne bude obuhvaćena. Na slici 2 dat je prikaz komponenti softverskog sistema za upravljanje insulin pumpom.



Slika 4.2 Komponente sistema za upravljanje insulin pumpom

PRIMER TESTIRANJA KOMPONENTA PRIMENOM TEHNIKE ODOZDO NAGORE

Tehnika odozdo nagore omogućava prolazak kroz sve komponente sistema, njihovo testiranje i validaciju.

Primenom pristupa odozdo nagore potrebno je izvršiti testiranje polazeći od komponenti najnižeg nivoa. U slučaju sistema sa slike 2 prvo se testiraju komponente: algoritam za nivo šećera u krvi i algoritam za nivo insulina. Ukoliko se ne pronađe greška u navedenim komponentama pojedinačno se testiraju komponente narednog nivoa a to su: kontrola senzora, računanje nivoa šećera, računanje vremena prosleđivanja doze, računanje nivoa insulina, kontrola sistema pumpe. Ukoliko se ni u ovim komponentama nakon jediničnog testiranja ne pronađe greška prelazi se na sledeći nivo.

Sledeći nivo testiranja je grupisanje komponenti na sledeći način: algoritam za nivo šećera, računanje nivoa šećera, kontrola senzora i izmereni nivo šećera. Na taj način vrši se testiranje određene grupe komponenti (u koju spadaju i prethodno samostalno testirane navedene komponente). U okviru ovog nivoa izvršeno je grupisanje komponenta sa komponentama

koje pozivaju i koje su prošle testiranje. Ukoliko se pojavi greška u toku testiranja jasno je da problem u komponenti izmereni nivo šećera ili u interfejsu između komponenti algoritam za nivo šećera i računanje nivoa šećera ili u interfejsu između komponenti kontrola senzora, računanje nivoa šećera i izmereni nivo šećera. Prethodnim testiranjem komponenti nižeg nivoa eliminisana je mogućnost za pojavu grešaka. Identičnim pristupom grupisanja vrši se testiranje komponenti algoritam za nivo insulina, računanje nivoa insulina, kontrola sistema pumpe i prosleđivanje doze insulina. Nakon toga moguće je izvršiti i testiranje interfejsa između komponenti računanje vremena prosleđivanja doze i količina prosleđene doze insulina u određenom vremenskom periodu. Poslednja stavka u testiranju komponenti načinom odozdo nagore predstavlja testiranje svih komponenti zajedno.

Ovakav vid testiranja odgovara objektno-orientisanom načinu programiranja i testiranju definisanih objekata programa.

ZADACI ZA SAMOSTALNI RAD

Zadaci koji obuhvataju testiranje komponenata sistema.

- 1. Zadatak:** Predstaviti komponente sistema za merenje metereoloških podataka i vezu između komponenata tako da postoji nekoliko nivoa komponenata i interfejsa između njih.
- 2. Zadatak:** Na osnovu primera datog na slici 2 izvršiti testiranje komponenata sistema za merenje metereoloških podataka primenom tehnike odozdo nagore.
- 3. Zadatak:** Identifikovati i opisati moguće probleme i nedostatke u procesu testiranja komponenata sistema za merenje metereoloških podataka primenom tehnike odozdo nagore.

▼ Poglavlje 5

Testiranje sistema

FOKUS TESTIRANJA SISTEMA

Testiranje sistema proverava da li su komponente kompatibilne, da li ispravno međusobno komuniciraju i da li prenose prave podatke u pravo vreme preko njihovih interfejsa.

Testiranje sistema za vreme razvoja obuhvata sve integrisane komponente koje čine jednu verziju sistema, te se na taj način testira integrisan sistem. Testiranje sistema proverava da li su komponente kompatibilne, da li ispravno međusobno komuniciraju i da li prenose prave podatke u pravo vreme preko njihovih interfejsa. Mada ima preklapanja sa testiranjem komponenata, ipak postoje dve važne razlike:

1. Za vreme testiranja sistema, komponente koje su nezavisno razvijene, kao i gotovi sistemi, integrišu se sa novo razvijenim komponentama, te se onda testira ceo sistem.
2. Komponente koje su razvijene od pojedinih članova razvojnog tima se integrišu u ovoj fazi. Testiranje sistema je jedan kolektivni, a ne individualni proces. U nekim kompanijama testiranje sistem vrši poseban tim bez uključenja projektanata i programera koji su razvili sistem.

Kada se integrišu sve komponente sistema, očekuje se da sistem pokaže ponašanje u skladu sa specifikacijom. To se mora testom utvrditi. Mora se razviti test za proveru da sistem radi samo ono što se od njega i očekuje.

Testiranje sistema se fokusira na testiranje interakcija komponenata sistema. Ovaj test interakcija bi trebalo da otkrije komponente sa greškama koje se otkrivaju samo kada je komponenta u interakciji sa drugim komponentama sistema. Testiranje sistema takođe doprinosi uklanjanju nerazumevanja među projektantima komponenata.

Kako je fokus na interakcijama, primena testiranja slučajeva korišćenja (engl., **use case-based testing**) je efektivan način testiranja sistema. Najčešće jedan slučaj korišćenja sistema se realizuje korišćenjem nekoliko komponenti ili objekata sistema. Testiranje slučajeva korišćenja provokira javljanje ovih interakcija. Ako ste razvili sekvencijalni dijagram radi modelovanja slučaja korišćenja, onda možete videti objekte ili komponente koje su uključenje u interakciju.

SLUČAJEVI TESTIRANJA SISTEMA

Na osnovu sekvenčnog dijagrama slučajeva korišćenja sistema utvrđuju se operacije koje se treba da testiraju i vrši se projektovanje slučajeva testiranja za izvršenje testova

Radi ilustracije, na slici 1 je prikazan sekvenčni dijagram jednog slučaja korišćenja stanice za prikupljanje podataka o vremenskim prilikama. Prikazan je slučaj korišćenja koji treba da izvesti udaljeni kompjuter o prikupljenim podacima o vremenu.

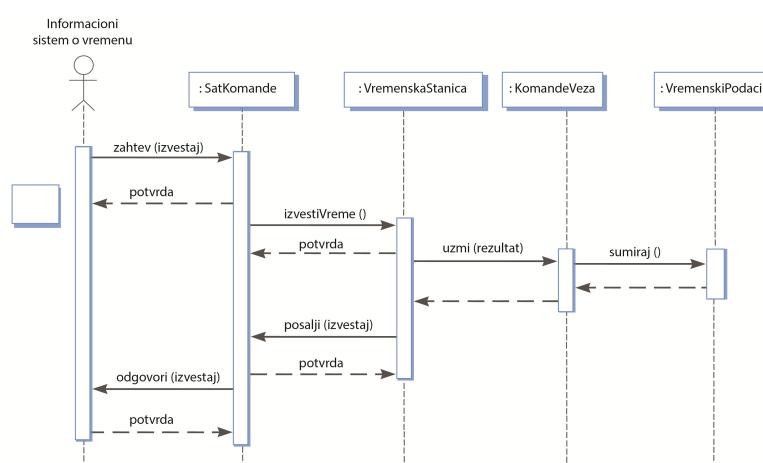
Na osnovu ovog dijagrama utvrđuju se operacije koje se treba da testiraju i vrši se projektovanje slučajeva testiranja za izvršenje testova. Zahtev da se isporuči izveštaj, pokreće izvršenje sledećeg redosleda izvršenja metoda:

SatKomun:zahtev→VremenskaStanica:izvestiVreme→KomunVeza:daj (izvestaj)
→VremenskiPodaci:sumiraj

Sekvenčni dijagram pomaže da se utvrdi koji su ulazi potrebni da bi se kreirali navedeni izlazi.

1. Zahtev za izveštaj treba da ima pridodatu potvrdu. Izveštaj treba da se prosledi onom koji ga je zahtevao. Za vreme testiranja, treba da se kreiraju sumarni rezultati koji se koriste za proveru da li je izveštaj tačan.
2. Ulazni zahtev za izveštaj upućen ovjektu VremenskaStanica se dobija u obliku generisanog sumiranog izveštaja, Ovo s može i posebno testirati izradom podataka koji odgovaraju sumarnom izveštaju koji se testira kod objekta SatKomun. I

koji proverava da li objekat VremenskStanica tačno proizvodi sumarni izveštaj. Postavljeni podaci se takođe koriste za testiranje objekta VremenskiPodaci.



Slika 5.1 Sekvenčni dijagram za slučaj upotrebe prikupljanja podataka o vremenu

KAKO ODREDITI SLUČAJEVE TESTIRNJA?

Slučajevi testiranja se definišu u skladu sa specifikacijom komponenti i u skladu sa iskustvom upotrebe sistema.

Nemoguće je sprovesti sveobuhvatno i detaljno testiranje sistema. Zato se ono sprovodi kreiranje skupa slučajeva testiranja. Kako odrediti ove slučajeve?

U idealnom slučaju, to se može uraditi u skladu sa specifikacijama komponenata. One bi trebalo da definišu način testiranja. Drugi način je da se testiranje oslanja na iskustvo od upotrebe sistema i da se testiranja usmere na testiranje svojstava sistema u radu. Na primer:

1. Sve funkcije sistema kojima se pristupa sa menija su testirane.
2. Kombinacija funkcija kojima se pristupa preko menija moraju da budu testirane.
3. Tamo gde se javlja unos od strane korisnika, moraju se sve funkcije testirati sa tačnim i netačnim ulazima.

Automatizovanp testiranje sistema je obično teže sprovodljivo u odnosu na automatizovano testiranje jedinica. Pri automatizovanom testiranju jedinica, unosi se predviđeni rezultat u sam program. Predikcija se onda poredi sa dobijenim rezultatom Međutim, u slučaju testiranja sistema, mogu se dobijate obimni rezultati i koji se ne mogu lako predvideti. Zato se ispituje dobijeni rezultat i vrši provera njegove kredibilnosti bez njegovog kreiranja unapred, tj. bez predviđanja unapred, kako će izgledati rezultat.

PRIMER TESTIRANJA SISTEMA ZA UPRAVLJANJE INSULIN PUMPOM

Primer se odnosi na testiranje sistema za upravljanje insulin pumpom koristeći definisane test slučajeve.

Cilj testiranja sistema za upravljanje insulin pumpom je da proceni da li je posmatrano ponašanje testiranog softvera u skladu sa specifikacijama. Testiranje obavlja tim koji je nezavisan od projektanata i programera sistema.

Tim je upoznat sa ulaznim i izlaznim podacima kao i akcijama sistema, pa testira sistem koristeći se ispravnim i neispravnim ulaznim podacima. Ovo testiranje je bazirano na test scenarijima (test cases) koji se prave na osnovu funkcionalne specifikacije zahteva.

Način sprovođenja testiranja:

- Sistem je tretiran kao celina
- Prilikom testiranja nije gledan programski kod ili unutrašnja struktura sistema
- Ukoliko su neke od funkcija koje sistem treba da izvrši poznate, testovi se usmeravaju ka demonstraciji da je svaka potpuno operaciona ali se u isto vreme traže i greške u svakoj funkciji
- U sistem su uneti ulazni podaci, praćen je izlaz i na kraju je donesena odluka da li je sistem prošao ili pao test.

Kao što se vidi na slici 2 za svaki test slučaj sistema potrebno je definisati slučaj korišćenja, funkciju koja se testira, inicijalno stanje, ulaz i izlaz. Dobijeni rezultati prikazuju da su svi test slučajevi prošli bez uočene greške. Ukoliko dođe do identifikovanja greške, potrebno je ponoviti testiranje i utvrditi iz kog razloga se uočena greška javlja.

Slučajevi korišćenja	Funkcija koja se testira	Inicijalno (početno stanje)	Ulaz (komanda, akcija)	Izlaz (odgovor sistema, krajnje stanje)
Merenje nivoa šećera u krvi	Merenje nivoa šećera u krvi korisnika.	Uslov je da je korisnik pristupio sistemu i odabrao opciju za merenje nivoa šećera u krvi.	Korisnik unosi datum i vreme (sate i minute) kada će biti izvršeno merenje insulina u krvi.	Korisnik dobija poruku o izmerenom nivou šećera u krvi na osnovu zadatog vremena.
Merenje doze insulina	Merenje količine datog insulina	Uslov je da je korisnik pristupio sistemu i odabrao opciju za merenje količine insulina u krvi.	Korisnik unosi datum i vreme za izvršavanje merenja doze insulina.	Korisnik dobija informaciju o dozi insulina koja se nalazi u pumpi. Na osnovu ove informacije korisnik može koristiti druge funkcionalnosti sistema.
Pregled režima rada insulin pumpe	Pregled režima rada pumpe	Uslov je da je korisnik pristupio sistemu i odabrao opciju za pregled rada pumpe.	Korisnik može vršiti pregled režima rada insulin pumpe, menjati režim unosom novog režima ili modifikacijom postojećeg. Ulazni podatak je parametar pregleda režima rada koji je moguće menjati shodno potrebama korisnika.	Korisnik dobija informaciju o režimu rada insulin pumpe u datom vremenskom intervalu.

Slika 5.2 Test slučajevi sistema za upravljanje insulin pumpom

ZADACI ZA SAMOSTALAN RAD

Dati su zadaci obuhvataju testiranja sistema.

1. Zadatak: Projektovati pet test slučajeva za proizvoljnu aplikaciju. Opisati test slučajeve kao u prethodnom primeru na slici 2.

2. Zadatak: Definisati fokus testiranja proizvoljne aplikacije na osnovu projektovanih test slučajeva. Obrazložiti donesenu odluku.

✓ Poglavlje 6

Testovima vođen razvoj sistema

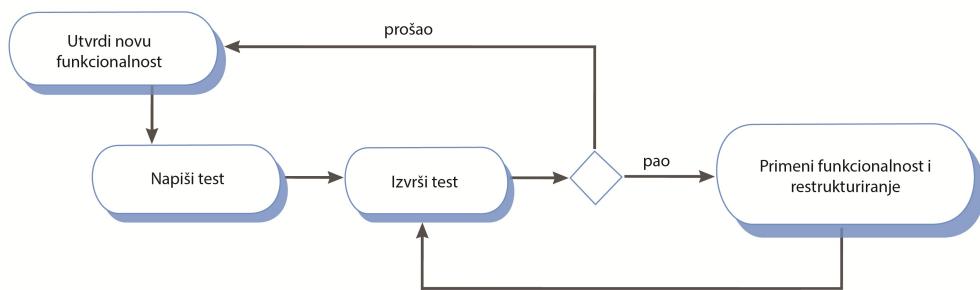
RAZVOJ SOFTVER VOĐEN TESTOVIMA

Razvoj vođen testovima je pristup razvoju programa u kome se istovremeno vrši i razvoj koda i njegovo testiranje.

Razvoj vođen testovima je pristup razvoju programa u kome se istovremeno vrši i razvoj koda i njegovo testiranje. Kod se inkrementalno razvija, i za svaki inkrement se sprovodi odgovarajući test. Dok se ne završi test jednog inkrementa, ne počinje se razvoj sledećeg.

Slika 1 prikazuje aktivnosti integrisanog razvoja i testiranja softvera.

1. Najpre utvrdite inkrement softvera sa odrešenom funkcionalnošću. Inkrement treba da abude mali i primenljiv u nekoliko linija koda.
2. Napišite test za funkcionalnost inkrementa i примените ga u vidu automatskog testa.
3. Izvršite test, zajedno sa drugim testovima. Trebalo bi da se dobiju uspešni rezultati testa za sve već razvijene inkremente, a negativne za one koji još nisu razvijeni.
4. Sada razvijete inkrement sa svojom funkcionalnošću pisanjem novog, odgovarajućeg koda.
5. Ako je sada test pozitivan, postupak se ponavlja sa utvrđivanjem novog inkrementa.



Slika 6.1 Razvoj softvera vođen testovima

AUTOMATSKO TESTIRANJE

Testovi su ugrađeni u jedan poseban program koji izvršava testove i koji aktivira program koji se testira.

Ako je kod pisan u Javi, onda se obično koristi JUnit za automatsko testiranje inkremenata. U ovom postupku se svi testovi automatski vrše kod testiranja svakog novog inkrementa. Testovi su ugrađeni u jedan poseban program koji izvršava testove i koji aktivira program koji se testira. Na ovaj način, za nekoliko sekundi, sprovede se na stotine testova (za svaki inkrement).

Da bi se napisali testovi, mora da se dobro razume šta je cilj rada svakog inkrementa, tj. njegova funkcionalnost. Zato, primena razvoja vođenim testovima, pomaže većem razumevanju svrhe i načina rada programa od strane programera. Pored toga, razvoj softvera vođenim testovima dovodi i do sledećih koristi:

1. **Pokrivanje koda:** Po pravilu, svaki deo koda ima bar jedan prigodan test. Kod se testira čim se napiše tako da se defekti otkrivaju rano u procesu razvoja.
2. **Regresiono testiranje:** Kako se program za testiranje razvija paralelno sa kodom, možete ga regresiono izvršiti (ponoviti ranije testove) radi provere da li novi softverski inkrement ne uzrokuje neku novu grešku (i u ranije testiranom delu softvera).
3. **Uprošćeno uklanjanje grešaka:** Kada je test negativan (test ukazuje na grešku), nalaženje uzroka greške je jasan, jer se odnosi na upravo napisan softverski inkrement. Taj deo se odmah ispravi i test ponavlja. Nisu potrebni posebni alati za nalaženje lokacije greške.
4. **Dokumentacija sistema:** Testovi ugrađeni u program predstavljaju jedan oblik dokumentacije koja opisuje šta bi kod trebalo da radi. Čitanjem testa vi razumete i kod.

Primena regresionog testiranja je vrlo skupa kod ručnog testiranja, jer treba da nađete i probate neki raniji test da bi videli da li novi deo koda nije uzrokovao javljanje neke greške u drugom delu softvera (koji je ranije razvijen i testiran). Primenom automatskog testiranja, ovaj nedostatak regresionog testiranja se uklanja, te se vrlo brzo proverava da li novi deo softvera nije napravio neku grešku u ranije razvijenom delu softvera, jer se pri svakom izvršenju programa (u fazi razvoja softvera), izvršavaju i svi testovi inkrementalnih delova ranije razvijenog softvera. Automatsko testiranja dramatično smanjuje troškove regresionog testiranja. Programer može lako da se uveri da novi deo koda nije napravio neki problem u ranije razvijenom delu koda.

UPOTREBA RAZVOJA SOFTVERA VOĐENIM TESTOVIMA

Razvoj vođen testovima se najčešće koristi kod razvoja novog softvera.

Razvoj vođen testovima se najčešće koristi kod razvoja novog softvera. Ako u vaš kod ubacujete neki ranije razvijen kod, onda je potrebno da za njega, kao celinu, ubacite automatske testove. Međutim, testovima vođen razvoj ne pokazuje efikasnost u slučaju softvera sa više niti (engl.**multi-threaded systems**), jer različite programske niti mogu preplitati u različitim vremenima u različitim testovima, što onda uzrokuje javljanje različitih rezultata.

Iako upotrebljavate testovima vođen razvoj, vi i dalje treba da koristite proces testiranja sistema, radi potvrde njegove ispravnosti. Time se dokazuje da on, kao celina, zadovoljava sve zahteve njegovih aktera. Testiranje sistema takođe, utvrđuje i performanse sistema,

njegovu pouzdanost, i proverava da li sistem radi nešto što ne bi trebalo da radi, kao što je proizvodnja nepoželjnih rezultata.

Testovima vođen razvoj se pokazao uspešnim kod projekata razvoja male i srednje veličine. Programeri koji ga primenjuju su najčešće vrlo zadovoljni njegovom primenom.

TEST-DRIVEN DEVELOPMENT TUTORIAL: WHAT IS TEST-DRIVEN DEVELOPMENT

Trajanje 4:57 minuta

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

TEST DRIVEN DESIGN

Trajanje 11:30 minuta

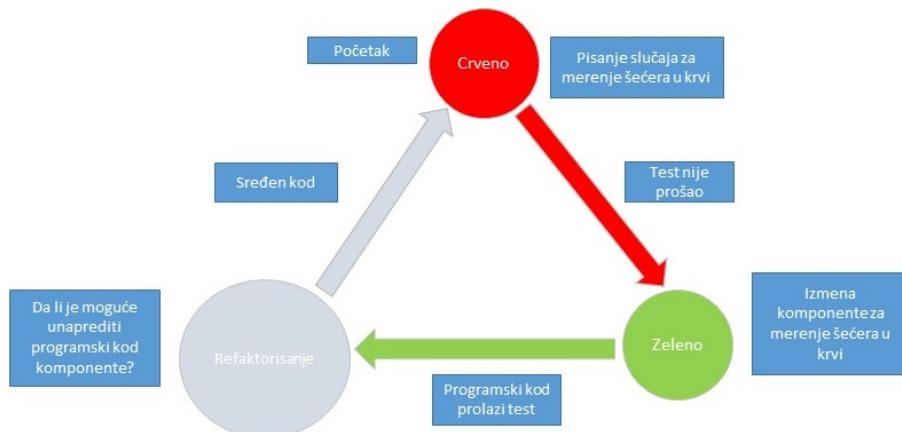
Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

PRIMER RAZVOJA SISTEMA ZA UPRAVLJANJE INSULIN PUMPOM VOĐEN TESTOVIMA

Kroz primer sistema za upravljanje insulin pumpom objašnjen je način razvoja sistema vođen testovima.

Na slici 2 prikazan je slučaj kada sistem za upravljanje insulin pumpom nije prošao test. Test koji je postavljen odnosi se na funkciju sistema koja omogućava merenje šećera u krvi (test slučaj preuzet iz prethodne tematske jedinice testiranje sistema). Postavljeni su ulazni parametri i očekivani izlazni parametri iz sistema. Kako se nakon test slučaja izlazni parametri nisu poklapali sa parametrima definisanim u test slučaju automatski znači da sistem nije prošao test.

Početak obuhvata pisanje test slučaja a crvena faza počinje kada sistem ne prođe test. Sledeći korak predstavlja izmenu programskog koda tako da sistem prođe test slučaj. U slučaju sistema za upravljanje insulin pumpom izmena se vrši unutar komponente za merenje šećera u krvi jer test obuhvata tu funkcionalnost. Ukoliko izvršenje ove funkcije obuhvata i druge komponente sistema u narednom koraku testom je potrebno obuhvatiti i te komponente. Kada programski kod prođe test nalazi se u zelenoj fazi i sledeća faza obuhvata refaktorisanje i sređivanje programskog koda. Ukoliko programski kod ne prođe test proces izmene programskog koda se ponavlja dok sistem ne prođe test. Refaktorisanje omogućava unapređenje prethodno izmenjenog programskog koda koji je prošao test i iz ove faze (obeležene sivom bojom na slici 2) izlazni programski kod je sređen i postupak testiranja se može ponoviti.



Slika 6.2 Testovima vođen razvoj sistema za upravljanje insulin pumpom

Prednost ovakvog načina testiranja je konstantno unapređenje funkcionalnosti sistema i samog programskog koda na osnovu izvršenih test slučajeva.

ZADACI ZA SAMOSTALAN RAD

Dati su zadaci koji obuhvataju testovima vođen razvoj sistema.

1. Zadatak: Prikazati testovima vođen razvoj proizvoljnog sistema (aplikacije):

1. Napraviti proizvoljnu aplikaciju ili koristiti već postojeću.
2. Definisati test slučajeve za testiranje aplikacije.
3. Simulirati grešku u toku testiranja.
4. Prikazati testovima vođen razvoj proizvoljnog sistema (kao na primeru na slici 2)

2. Zadatak: Za svaki inkrement proizvoljnog sistema kreirati odgovarajući test slučaj opisan kroz tabelu. Navesti ulaze i očekivane izlaze testa. Opisati komponente koje je potrebno izmeniti u slučaju da dođe do neprolaska određenog test slučaja (ukoliko postoji više od jedne komponente, navesti i ostale komponente koje je potrebno obuhvatiti test slučajem).

▼ Poglavlje 7

Testiranje pri primopredaji sistema

TESTIRANJE SOFTVERA SPREMNOG ZA ISPORUKU

Primarni cilj procesa testiranja softvera za isporuku je da proizvođač softvera uveri sebe da je sistem dovoljno dobar za upotrebu.

Testiranje softvera spremnog za isporuku je proces testiranja određene verzije (izdanja) sistema koji će se koristiti van rada projektnog tima., tj. kod kupca ili korisnika softverskog sistema. U slučaju velikih projekata, verzija softvera se može raditi i za druge timove u istom projektu. U slučaju softverskih proizvoda, softver za isporuku se predaje menadžmentu proizvoda koji onda pokreće proces prodaje.

Postoje dve razlike između testiranja sistema i testiranja sistema spremnog za isporuku:

1. Poseban tim, koji nije uključen u razvoj sistema, bi trebalo da bude odgovoran za testiranje softvera spremnog za isporuku.
2. Testiranje sistema sprovodi razvojni tim u cilju otkrivanja grešaka u sistemu. Cilj testiranja sistema za isporuku je provera da li sistem zadovoljava specificirane zahteve i svoju specifikaciju svojstava, i provera da li je dovoljno dobar za spoljnu upotrebu (test validacije-potvrđivanja)-

Primarni cilj procesa testiranja softvera za isporuku je da proizvođač softvera uveri sebe da je sistem dovoljno dobar za upotrebu. Ako je dobar, isporučuje se kao proizvod ili šalje naručiocu. Sistem za isporuku mora da ima ranije definisanu funkcionalnost, propisane performanse, utvrđene zavisnosti i da ne prestaje sa radom prilikom normalnog korišćenja. Softver treba da zadovolji sve sistemske zahteve, a ne samo zahteve krajnjih korisnika sistema.

Testiranje softvera za isporuku se odovija u formi „testitanja crne kutije“ jer se zasniva na specifikaciji sistema. Drugi naziv za ovo testiranje je i *funkcionalno testiranje*.

KAKO ODREDITI TESTOVE NA OSNOVU ZAHTEVA?

Testiranje zasnovano na zahtevima je validaciono testiranje, jer treba da utvrdi da li sistem ostvaruje zadate zahteve.

Dobri zahtevi su oni koji su tako napisani da se mogu testirati. Testiranje zasnovano na zahtevima je sistemski pristup projektovanju slučajeva testiranja u kome se uzima u obzir svaki zahtev i dobija skup testova za svaki zahtev. Testiranje zasnovano na zahtevima je

validaciono testiranje (testiranje potvrđivanja) a ne testiranje radi testiranje na greške, jer je cilj da se pokaže da sistem ispravno ostvaruje svoje zahteve.

Na primer, uzmimo sledeće zahteve definisane za MHV-PMS sistem (lekcija 1), a koji su vezani za proveru leka na izazivanje alergije:

- „*Ako pacijent ima alergijske reakcije na neki lek, onda recept teba da sadrži upozorenje za korisnika sistema.*
- *Ako lekar ignoriše upozorenje, onda mora da navede razlog za to“.*

Da bi se proverilo da li sistem zadovoljava ove zahteve, u ujetom primeru sistema MHV-PMC, potrebno je da razvijete nekoliko testova:

1. Uspostavite slog (elektronski zapis) pacijenta za koga nema naznake da je alergičan na neki lek. Propišite lek za koji je poznato da izaziva alergije. Proverite odsustvo poruka upozorenja od strane sistema (što je ispravno).
2. Uspostavite slog pacijenta kome je upisano da ima alergijske reakcije na neki lek. Propišite lek za koji je poznato da izaziva alergije. Proverite da li je sistem generisao poruku sa upozorenjem (što je ispravno).
3. Postavite slog pacijenta koji imaju alergijske reakcije na dva ili više lekova. Propišite nezavisno te lekove. Proverite da li je sistem kod svakog leka generisao upozorenje (što je ispravno).
4. Pišite dva leka na koje je pacijent alergičan. Proverite da li sistem generiše dve poruke upozorenja na pravi način.
5. Propišite lek za koji sistem generiše upozorenje, i onda odbite to upozorenje. Proverite da li sistem zahteva od korisnika da unese informaciju sa obješnjnjem zašto je odbacio upozorenje.

Kao što se vidi, ne mora da se piše samo jedan test za proveru ispunjenja jednog zahteva. Obično je potrebno napisati nekoliko testova da bi se pokrilo u potpunosti jedan zahtev.

Trebalo bi da obezbedite zapisivanje slogova sa izvršenim testiranjima zahteva, a koji povezuju testove sa specifičnim zahtevima koji su testirani.

KORIŠĆENJE SCENARIJA TESTIRANJA

Scenario testiranja je jedan pristup testiranja softvera pre isporuke u kome vi osmislite tipične scenarije korišćenja i onda ih iskoristite za razvoj slučajeva testiranja sistema.

Scenario testiranja je jedan pristup testiranja softvera pre isporuke u kome vi osmislite tipične scenarije korišćenja i onda ih iskoristite za razvoj slučajeva testiranja sistema. Scenario je priča koja opisuje jedan način korišćenja sistema. Scenariji bi trebalo da budu realistični i stvarni korisnici sistema bi trebalo da budu povezani sa njima.

Scenario je priča koja je precizna i umereno složena. Scenario treba da motiviše aktere, tj. da se sebe vide u njemu i da veruju da je važno da sistem prođe taj test. Trebalo bi da omogući laku evaluaciju i da ukaže na mesto javljanja eventualnog problema.

Na slici 1 je prikazan mogući scenario za sistem MHC-PMS koji pokazuje način kako se sistem može koristiti prilikom obilaska pacijenata po kućama.

Kate je medicinska sestra koja je socijalizovana za negu bolesnika sa mentalnim problemima. Njen posao je, pored ostalih, da posećuje pacijente po kućama da bi proverila efekat njihovih terapija i da li imaju neke uzgredne probleme zbog uzimanja lekova.
Kada ide u posetu, Kate se loguje na MHC-PMS sistemi i odštampa raspored njenih vizita za taj dan, zajedno sa sumarnom informacijom o pacijentima koje poseće. Ona zahteva da se op. Od nje sistem zahteva da ukucu ključne fraze da bi se izvršila šifrovanje dosjeda u njenom laptopu.
Jadan od pacijenata koga poseće tог дана је и Džim, који добија лекове против депресије. Džim primećује да му лекови помажу, али primećује и uzgredan efekat – nesanica u toku ноћи. Kate hoće da pogleda anjegov dosje na laptopu, и на почетку је upitana д aotkuca ključnu frazu za sklanjanje šifre sa dosjea. Ona provera propisan lek i ispituje uzgredne efekte. Nesonica je poznat uzgredan efekat te је ona ubeležила problem u Džimov dosje, а njemu sugerише да poseti kliniku da bi mu propisali неки drugi lek. On se сa tim slaže, te Kate upisuje podsetnik da treba da ga zove da bi ga obavestila када се врати на kliniku i kada dogovori susret sa lekarom. Ona završava konsultaciju i sistem ponovo šifrira Džimov dosje. o
Posle, kada je Kate završila sve vizite i konsultacije, Kate se враћа на kliniku i kopira dosjea pacijenata nazad u sistem, tj. stavlja ih u njegovu bazu podataka. Sistem generiše listu poziva vih pacijenata за Kate i onda obezbeđuje listu pacijenata које она мора да kontaktira radi информација о daljim akcijama које slede i dogovara vreme prijema pacijenata kod lekara.

Slika 7.1 Primer jednog scenarija

SVOJSTVA KOJA SE TESTIRAJU NA BAZI DATOG SCENARIJA

Kada primenujete testiranja korišćenjem scenarija, vi najčešće testirate nekoliko zahteva sa jednim scenarijom

Upotreбом navedenog scenarija scenario, mogu se testirati sledeća svojstva MHC.PMS sistema:

1. Provera korisnika prilikom uključenja u sistem.
2. Preuzimanje na laptop i vraćanje u bazu elektronskih dosjeda pacijenata
3. Planiranje poseta pacijenata
4. Stavljanje i skidanje šifri za pristup dosjeima pacijenata na mobilnom uređaju.
5. Pretraživanje dosjea i modifikacija
6. Povezivanje sa bazom podataka o lekovima koja sadrži informacije o uzgrednim efektima lekova.
7. Sistemski poziv za upis.

Ako ste tester, vi koristite ovaj scenario i izvršavate testove, igrajući ulogu Kate. Pratite pri tome kako se sistem ponaša pri različitim ulazima. Kao „Kate“ vi pravite namerne greške u kucanju, као на primer lošu ključnu frazu za dešifrovanje dosjea. Na ovaj način proverava se odgovor sistema na greške. Pratite svaki problem, uključujući i performanse. Ako je sistem isuviše spor, onda se mora promeniti način njegovog korišćenja.

Kada primenujete testiranja korišćenjem scenarija, vi najčešće testirate nekoliko zahteva sa jednim scenarijom. Pored testiranja pojedinačnih zahteva, vi proveravate da li neka kombinacija zahteva ne pravi neki problem.

TESTIRANJE PERFORMANSI SISTEMA

Testiranje se vrši izvršavanjem više testova sa postepenim povećanjem opterećenja sistema sve dok te performanse postanu neprihvatljive.

Kada je sistem potpuno integrisan, može se pristupiti ispitivanju njegovih performansi i pouzdanosti rada. Testovi za merenje performansi treba da budu projektovani tako da provere da li sistem može da radi sa očekivanim performansama kada je pod punim opterećenjem. Testiranje se vrši izvršavanjem više testova sa postepenim povećanjem opterećenja sistema sve dok te performanse postanu neprihvatljive.

Da bi testiranje utvrdili da li vaš sistem zadovoljava zahtevane performanse, morate da pripremi operativni profil. **Operativni profil** (ili profil rada) je skup testova koji odražava stvarnu mešavinu posla koje sistem treba da odradi. To je i preduslov da dobijete tačne operativne performanse sistema. Ali to ne znači i da je to i najefektniji način za otkrivanje grešaka. Praksa je pokazala da je za to najbolje da projektujete testove uzimajući granice sistema. Kod testiranja performansi, to znači da sistem treba opteretiti i preko projektovanih granica softvera. To je tzv. „stres test“. Na primer, ako se sistem projektuje da izvrši 300 transakcija u sekundi, onda se testiranje počinje sa manjim brojem transakcija, a završava sa brojem transakcija značajno većim od 300 u sekundu. Ova vrsta testiranja ima dve funkcije:

1. Testira ponašanje sistema na prestanak rada (pad) sistema. Tada je važno da pad sistema ne dovede do oštećenja podataka ili da dođe do gubitka korisničkih servisa. Stre testiranje proverava da li preopterećenje sistema prouzrokuje „blagi pad“ sistema, umesto kolapsa pod opterećenjem.
2. *On opterećuje maksimalno sistem što može da dovede do javljanja grešaka koje u normalnim okolnostima se ne bi javile.* Takva stresna situacija može doći i kada sistem radi pod normalnim okolnostima, ali se desi neka neuobičajena kombinacija normalnih okolnosti, koja može da proizvede sličan efekat kao što proizvodi preopterećenje sistema.

Testiranje na stres (tj. na preopterećenje) je posebno važno za slučaj distributivnih sistema, koje koriste mrežu procesora. Ovi sistemi često doživljavaju ozbiljan pad performansi kada su značajno opterećeni. Mreža postaje ugušena, procesori se usporavaju jer čekaju potrebne podatke od drugih procesora (preko mreže). Stres testiranje pomaže da se otkrije situacija kada počinje pad performansi tako da vi možete dodati provere, koje bi trebalo da u tim slučaju odbiju transakcije koje povećavaju opterećenje sistema u toj tački.

PRIMER TESTIRANJA PRI PRIMOPREDAJI SISTEMA

Kroz test slučaj testiranja performansi sistema za upravljanje insulin pumpom prikazan je jedan od mogućih načina testiranja pri primopredaji sistema.

Merenje performansi sistema omogućava proveru mogućnosti sistema prilikom upotrebe u realnim uslovima. Za sistem za upravljanje insulin pumpom prikazan je test slučaj merenja performansi prilikom merenja nivoa šećera u krvi. (slika 2)

Test slučaj: Merenje nivoa šećera u krvi				
Opis test slučaja: Test simulira akciju koja se očekuje od sistema u toku svakodnevnog korišćenja. Korisnik pristupa sistemu i vrši merenje nivoa šećera u krvi a sistem beleži dobijeni rezultat nivoa šećera u krvi i vrši ažuriranje nivoa šećera na osnovu poslednjeg merenja.				
Zahtevi:				
Korisniku je potrebno omogućiti pristup sistemu za kontrolu insulin pumpe. {measure_blood_sugar_level} - merenje kao rezultat mora prikazati određenu vrednost (blood_sugar_level) - mora biti u odgovarajućem rasponu vrednosti definisanim unutar sistema Napomena: Očekuje se da jedan korisnik koristi sistem za kontrolu insulin pumpe i da se merenja i ažuriranja nivoa šećera u krvi odnose samo na jednog korisnika.				
Broj koraka	Opis koraka	Očekivani rezultat	Naziv transakcije	Vreme razmišljanja korisnika (u sekundama)
01	Pokretanje sistema	Prikaz početnog ekran	user_login	3
02	Selektovanje merenja nivoa šećera u krvi	Prikazan je ekran za odabir merenja šećera u krvi	select_measure_blood_sugar_level	5
03	Unos nivoa šećera u krvi {blood_sugar_level} u tačno polje.	Uneta je vrednost nivoa šećera u krvi	insert_blood_sugar_level	15
04	Ažuriranje vrednosti {blood_sugar_level} u polju nivoa šećera u krvi	Ažurirana poslednja vrednost nivoa šećera i prikazana korisniku	update_blood_sugar_level	8
05	Izlazak iz menija	Izlazna porukaje prikazana	return_to_start_screen	5

Slika 7.2 Test slučaj merenja performansi pri primopredaji sistema

Test slučaj na slici 2 prikazuje testiranje merenja nivoa šećera u krvi gde je definisano pet koraka u sistemu, očekivani rezultati, transakcije koje se izvršavaju u toku navedenih koraka kao i predviđeno vreme razmišljanja korisnika na određenom koraku. Potrebno je da korisnik pokrene sistem, izvrši odabir opcije za merenje nivoa šećera gde se tokom merenja nivoa šećera vrši unos i ažuriranje vrednosti nivoa šećera u krvi (blood_sugar_level). Kada je vrednost ažurirana, korisnik može pregledati novu vrednost nivoa šećera u krvi i vratiti se na početni ekran gde ponovo može pokrenuti merenje. U toku izvršavanja test slučaja vrši se merenje vremena koje je potrebno da korisnik izvrši određeni korak kao i vreme koje mu je potrebno da se zadrži (razmisli) o tom koraku u sistemu. Vreme razmišljanja u ovom slučaju podrazumeva vreme potrebno da korisnik pronađe opciju i pokrene aktivnost u datom koraku.

U toku testiranja performansi sistema moguće je izvršiti simuliranje rada korisnika kroz skrpite koje će automatski unositi vrednosti merenja nivoa šećera u krvi i ponavljati testiranje sa različitim vrednostima i ažuriranjima. Na taj način, sistem će biti testiran na različite vrednosti nivoa šećera u krvi (blood_sugar_level) i različit broj ažuriranja tih vrednosti što će kao rezultat pokazati koliko je sistem spremjan na konstantna ažuriranja vrednosti i veliki broj unosa u kratkom vremenskom periodu.

ZADACI ZA SAMOSTALAN RAD

Dati su zadaci za testiranje pri primopredaji sistema.

- 1. Zadatak:** Napraviti scenario testiranja pri primopredaji za sistem za merenje metereoloških podataka. Definisati svojstva koja se testiraju na osnovu datog scenarija sistema za merenje metereoloških podataka.
- 2. Zadatak:** Osmisliti pet test slučajeva merenja performansi sistema za merenje metereoloških podataka. Napraviti tabele za svaki test slučaj kao u prethodnom primeru (slika 2). Opisati moguće nedostatke test slučajeva i mogućnosti da se navedeni test slučajevi unaprede.

▼ Poglavlje 8

Korisničko testiranje

TIPOVI KORISNIČKOG TESTIRANJA

Testiranje od strane kupca, je faza procesa testiranja u kojoj naručioci softvera definišu ulazne podatke i interakcije radi donošenja odluke o preuzimanju sistema i njegovog korišćenja.

Korisničko testiranje, ili testiranje od strane kupca, je faza procesa testiranja u kojoj korisnici ili kupci softvera definišu ulazne podatke i interakcije i daju savete za testiranje sistema. To može biti u obliku formalnog testiranja radi preuzimanja softvera od njegovog prodavca, ili u obliku neformalnog procesa u kome korisnici eksperimentišu sa novim softverskim proizvodom da bi videli da li im se sviđa i da li radi ono što im je potrebno. Korisničko testiranje je bitno, jer uticaji iz radnog okruženja korisnika imaju glavni efekat na pouzdanost, performanse, korisnost, i robusnost sistema.

U praksi, postoje tri tipa korisničkog testiranja:

Alfa testiranje, kada korisnici softvera rade zajedno sa razvojnim timom na testiranju softvera u organizaciji koja razvija softver.

Beta testiranje, kada određeno izdanje softvera je dato na raspolaganje pojedinim korisnicima da bi i testirali isporučen softver, eksperimentisali sa njim, i izvestili razvojni tim o problemima na koje su naišli.

Test prihvatanja, kada kupci testiraju sistem da bi odlučili da li je on spremna da bude preuzet od razvojnog tima i instalisan u okruženju kupca.

Kako nisu uvek zahtevi sistema kompletni i precizni, **alfa testiranje** omogućuje da se sistem „dotera“ u skladu sa stvarnim zahtevima korisnika softvera. Alfa testiranje, zajedno sa korisnicima, se najviše primenjuje pri kupovina softvera koji se razvija po specijalnoj narudžbini. Naručioci softvera žele da budu što pre informisani o radu softvera. To im smanjuje rizik od neželjenih svojstava softvera koji su naručili. Takođe, Agilne metode razvoja softvera, kao što je XP, takođe podstiču učešće korisnika softvera u projektovanju testova sistema.

Beta testiranje omogućuje kupcima da ispitaju novo izdanje softvera, čak ponekad, i kad njen razvoj i nije potpuno okončan. Beta testeri su izabrane grupe kupaca koji su prvi korisnici softvera. Beta testiranje se najčešće koristi kod softverskih proizvoda koji se koriste u vrlo različitim korisničkim okruženjima. Zbog toga, beta testiranje je važno za otkrivanje problema interakcije između softvera i sistema u njegovom okruženju.

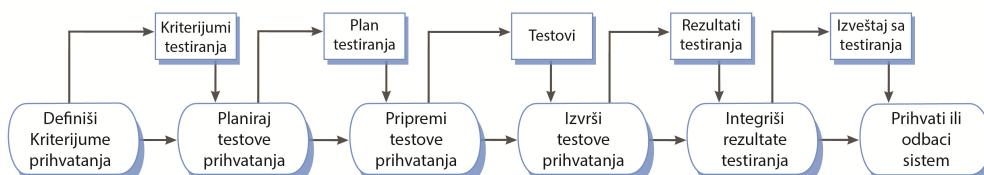
Test prihvatanja je sastavni deo projekta razvoja sistema za određenog kupca, tj. po narudžbini. On se sprovodi posle testiranja izdanja softvera, tj. testa pred isporuku. Test prihvatanja je formalno testiranje sistema radi njegovog preuzimanja od strane naručioca. Zato, posle njega, vrši se plaćanje softvera od strane naručioca.

PROCES TESTIRANJA RADI PRIHVATANJA SISTEMA

Proces prihvatanja softverskog sistema sdrži šest faza: definisanje kriterijuma, planiranje testa, priprema tasta, izvršavanje testa, analiza rezultata i odbijanje ili prihvatanje sistema.

Postoje šest faza procesa testiranja softvera radi njegovog prihvatanja:

1. **Definisanje kriterijuma** prihvatanja: Idealno, ovi uslovi se definišu ugovorom o kupovini softvera..
2. **Planiranje testa prihvatanja**: Planiraju se resursi, vreme, i budžet za test prihvatanja, kao i terminski plan. Plan testiranja definiše obuhvatnost zahteva i redosled testiranja svojstava sistema. Trebalo bi da definišete rizike na proces testiranja, kao što su pad sistema i neodgovarajuće performanse. Ukažite i na način reagovanja na rizike.
3. **Priprema testova prihvatanja**: Na bazi postavljenih kriterijuma prihvatanja softvera, vrši se projektovanje testova prihvatanja s ciljem da se testiraju ovi funkcionalne i nefunkcionalne karakteristike sistema.
4. **Izvršenje testova prihvatanja**: Idealno, ovi testovi bi trebalo da se izvršavaju u radnom okruženju naručioca softvera. Međutim, to nije uvek mogućno (zbog ometanja redovnog poslovanja), te se prave posebna okruženja za ova testiranja. Zbog unačajnog u dela interakcija korisnika sa softverom, vrlo je teško automatizovati testova prihvatanja.
5. **Pregovaranje vezano za rezultate testiranja**: Vrlo često nisu svi testovi pozitivni, a utvrđuju se i pojedini problemi. U takvom slučaju, isporučilac softvera i naručilac softvera pregovaraju da bi utvrdili da li je sistem dovoljno dobar da bi se stavio u upotrebu. Takođe se dogovaraju kako i kada će isporučilac softvera da otkloni uočene nedostatke.
6. **Odbijanje/prihvatanje sistema**: Održava se sastanak isporučioca i naručioca sistema da bi se odlučilo da li će sistem biti prihvaćen ili odbijen. Ako sistem nije dovoljno dobar za upotrebu, dogovara se dodatni razvoj da bi se uklonili uočeni nedostaci. Kada se taj razvoj završi, testovi prihvatanja se ponavljaju.



Slika 8.1 Proces testiranja radi prihvatanja softverskog sistema

TEST PRIHVTANJA KOD AGILNIH METODA RAZVOJA SISTEMA

U slučaju primene agilnog metoda u razvoju softvera, nema posebne aktivnosti kojom se vrši testiranje sistema radi njegovog prihvatanja od strane korisnika.

Pri primeni agilnih metoda, korisnici su i deo razvojnog tima, te obezbeđuju sistemske zahteve u vidu scenarija. Oni su odgovorni da definišu testove koji odlučuju da li razvijeni softver podržava definisan scenario (korisničku priču). Testovi su automatski i razvoj se ne nastavlja dok se ne prođu testovi prihvatanja. Nema posebne aktivnosti za testove prihvatanja.

Rizik kod testova prihvatanja je da predstavnik korisnika sistema nije tipičan, te ne odražava realno se potrebe korisnika. Takođe, automatizacija testova ograničava testiranje interaktivnih sistema. Za njihovo ispitivanje, potrebno je formirati grupe krajnjih korisnika sistema da bi ga koristili u svom svakodnevnom radu.

Ako test prihvatanja nije u celini pozitivan, to ne znači uvek da će biti odbijen i vraćen na doradu a njegova primena odložena. Mnogi korisnici imaju potrebu za softverom, uradili su sve pripreme, i analize da je veća šteta odložiti primenu softvera od primene nedovoljno kvalitetnog softvera. Zato pregovaraju sa isporučiocem softvera i dogovaraju dodatni razvoj i isporuku nove verzije softvera (koja se takođe opet testira), a u međuvremenu, prihvataju da koriste i ovu, nepotpunu i samo uslovno prihvaćenu verziju softvera.

PRIMER KORISNIČKOG TESTIRANJA

Primer korisničkog testiranja prilikom prihvatanja sistema za upravljanje insulin pumpom.

Testiranje zahteva sistema prilikom prihvatanja sistema:

Testiranje zahteva za merenje nivoa šećera u krvi – sistem za upravljanje insulin pumpom

- Unesi zapis o nivou šećera u krvi korisnika.
- Proveri da sistem nije izbacio poruku upozorenja u toku merenja
- Prikaži nivo šećera u krvi korisniku
- Sačuvaj zapis o nivou šećera u krvi korisnika.
- Prikaži poruku korisniku da li želi ponovo da izvrši merenje . .
- Proveri da li sistem zahteva da korisnik odabere ponovno merenje ili izlazak iz sistema
- Omogući korisniku da izđe iz sistema

Alfa testiranje obuhvata angažovanje test menadžera i analitičara testova, koji su pisali i analizirali zahteve, u testiranju sistema kako bi odredili da li sistem odgovara postavljenim zahtevima. Testiraju se svi korisnički zahtevi a ne samo merenje nivoa šećera u krvi.

Beta testiranje grupa korisnika će obavljati testiranje sistema (budući korisnici sistema), kako bi se osiguralo da sistem odgovara njihovim potrebama. Moguće je podeliti korisnike na grupe koje će testirati određene delove sistema i tako detaljno proveriti svaki deo sistema.

Primer tabele za ulazne i izlazne kriterijume testiranja prihvatljivosti korisnika u slučaju uspešno završenog testiranja prihvatljivosti:

Faza	Trajanje u danima
Izrada plana testiranja	15 dana
Definisanje test slučajeva	7 dana
Dizajniranje testova	10 dana
Jedinično testiranje	30 dana
Integraciono testiranje	20 dana
Sistemsko testiranje	50 dana
Alfa testiranje	10 dana
Beta testiranje	25 dana

Slika 8.2 Ulagni i izlazni kriterijumi za testiranje prihvatljivosti korisnika

ZADACI ZA SAMOSTALAN RAD

Zadaci koji se odnose na testiranje prihvatanja sistema od strane korisnika.

- 1. Zadatak:** Osmisliti korisničko testiranje za sistem za merenje metereoloških podataka. Definisati ulazne i izlazne kriterijume za test prihvatljivosti i potrebno vreme za njihovo izvršavanje.
- 2. Zadatak:** Napisati testiranje zahteva za pet korisničkih zahteva kao u prethodnom primeru. Odabrat korisničke zahteve koji su postavljeni za glavne funkcionalnosti sistema i definisati tok testiranja.

✓ Poglavlje 9

Zaključak

ZAKLJUČAK

Glavne poruke

1. Testiranje može samo da pokaže greške programa. Ne može da ukaže da program nema grešaka, jer možda one nisu otkrivene.
2. Testiranje softvera u razvoju je odgovornost tima koji razvija softver. Poseban tim bi trebalo da bude odgovoran za testiranje sistema pre nego što se isporuči kupcu. U procesu korisničkog testiranja, kupci ili korisnici sistema obezbeđuju podatke za testiranje i proveravaju da li je test uspešan.
3. Testiranje softvera u razvoju uključuje: testiranje jedinica, kada testirate pojedine objekte(klase) i metode; testiranje komponenata, kada testirate grupu objekata; i testiranje sistema, kada testirate deo ili kompletan sistem.
4. Pri testiranju softvera, trebalo bi da „odelite“ softver u skladu sa iskustvom i preporukama kako bi izabrali tipove slučajeva testiranja koji su se pokazali efektivnim u otkrivanju grešaka kod drugih sistema.
5. Uvek kada je mogućno, pišite automatske testove. Testovi su ugrađeni u program tako da se izvršavaju uvek kada se izvrši neka izmena u izvornom kodu softvera.
6. Razvoj vođen testom je pristup razvoju softvera u kome se testovi pišu pre koda koji treba da se testira. Prave se mali izmene koda i kod se doteruje sve dok svi testovi postanu uspešni.
7. Testiranje korišćenjem scenarija je korisno jer kopira praktičnu upotrebu sistema. Ono usavršava tipičan scenario korišćenja i to koristi za dobijanje slučajeva za testiranje.
8. Test prihvatanja je proces korisničkog testiranja u kome je cilj da se odluči da li je softver dovoljno dobar da se instalise kod korisnika i koristi u radnom okruženju.