# DIZAJN I IMPLEMENTACIJA

## PREDLOŠCI DIZAJNA

*Predložak* – je opis problema i suštine rješenja, kako bi se to rješenje moglo ponovno koristiti u različitim situacijama.

* Razvili su se iz ideje da postoje neki zajednički predlošci izrade dizajna koji su jako efektivni.

Naišli su na veliku primjenu kod razvoja objektno orijentiranog softvera.

* Neki od najpoznatijih predložaka su objašnjeni u literaturi.
* Design Patterns: Elements of reusable Object-Oriented Software”, E. Gamma, R. Helm, R. Johnson i J. Vlissides (tzv. “Gang of Four”) – Knjiga opisuje 23 klasična softverska uzorka dizajna uz primjere u C++ i Smalltalk-u.
* Najčešće se temelje na nasljeđivanju i polimorfizmu.

## OSNOVNI ELEMENTI

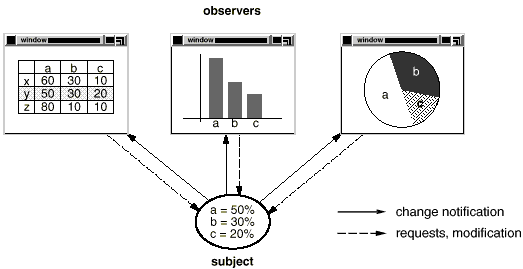
Četiri osnovna elementa svakog predloška dizajna:

1. *Naziv predloška* – smislena referenca na predložak.
2. *Opis predloška i problema* – objašnjava kada se predložak može primijeniti.
3. *Rješenje* – opis dijelova rješenja dizajna, njihovih veza i odgovornosti.

* Ne nudi opis dizajna nego daje samo predložak rješenja koji se zatim može realizirati na različite načine.
* Najčešće se prikazuje grafički i prikazuje veze među objektima i klasama objekata u rješenju.

1. *Posljedice* – opisuje rezultate i kompromise koji su rezultat primjene ovog predloška. Pomaže dizajnerima da shvate isplati li se koristiti predložak za rješavanje problema.

## VIŠESTRUKI PRIKAZI KORIŠTENJEM PREDLOŠKA “PREGLED”

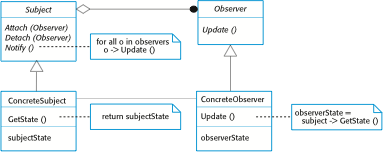


## PRIMJER PREDLOŠKA DIZAJNA: „PREGLED“

|  |  |
| --- | --- |
| **Naziv predloška** | **Pregled** |
| **Opis** | Predložak odvaja prikaz stanja objekta od samog objekta i dozvoljava korištenje dodatnih pregleda.  Kada se promjeni stanje objekta, svi pregledi se automatski obavještavaju i osvježavaju kako bi prikazali promjenu. |
| **Opis problema** | U mnogim situacijama je potrebno prikazati neku informaciju na različite načine (tablica, graf, …). Pri tome svi alternativni načini trebaju podržavati interakciju i kada se stanje promjeni svi se prikazi automatski osvježavaju.  Ovaj predložak se može koristiti u svim situacijama gdje se podatak o stanju prikazuje na više načina, a objekt koji sadrži to stanje nema informaciju o tome koji se formati prikaza koriste. |

|  |  |
| --- | --- |
| **Naziv predloška** | **Pregled** |
| **Rješenje** | Ovo uključuje dva apstraktna objekta (Subject i Observer) i dva konkretna objekta (ConcreteSubject i ConcreteObject). Konkretni objekti nasljeđuju atribute odgovarajućih apstraktnih objekata.  Apstraktni objekti uključuju generalne operacije koje se mogu primijeniti u svim situacijama.  Stanje koje se prikazuje se čuva u ConcreteSubject, koji nasljeđuje operacije od Subject, što dozvoljava dodavanje i uklanjanje Observer-a (svaki Observer odgovara nekom prikazu) i slanje upozorenja kada dođe do promjene stanja.  ConcreteObserver čuva kopiju stanja ConcreteSubject-a i automatski osvježava prikaz stanja kada dođe do promjene. |
| **Posljedice** | Subject je svjestan jedino Observer-a i nema veze s detaljima konkretnih klasa, što znači da su ti objekti minimalno povezani. Zbog toga, optimizacije koje bi poboljšale performanse nisu praktične. |

## UML MODEL PREDLOŠKA “PREGLED”



## PROBLEMI DIZAJNA

Za korištenje obrazaca u svom dizajnu, morate prepoznati da bilo koji problem dizajna s kojim ste suočeni, može imati povezane obrasce koji se mogu primjeniti.

* Navesti nekoliko objekata koje je stanje nekog drugog objekta promijenio (Observer pattern).
* Razraditi sučelja za niz povezanih objekata koji su često bili razvijeni postupno (Façade pattern).
* Osigurati standardni način pristupa elemenata u kolekciju, bez obzira na način kako je kolekcija implementirana (Iterator pattern).
* Dopustiti mogućnost proširenja funkcionalnosti postojećih klasa pri izvođenju (Decorator pattern).

## KORIŠTENJE PREDLOŽAKA

Predlošci podržavaju ponovno korištenje, ali na jako visokom – logičkom nivou.

* Kada se ponovno koriste izvršne komponente ograničenje predstavljaju odluke donesene u fazi dizajna (to mogu biti korišteni algoritmi, objekti ili sučelja).
* Kada su te odluke u sukobu s nekim od zahtjeva sustava koji se razvija ne može ga se iskoristiti.

Korištenje predložaka zapravo predstavlja ponovno korištenje ideje čija se implementacija može prilagoditi implementaciji sustava koji se razvija.

Na početku dizajna može biti problem unaprijed znati hoće li biti moguće korištenje nekog od predložaka.

* Korištenje uzoraka u fazi dizajna najčešće se javlja tek kada se jave problemi kod razvoja dizajna nakon čega se prepozna da je to moguće riješiti određenim predloškom.

## IMPLEMENTACIJA

Implementacija sustava je faza u kojoj se stvara izvršna verzija programa.

Aspekti implementacije koji su važni u programskom inženjerstvu, a ne spominju se u knjigama o programiranju:

* *Ponovno korištenje (eng. reuse)* – Većina modernog softvera se izrađuje ponovnim korištenjem postojećih komponenti ili sustava. Kada se razvija softver trebalo bi koristiti što više postojećeg koda.
* *Upravljanje konfiguracijom (eng. Configuration management)* – Za vrijeme razvojnih procesa, treba voditi računa o velikom broju različitih verzija svake komponente sustava koji se razvija, za što se koristi sustav za upravljanje konfiguracijom.
* *“Host - target” razvoj* - Najčešće se softver razvija na jednom sustavu (tzv. host), dok se izvršava na nekom drugom (tzv. target). Ova dva sustava mogu i ne moraju biti istog tipa.

## PONOVNO KORIŠTENJE

Do 1990 u razvoj novog softvera se kretalo iz početka, pisanjem cijelog koda u nekom od programskih jezika.

* Ponovno korištenje se odnosilo na ponovno korištenje funkcija i objekata u bibliotekama programskih jezika.

Cijena i brzina isporuke su postale neprihvatljive za ovaj način rada pogotovo za komercijalne sustave i sustave bazirane na Internetu.

Jedan od načina da se riješe ti problemi je bio bazirati razvoj na ponovnom korištenju postojećeg softvera.

* Danas se on koristi za razvoj poslovnog i znanstvenog softvera.

Ponovno korištenje softvera je moguće na različitim nivoima:

* Apstraktnom nivou:
* Ponovno korištenje softvera na ovom nivou se ne radi direktno nego se ponovno koristi znanje u dizajnu softvera (predlošci dizajna i arhitekture).
* Objektnom nivou:
* Ponovno se koriste objekti iz postojećih biblioteka, umjesto da sami pišete kod. Kako bi se implementirao ovaj način potrebno je pronaći prikladne biblioteke i provjeriti nude li objekti i metode potrebnu funkcionalnost.
* Nivou komponenti:
* Komponente predstavljaju kolekciju objekata i klasa objekata koje pružaju određenu funkcionalnost. Najčešće je potrebno prilagoditi i proširiti komponentu vlastitim kodom.
* Sistemskom nivou:
* Na ovom nivou se ponovno koristi čitavi aplikacijski sustavi.

## \*TROŠAK PONOVNOG KORIŠTENJA SOFTVERA

* Vrijeme koje se provede u traženju postojećeg softvera koji se može iskoristiti, te procjeni zadovoljava li naše potrebe ili ne. Taj softver je potrebno testirati kako bi se provjerilo hoće li ispravno funkcionirati u traženoj okolini.
* U nekim slučajevima je potrebno kupiti taj softver, a za velike komercijalne sustave cijena i nije beznačajna.
* Trošak prilagodbe i konfiguriranja komponenti koje se ponovno koriste kako bi zadovoljile potrebe sustava koji se razvija.
* Trošak integracije ponovno korištenih elemenata kako bi radili zajedno (ukoliko se koristi softver iz različitih izvora) i s novim kodom koji smo dodali.

## UPRAVLJANJE KONFIGURACIJOM

Upravljanje konfiguracijom generalno označava proces upravljanja izmjenama softvera koji se razvija.

Tri osnovne aktivnosti koje takav softver može pružiti su:

1. ***Upravljanje verzijama*** – pruža podršku praćenju različitih verzija komponenti softvera i podržava koordinaciju rada nekoliko programera.
2. ***Integracija sustava*** – pruža podršku razvojnom timu kod definiranja koje verzije komponenata su korištene za kreiranje svake verzije sustava. Ovaj se opis onda koristi za automatsko stvaranje izvršne verzije sustava na način da se prevode i povezuju odgovarajuće komponente.
3. ***Praćenje problema***– se jedne strane omogućava korisnicima prijavu grešaka i drugih problema, a s druge omogućava svim članovima razvojnog tima da prate tko radi na kojem problemu i kada je riješen.

## HOST-TARGET” RAZVOJ

Većina softvera se razvija na jednom računalu (tzv. host), a izvršava se na nekom drugom (tzv. target).

Ukoliko se gleda još općenitije može se pričati o razvojnoj i izvršnoj platformi.

* Platforma nije samo hardver.
* Uključuje operacijski sustav kao i drugi neophodni softver (npr. sustav za upravljanje bazom podataka ili kod razvojne platforme nekakvu interaktivnu razvojnu okolinu).

Ukoliko razvojna i izvršna platforma imaju instaliran različit softver i/ili različite arhitekture moguća rješenja su:

* Softver se testira tako da se instalira na izvršnoj platformi;
* Na razvojnoj potrebno pokrenuti simulator izvršne platforme.
* Simulatori se najčešće koriste kod razvoja ugradbenih sustava za simuliranje:
* uređaja (npr. senzora)
* događaja na koje sustav reagira.
* Korištenje simulatora ubrzava razvojni proces, ali je njihov razvoj jako skup pa su dostupni samo za najpopularnije hardverske arhitekture.

## “OPEN SOURCE” RAZVOJ

* “Open source” razvoj je način razvoja softvera u kojem je izvorni kod softvera javno objavljen i dobrovoljci su pozvani da sudjeluju u razvojnim procesima.
* Potječe od Free Softver Foundation ([www.fsf.org](http://www.fsf.org/)) koja zastupa ideju da izvorni kod ne bi trebao biti privatno vlasništvo, nego dostupan korisnicima koji ga onda mogu pregledavati i mijenjati po želji.
* Korištenje “open source” softvera je dobilo na važnosti korištenjem Interneta, kao medija za regrutaciju većeg broja dobrovoljaca (većina njih su istovremeno i korisnici koda).
* U teoriji, bilo koji sudionik “open source” projekta može prijaviti i ispraviti greške, te predložiti nove funkcionalnosti.
* U praksi uspješni “open source” projekti su i dalje bazirani na osnovnom razvojnom timu koji kontrolira izmjene softvera.

## “OPEN SOURCE” SUSTAVI

Najpoznatiji “open source” sustavi su:

* Linux operacijski sustav
* Java
* Apache web server
* mySQL baza podataka

“Open source” razvoj podržavaju IBM i Sun, te na njemu baziraju svoje produkte.

Najčešće se “open source” softver jednostavno preuzima bez naplate, s tim da se plaćaju dokumentacija i podrška.

Važna prednost korištenja “open source” softvera je što su tzv. zreli “open source” sustavi vrlo pouzdani.

## SPORNA PITANJA “OPEN SOURCE-A”

Za firmu koja je uključena u razvoj softvera postoje dva sporna pitanja povezana s korištenjem “open source” rješenja:

* Hoće li produkt koji se razvija koristiti “open source” rješenja?
* Hoće li “open source” pristup biti korišten za razvoj softvera?

Odgovor na ova pitanja najčešće ovisi o vrsti softvera koji se razvija:

* Ukoliko budući softver spada u domenu u kojoj postoje kvalitetna “open source” rješenja, štedi se vrijeme i novac njihovim korištenjem.
* Ukoliko se razvija softver za specifične organizacijske zahtjeve ili ga je potrebno integrirati s komponentama koje ne podržavaju “open source” rješenja korištenje “open source-a” možda neće biti moguće.

## POSLOVANJE S “OPEN SOURCE” RJEŠENJIMA

* Sve više kompanija koje se bave razvojem softvera koristi “open source” pristup razvoju.
* Njihova poslovna strategija se ne zasniva na prodaji softvera nego na prodaji podrške za taj produkt.
* Ideja je da uključivanje “open source” zajednice će omogućiti jeftiniji i brži razvoj softvera, te ujedno stvoriti buduće korisnike.

Ovo se jedino može primijeniti za općenitije produkte.

## LICENCIRANJE “OPEN SOURCE-A”

Većina “open source” licenci je izvedena od jednog od slijedećih modela:

* *The GNU General Public Licence (GPL)* – tzv. licenca reciprociteta koja označava da ukoliko se koristi softver s ovom licencom tada rezultanti softver također treba biti “open source”.
* *The GNU Lesser General Public License (LGPL)* – varijanta GPL licence koja dozvoljava da se u aplikaciji koriste “open source” komponente, bez da sama aplikacija bude “open source”, ali se sve izmjene u “open source” komponentama moraju objaviti.
* *The Berkley Standard Distribution (BSD) License*– korisnik ovako licenciranog “open source” koda ne mora objaviti nikakve izmjene na “open source” kodu, nema obavezu javne objave svog koda.

# UBRZANI RAZVOJ SOFTVERA

## UBRZANI RAZVOJ SOFTVERA

* Zbog velikih promjena poslovne okoline potrebno je brzo reagirati na nove prilike.
* To zahtjeva brz razvoj i isporuku potrebnog softvera, pri čemu je najčešće brzina isporuke jedan od najvažnijih uvjeta.
* Klijenti su spremni prihvatiti softver “niže kvalitete” ukoliko je moguće važne funkcionalnosti isporučiti na vrijeme.

## OSNOVNE KARAKTERISTIKE

Svrha ubrzanog razvoja softvera je što brža produkcija korisnog softvera.

Iako prostoje različiti pristupi, svi oni dijele neke zajedničke karakteristike:

* Softver se razvija kroz niz inkrementa, od kojih svaki dodaje nove funkcionalnosti. Korisnik procjenjuje svaki inkrement i daje prijedloge za slijedeće inkremente.
* To su iterativni procesi u kojim se isprepliću specifikacija, dizajn i razvoj i sve tri faze se rade istovremeno.
* Dokumentacija je minimizirana ili je programska okolina koja se koristi za implementaciju sustava automatski generira.
* Ukoliko postoji Specifikacija zahtjeva definira samo najvažnije karakteristike sustava.
* Korisnička sučelja se razvijaju



korištenjem interaktivnog razvojnog

sustava.

## ITERATIVNI RAZVOJNI PROCES

## PREDNOSTI UBRZANOG RAZVOJA

* ***Ubrzana isporuka važnih korisničkih zahtjeva***– svaki inkrement isporučuje funkcionalnosti s najvećim prioritetom.
* ***Angažiranost korisnika*** – Korisnici moraju biti uključeni u razvoj pa je vjerojatnost da će taj sustav odgovarati njihovim zahtjevima puno veća.

## NEDOSTACI

* *Problem menadžmenta* – teško je ocijeniti napredak i moguće probleme, s obzirom da nema dokumentacije s kojom je moguće usporediti tijek rada.
* *Problem ugovora* – bez specifikacije teško je dogovoriti cijenu projekta:
* Naručitelj možda neće biti spreman platiti po broju radnih sati.
* Razvojni tim možda neće biti spreman prihvatiti fiksnu cijenu ugovora bez da su poznati svi zahtjevi.
  + *Problem validacije* – što se koristi za provjeru sustava ukoliko nema specifikacije.
  + *Problem održavanja* – Stalne izmjene koda narušavaju strukturu sustava i zahtijevaju izmjene kako bi se ostvarili zahtjevi korisnika.

## AGILNE METODE

Agilne metode obuhvaćaju cijeli niz metoda razvoja softvera, a baziraju se na iterativnim i inkrementnom razvoju s ciljem što brže isporuke traženog softvera.

Osnovne karakteristike:

* Fokus je na kodiranju, a ne dizajnu i pisanju dokumentacije;
* Baziraju se na iterativnom pristupu razvoja softvera;
* Namijenjene su brzoj isporuci funkcionalnog softvera, te brzim izmjenama postojećeg kako bi se prilagodio zahtjevima tržišta.

Agilne metode su najprikladnije za male/srednje velike poslovne sustave.

* Nisu pogodne za razvoj: velikih sustava (razvojni timovi su dislocirani), ukoliko postoji složena veza s drugim softverskim ili hardverskim sustavima, kritičnih sustava.

## PRINCIPI AGILNIH METODA

|  |  |
| --- | --- |
| **Princip** | **Opis** |
| **Uključenost korisnika** | Korisnik bi trebao biti uključen u cijeli proces razvoja softvera. Njegova je uloga zadavanje novih zahtjeva, postavljanje njihovog prioriteta i procjene gotovih iteracija sustava. |
| **Inkrementne isporuke** | Softver se isporučuje u inkrementima, pri čemu korisnik specificira koji će zahtjevi biti uključeni u pojedini inkrement. |
| **Ljudi umjesto procesa** | Koriste se vještine ljudi u razvojnom timu, procesi nisu unaprijed zadani nego razvojni tim sam definira svoj način rada. |
| **Prihvaćanje promjena** | Očekuje se da će se zahtjevi sustava promijeniti i on se dizajnira tako da im se može prilagoditi. |
| **Jednostavnost** | Važno je održati softver koji se razvija što jednostavnijim, kao i razvojni proces koji se koristi. Kada je god moguće potrebno je uklanjati složene dijelove iz sustava. |

## PROBLEMI S AGILNIM METODAMA

* Može biti teško zadržati interes korisnika koji uključen u proces.
* Osobine članova tima ne moraju biti prikladne za grupni rad koji podrazumijevaju agilne metode.
* Ukoliko ima više zainteresiranih strana može biti teško odrediti prioritet pojedinih zahtjeva.
* Održavanje jednostavnosti može zahtijevati dodatni rad, što nekada nije jednostavno zbog rokova.
* Problem je sastavljanje ugovora kao kod drugih inkrementnih metoda.

## EKSTREMNO PROGRAMIRANJE (XP)

* Eng. Extreme programming - XP
* Najpoznatija i najkorištenija od svih agilnih metoda.
* Dobila je ime po tome jer forsira iterativan pristup i uključenost korisnika do krajnjih granica:
* Nove verzije se mogu isporučivati nekoliko puta dnevno;
* Inkrementi se isporučuju korisniku svaka 2 tjedna;
* Programeri rade u paru;
* Prvo se razvija test pa se tek onda kodira;
* Svi testovi se ponovno pokreću za svaku verziju i ona se prihvaća tek kada se svi izvrše uspješno.

## XP - SCENARIJI ZAHTJEVA

U XP-u korisnički zahtjevi se prikazuju u obliku scenarija ili korisničkih priča.

Svaki scenarij je zapisan na kartici i razvojni tim ga dijeli na pojedinačne zadatke za implementaciju.

* Zadaci su osnova za planiranje radnih sati i rasporeda, te za naplatu.
* Korisnici biraju koji će se scenariji uključiti u sljedeću verziju softvera, ovisno o njihovom prioritetu i procjeni rasporeda.

## PRIMJER – LIBSYS SUSTAV

Scenarij za preuzimanje i ispis članka



Kartice zadataka - učitavanje dokumenata



## PROCES OBJAVE VERZIJA KOD XP-A



## XP U PRAKSI

|  |  |
| --- | --- |
|  |  |
| **Inkrementno planiranje** | Zahtjevi su opisani u scenarijima, a što će biti uključeno u određenu verziju ovisi o dostupnom vremenu i prioritetu zahtjeva. Razvojni tim razdvaja scenarije u razvojne zadatke. |
| **Male verzije za objavu** | Minimalni korisni niz funkcionalnosti koji pruža nekakvu poslovnu vrijednost se prvo razvija. Često se izdaju nove verzije i inkrementalno se dodaju nove funkcionalnosti. |
| **Jednostavan dizajn** | Jednostavan dizajn koji je neophodan za postizanje trenutnih zahtjeva. |
| **Razvoj baziran na testiranju** | Koristi se razvojna okolina koja podržava jedinično testiranje i testovi se pišu prije implementacije određene funkcionalnosti. |
| **Procjenjivanje** | Cijeli razvojni tim bi trebao cijelo vrijeme procjenjivati kod i mijenjati ga kako bi ostao što jednostavniji i lakši za održavanje. |

|  |  |
| --- | --- |
|  |  |
| **Programiranje u paru** | Članovi razvojnog tima rade u paru, provjeravaju jedan drugog i pružaju podršku. |
| **Zajedničko vlasništvo** | Članovi razvojnog tima rade na svim dijelovima sustava, svi poznaju cijeli kod – svi mogu raditi izmjene. |
| **Kontinuirane integracije** | Čim se završi rad na nekom zadatku, on se integrira u sustav. Nakon svake integracije pokreću se svi jedinični testovi na cijelom sustavu. |
| **Održiv tempo** | Velik broj prekovremenih sati nije prihvatljiv jer kao rezultat dolazi do smanjenja kvalitete rada i produktivnosti. |
| **Prisutnost korisnika** | Predstavnik krajnjeg korisnika bi trebao postati član razvojnog tima, a njegova je uloga opis zahtjeva za implementaciju. |

## XP I AGILNI PRINCIPI

* **Inkrementni razvoj -** Podržan kroz brojne, male nove verzije sustava.
* **Uključenost korisnika -** Korisnik je član razvojnog tima.
* **Ljudi a ne procesi -** Programiranje u paru, kolektivno poznavanje koda i procesi koji podržavaju izbjegavanje velikog broja prekovremenih.
* **Prihvaćanje promjena -** Promjene podržavaju česte objave novih verzija.
* **Jednostavnost -** Konstantno procjenjivanje i izmjene koda.

## XP I PROMJENE

Dobra praksa u programskom inženjerstvu je dizajnirati sustav da omogući promjene.

* Isplati se utrošiti vrijeme i trud kako bi se predvidjele promjene jer to štedi vrijeme (novac) u kasnijim fazama životnog ciklusa softvera.

Kod XP-a se smatra da se ne isplati trošiti vrijeme na predviđanje promjene jer se to i onako ne može dovoljno kvalitetno napraviti.

* Radije, on predlaže konstantne izmjene koda kako bi promjene bile što jednostavnije kada ih bude trebalo implementirati.

## TESTIRANJE U XP-U

Kako bi izbjegao probleme testiranja i validacije sustava, za razliku od ostalih agilnih metoda XP postavlja veći naglasak na testiranje.

Osnovne karakteristike testiranja u XP-u su:

* “Test-first” razvoj;
* Inkrementni razvoj testova iz scenarija;
* Sudjelovanje korisnika u razvoju i validaciji testova;
* Korištenje automatiziranih okolina koje svaki put kada se pokrene testiranje provodi sve do tada napravljene testove.

## PRIMJER – LIBSYS SUSTAV

Opis testa



## “TEST-FIRST” RAZVOJ

* Pisanje testova prije samog koda poboljšava razumijevanje zahtjeva koji će se implementirati.
* Testovi su napravljeni kao programi, pa se mogu izvršavati automatski.
* Kada se doda nova funkcionalnost, svi prethodni i novi testovi se automatski izvršavaju i na taj se način provjerava da dodavanje nove funkcionalnosti nije uzrokovalo grešku u prethodnim dijelovima koda.

## PROBLEMI KOD “TEST-FIRST” RAZVOJA

* Programeri preferiraju pisanje koda, a ne testova pa se ponekad pišu nepotpuni testovi koji ne provjeravaju neke iznimne situacije.
* Ponekad je teško napraviti testove (npr. kod složenih korisničkih sučelja problem je napraviti jedinične testove za dio koda koji se odnosi na prikaz na ekranu, kao i za prebacivanje između različitih formi u aplikaciji).
* Problem je ocijeniti u kolikoj mjeri testovi pokrivaju aplikaciju.
* Korisnik najčešće ne sudjeluje cijelo vrijeme u radu s razvojnim timom.

## PROGRAMIRANJE U PARU

Kod XP-a programeri sjede zajedno i rade na istom dijelu koda.

Parovi se formiraju dinamički, tako da svi članovi tima modu raditi jedni s drugima u paru za vrijeme trajanja projekta.

Prednosti:

* Svi članovi tima su upoznati s cijelim kodom i rade na njegovom poboljšanju.
* Ovaj način rada se koristi kao neformalni proces provjere, s obzirom da s obzirom da svaku liniju koda provjerava više od jedne osobe, pa se otkrije veći broj grešaka.
* Podržava procjenjivanje koda .
* Mjerenja su pokazala da je programiranje u paru približno jednako produktivno kao i da dvoje ljudi radi neovisno na istom problemu.

## UBRZANI RAZVOJ APLIKACIJA

Iako su Agilne metode jako popularne, uz njih postoji čitav niz drugih pristupa ubrzanom razvoju softvera.

Tehnike za ubrzani razvoj aplikacija (eng. Rapid Application Development - RAD) su se razvile iz tzv. jezika četvrte generacije oko 1980 .

Usmjerene su ka proizvodnji interaktivnih poslovnih aplikacija baziranih na podacima.

* Najčešće su organizirane kao niz alata koji omogućava unos, pretraživanje i prikaz tih podataka iz baze.



## ALATI RAD OKOLINE

* *Programski jezik za pristup bazi podataka* – SQL – standardan jezik za programiranje baza podataka.
* *Generator sučelja* – koristi se za kreiranje formi za komunikaciju s korisnikom, unos podataka i pokretanje izvještaja.
* *Veze prema uredskim aplikacijama* – aplikacije poput MS Worda ili Excela koje se koriste za analizu i manipulaciju numeričkim informacijama ili stvaranje predložaka izvještaja.
* *Generiranje izvještaja* – Koristi se za definiranje i stvaranje izvještaja iz baze podataka.

## GENERIRANJE SUČELJA

Većina poslovnih aplikacija je bazirana na kompleksnim formama i njihov “ručni” razvoj bi predugo trajao.

Zbog toga RAD okoline podržavaju generiranje sučelja i pri tome uključuju:

* Jednostavno, interaktivno generiranje formi korištenjem vuci i ispusti (eng. drag and drop) tehnike.
* Povezivanje formi u slučajevima kada je potrebno prikazati različite forme (razvojni tim bira koje akcije uzrokuju pojavljivanje određenih formi).
* Provjeru vrijednosti koje se unose u pojedinim poljima forme (razvojni tim za svako polje može definirati koje se vrijednosti mogu unijeti).

Danas većina RAD okolina podržava transfer definiranog sučelja u verziju za web pretraživač.

## VIZUALNO PROGRAMIRANJE

Skriptni jezici poput Visual Basic-a podržavaju vizualno programiranje.

* Postoji velika biblioteka standardnih komponenti i one se mogu prilagođavati specifičnim zahtjevima.

Nedostatak:

* Problem je koordinirati timski razvoj.
* Ne postoji jasna arhitektura sustava.
* Implementacija nestandardnih korisničkih sučelja može biti problem.

# TESTIRANJE SOFTVERA

## TESTIRANJE PROGRAMA

Testiranje je proces koji dokazuje da program radi ono za što je namijenjen, te pomaže u otkrivanju propusta ili grešaka, koji su napravljeni u razvoju, prije nego se krene s korištenjem programa.

* Testovi se provode s posebnim tzv. testnim setom podataka.
* U rezultatima testnog pokretanja programa se zatim traže greške, anomalije ili informacije o nefunkcionalnim atributima.

***Test otkriva postojanje greške, a ne da je NEMA.***

Testiranje je dio općenitijeg procesa verifikacije i validacije softvera.

## CILJEVI TESTIRANJA PROGRAMA

Testni procesi imaju dva važna cilja:

1. Demonstrirati i razvojnom timu i korisnicima da softver odgovara zahtjevima

* Za softver po narudžbi bi trebao biti bar jedan test za svaki zahtjev u Specifikaciji zahtjeva.
* Za generički softver to znači da bi trebao napraviti testove za svaku od mogućnosti tog softvera, za svaku njihovu kombinaciju koja će biti uključena u verziju za objavu.

1. Otkriti u kojim situacijama se softver ne ponaša ispravno ili na način koji ne odgovara specifikaciji.

* Testiranje grešaka (eng. defect testing) se koristi za pronalaženje grešaka u smislu neželjenog ponašanja sustava (pad sustava), neželjenih interakcija s drugim sustavima, neispravnih proračuna te narušavanja integriteta podataka.

## VALIDACIJA SUSTAVA I TESTIRANJE GREŠAKA

* Prvi cilj rezultira *validacijom sustava*
* Očekuje se da se sustav ponaša ispravno za zadani niz testova koji prikazuju očekivano korištenje sustava.
* Demonstrira članovima razvojnog tima i korisnicima da softver odgovara zahtjevima.
* Uspješan test pokazuje da sustav radi kako bi trebao.
  + Drugi cilj rezultira *testiranjem grešaka*
  + Testovi su dizajnirani tako da otkriju greške zbog čega se namjerno rade nejasni testovi koji ne trebaju prikazati normalno korištenje sustava.
  + Cilj je otkrivanje grešaka u softveru, tj. gdje se softver ne ponaša ispravno ili nije u skladu sa specifikacijom.



## ULAZNO IZLAZNI MODEL TESTIRANJA

Sustav se promatra kao crna kutija (tzv. Funkcijsko

testiranje ili eng. black-box testing.)

## VERIFIKACIJA I VALIDACIJA

Testiranje je u biti samo dio jedno šireg procesa koji se naziva verifikacija i validacija (V&V).

***Verifikacija*: "Radimo li produkt ispravno?"**

* Provjerava odgovara li softver specifikaciji zahtjeva, tj. funkcionalnim i nefunkcionalnim zahtjevima.

***Validacija*: "Radimo li ispravan produkt?"**

* Provjerava odgovara li softver stvarnim potrebama korisnika.

Konačni cilj procesa V&V je potvrditi da je softver dovoljno dobar i da služi svojoj svrsi.

Pri tome ne mora biti u potpunosti bez grešaka, a stupanj tolerancije na greške ovisi o:

* ***Funkciji softvera***– što je softver kritičnije to je važnije da bude pouzdan i stupanj tolerancije na greške je manji.
* ***Očekivanju korisnika***– korisnici mogu imati niska očekivanja od određene vrste softvera zbog prethodnih iskustava. Nakon instalacije softvera korisnici mogu tolerirati padove sustava zbog pogodnosti koji im taj softver donosi i u tim situacijama nije potrebno provesti opsežno testiranje, ali kako softver sazrijeva potrebno je proširiti opseg testova.
* ***Tržišnim uvjetima***– objava produkta na tržištu može biti važnija nego pronalaženje grešaka (objava verzije prije konkurencije, cijena koju su korisnici spremni platiti, datum isporuke).

## INSPEKCIJE I TESTIRANJA

Procesi V&V uključuju:

* *Testiranje softvera* – dinamička provjera ponašanja sustava koja promatra ponašanje sustava.
* Sustav se izvršava s testnim podacima i promatra se njegov rad.
  + - *Inspekcije i kontrole softvera* – statička provjera ponašanja sustava koja se bavi analizom statičke prezentacije sustava kako bi se otkrili problemi.



* + - Korištenjem alata za analizu koda.

Inspekcije i testiranje podržavaju

V&V u različitim fazama

softverskih procesa.

* Inspekcije softvera se uglavnom fokusiraju na provjeravanje izvornog koda s ciljem otkrivanja anomalija i grešaka.
* Uz to inspekcije se još fokusirati na bilo koji čitljivi dio kao što su specifikacija zahtjeva, dizajna, konfiguraciju, testne podatke, …
* U praksi se pokazalo da su inspekcije efektivna tehnika za otkrivanje grešaka.

## PREDNOSTI INSPEKCIJA U ODNOSU NA TESTIRANJE

* Za vrijeme testiranja neke greške mogu prikriti druge greške. S obzirom da su inspekcije statički procesi ne treba voditi računa o interakciji među greškama i u jednoj inspekciji je moguće pronaći više grešaka.
* Nedovršene verzije sustava se mogu provjeravati bez dodatnih troškova . U slučajevima kada program nije dovršen potrebno je razviti specijalne testne okoline kako bi se testirali dostupni dijelovi.
* Osim što inspekcija pronalazi greške u programu, može se promatrati i širi aspekti programa kao kompatibilnost sa standardima, prenosivost i održavanje.

## INSPEKCIJE I TESTIRANJE

Inspekcije se koriste jako dugo vremena i postoje nekakve studije i eksperimenti koji pokazuju da su inspekcije efikasnije u pronalaženju grešaka od testiranja.

* 1986 Fagan – više od 60% grešaka može otkriti u programu korištenjem neformalne inspekcije.
* 1999 Prowell at al. – više od 90% grešaka se može otkriti inspekcijama.

Inspekcije i testiranje su komplementarne tehnike provjere ispravnosti sustava i obje se koriste za V&V sustava.

* Inspekcije se ne mogu koristiti za otkrivanje grešaka koje se javljaju zbog neočekivanih interakcija različitih dijelova programa, vremenske usklađenosti komponenti, performansi sustava.

## MODEL TESTNOG PROCESA



* Testni primjeri (eng. test cases) – specificiraju ulazne vrijednosti u testove, očekivani rezultat testa (izlazne vrijednosti), te opis što se testira.
* Testni podaci – ulazni parametri osmišljeni tako da testiraju sustav. U nekim slučajevima se mogu generirati automatski ali češće su u to uključeni ljudi koji razumiju što bi sustav trebao raditi.
* Izvršavanje testova može biti automatsko i očekivani rezultati se automatski uspoređuju s testnim rezultatima .

## FAZE TESTIRANJA

Komercijalni softverski sustavi tipično prolaze kroz tri faze testiranja:

1. ***Razvojno testiranje***– sustav se testira u fazi razvoja kako bi se otkrile greške i nepravilnosti, a u testiranju sudjeluju dizajneri i programeri.
2. ***Testiranje verzija za objavu***(eng. release testing) – odvojeni testni timovi testiraju gotove verzije sustava prije nego se predaju korisnicima. Cilj je provjera da sustav odgovara zahtjevima zainteresiranih strana.
3. ***Korisničko testiranje***– korisnici ili potencijalni korisnici testiraju sustav u svojoj okolini.

## *RAZVOJNO TESTIRANJE*

* Razvojno testiranje uključuje sve testne aktivnosti koje provodi razvojni tim za vrijeme izrade sustava.
* Test najčešće provodi programer koji radi na tom dijelu koda, ili u nekim razvojnim procesima se koriste parovi programer/tester gdje svaki programer radi u suradnji s jednim testerom koji razvija testove i pomaže u testnim procesima.
* Kod kritičnih sustava koristi se još formalniji pristup koji podrazumijeva postojanje odvojene grupe za testiranje unutar razvojnog tima.
* Za vrijeme razvoja testiranje se može provesti na tri nivoa:
* *Jedinično testiranje* (eng. unit testing) – testiraju se pojedinačne cjeline programa ili klase objekata. Fokus je postavljen na provjeru funkcionalnosti objekata ili metoda.
* *Testiranje komponenti* – nekoliko pojedinačnih cjelina se integrira s ciljem stvaranja komponenti. Cilj je testiranje sučelja komponenti.
* *Testiranje sustava* – neke ili sve komponente sustava se integriraju u sustav i on se testira kao cjelina. Testiranje se fokusira na interakciju komponenti.

## *Jedinično testiranje*

Jedinično testiranje je proces izoliranog testiranja pojedinačnih komponenti programa.

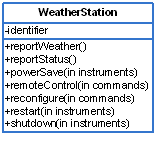
Jedinice mogu biti:

* Pojedinačne funkcije ili metode unutar objekta.
* Jedinični testovi pozivaju ove funkcije / metode s različitim ulaznim parametrima i provjeravaju slažu li se stvaran i očekivani rezultat.
* Klase objekata s nekoliko atributa i metoda.
* Komponente s definiranim sučeljem koje se koristi za pristup njihovim funkcionalnostima.

## *Testiranje klasa objekata*

U ovom slučaju test treba kreirati tako da u potpunosti pokriva sva svojstva objekta:

* testirati sve operacije povezane s objektom;
* postaviti i provjeriti vrijednosti svih atributa povezanih s objektom;
* postaviti objekt u sva moguća stanja, što znači da treba simulirati sve događaje koji uzrokuju promjenu stanja.



Nasljeđivanje otežava testiranje klasa objekata jer informacije koje treba testirati nisu lokalizirane.

Primjer meteorološke stanice: 🡪

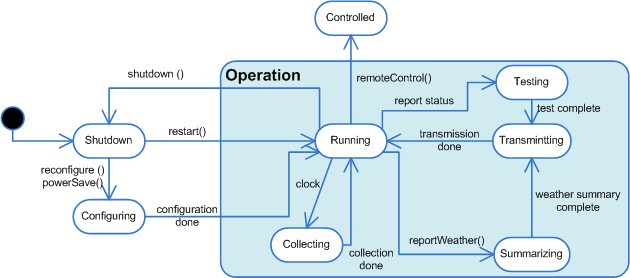
Ovaj objekt ima samo jedan atribut: *identifier*.

* To je konstanta koja se postavlja kod inicijalizacije stanice.
* Test jedino treba provjeriti je li on postavljen.

Objekt ima 7 metoda i sve se trebaju testirati.

* Idealno bi ih bilo testirati izolirano, ali u nekim slučajevima slijed testova je neophodan.

## *Testiranje meteorološke stanice*



Za testiranje stanja objekata pogodno je koristiti dijagram stanja, koji identificira slijed prijelaza stanja koji treba testirati, te slijed događaja koji uzrokuju ove tranzicije.

Za primjer mete. stanice trebalo bi testirati slijedeće sljedove događaja:

* + Shutdown -> Running-> Shutdown
  + Configuring-> Running-> Testing -> Transmitting -> Running
  + Running->Collecting->Running->Summarizing->Transmitting->Running

## *Automatizirano testiranje*

* Kada je god to moguće potrebno je automatizirati jedinično testiranje kako bi se obavljalo bez intervencije programera.
* Realizira se korištenjem automatizirane testne okoline (npr. JUnit) za pisanje i pokretanje testova.
* Testne okoline za jedinično testiranje sadrže općenite testne klase koje se zatim proširuju kako bi se stvorili konkretni testovi.
* Okoline pokreću sve implementirane testove i daju izvještaj o uspješnosti izvršavanja testova.

## *Komponente automatiziranih testova*

Automatizirani testovi se sastoje od tri dijela:

1. ***Inicijalizacijski dio*** – U sustav se inicijalizira s testnim podacima (definiraju se ulazne vrijednosti i očekivani rezultati testova).
2. ***Pozivni dio***– dio u kojem se poziva objekt ili metoda koji se testiraju.
3. ***Dio tvrdnje***(eng. assertion) – u kojem se uspoređuje rezultat poziva objekta ili metode s očekivanim rezultatom. Ukoliko je rezultat tvrdnje isitnit test je prošao, u suprotnom nije.

## *Primjer – jediničnog testiranja (JUnit )*

import junit.framework.TestCase;

public class CalculatorTest extends TestCase {

public void testAdd() {

// Testing if 2+3=5:

assertEquals (5, Calculator.add (2,3));

}

}

public class Calculator {

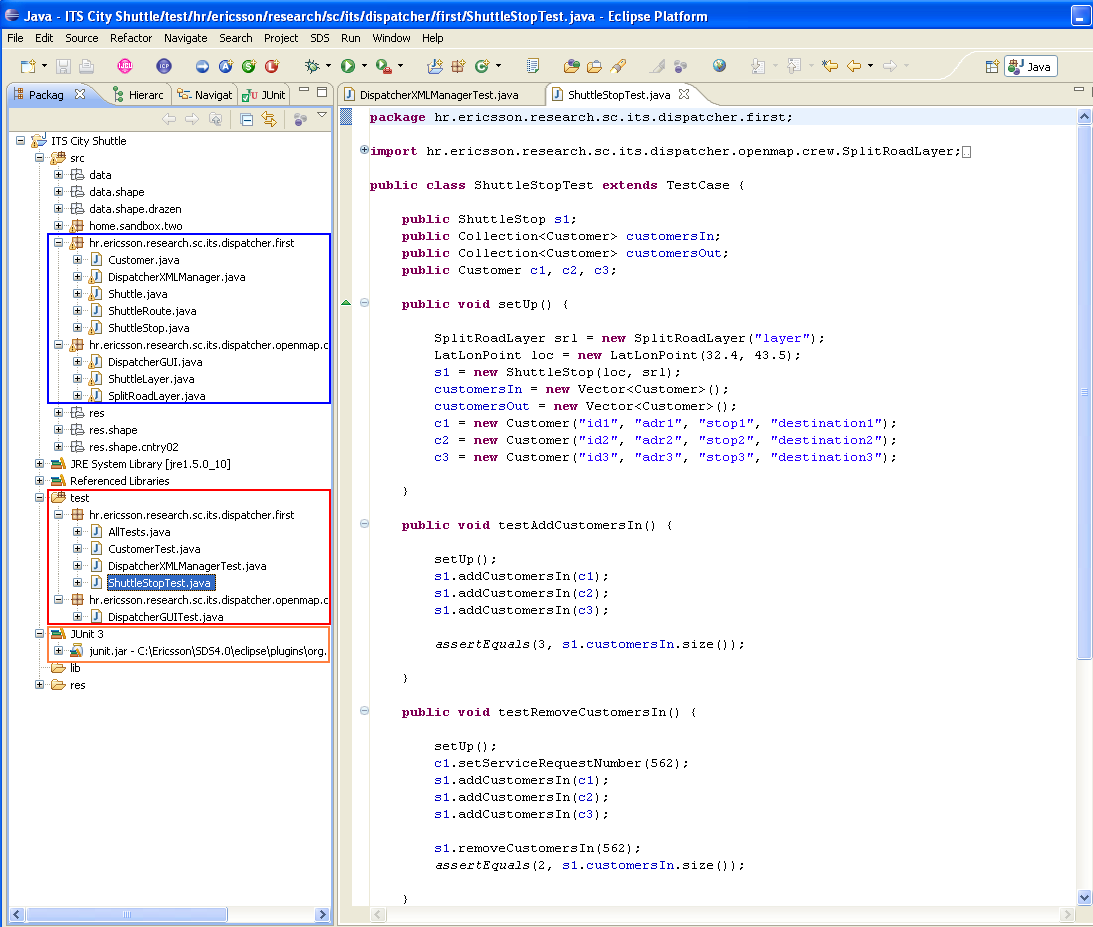
public int add (int a, int b){

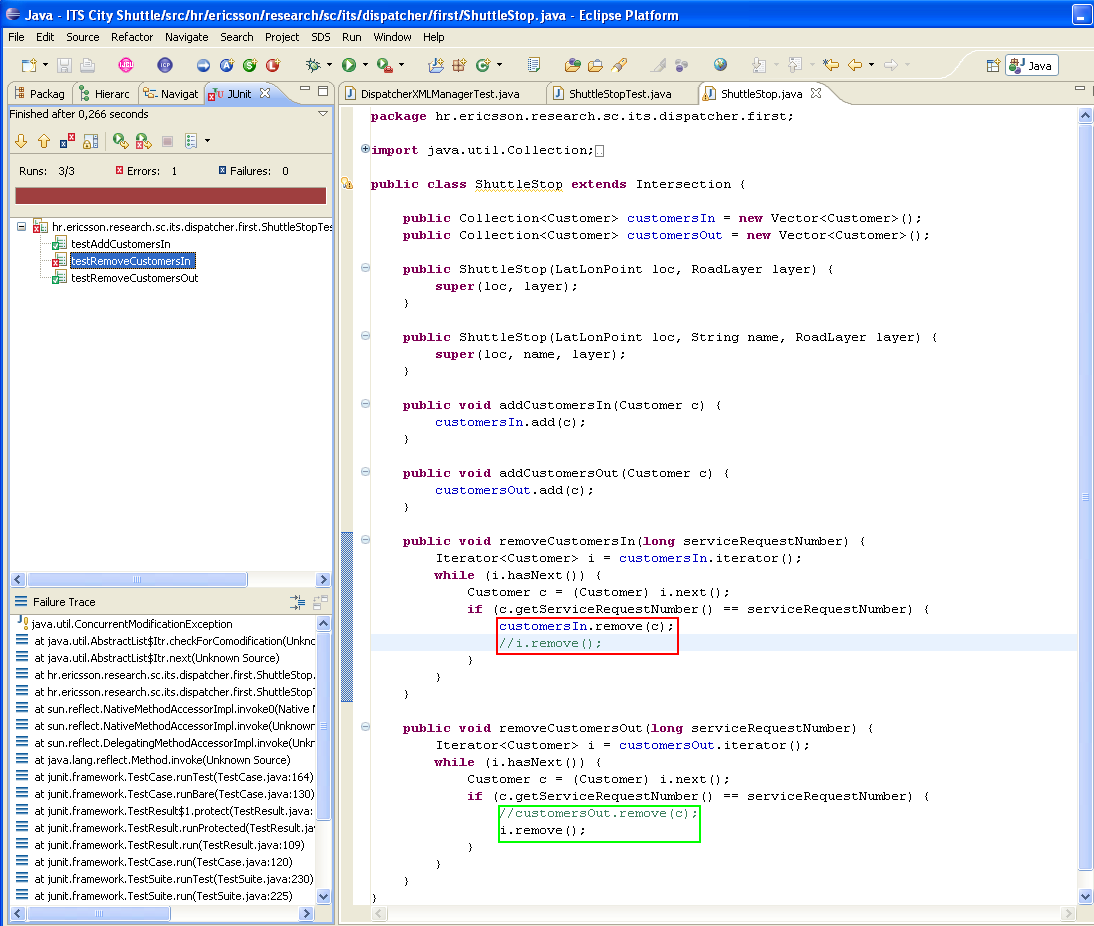
return a + b;

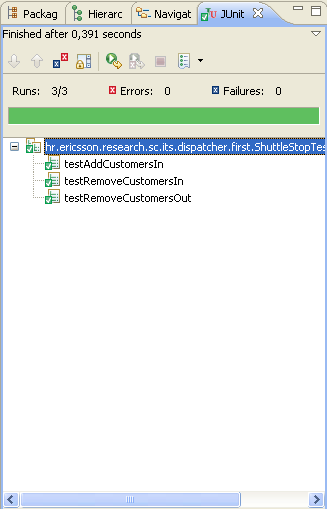
}

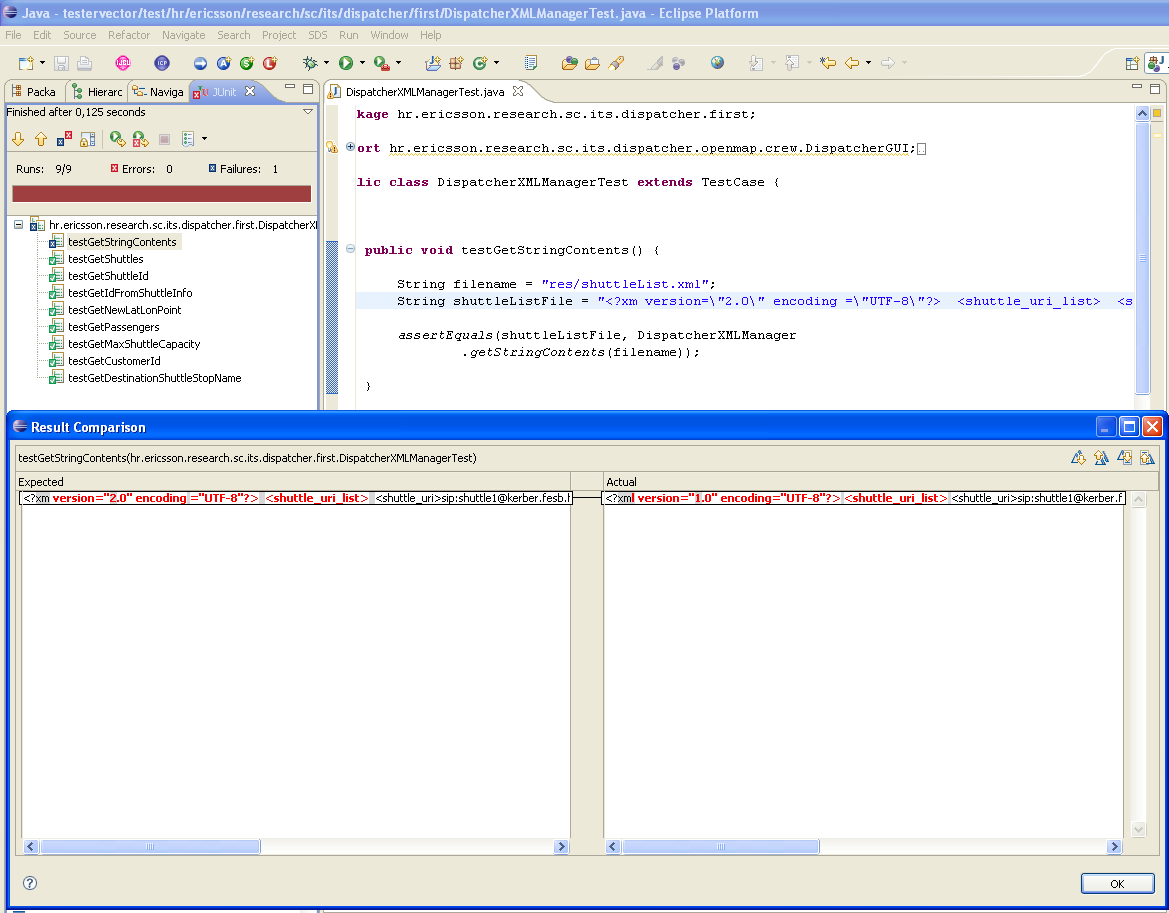
}

## *Primjer – jediničnog testiranja*









## *Vrste jediničnih testova*

Postoje dvije vrste jediničnih testova:

* ***Testovi koji provjeravaju normalno ponašanje sustava***– testovi pokazuju da komponenta koja se testira kada se koristi na ispravan način radi ono što se od nje očekuje (npr. unos podataka u bazu rezultira s novim redcima u odgovarajućim tablicama i da su svi podaci ispravno uneseni).
* ***Testovi koji provjeravaju problematično ponašanje sustava*** – testovi koriste neispravne ulazne podatke i provjeravaju vode li oni do pada komponente ili su ispravno obrađeni.

## *Testne strategije*

Postoje dvije osnovne testne strategije koje omogućavaju efikasno planiranje testiranja:

* ***Testiranje particija***– identificiraju se grupe ulaznih podataka koje imaju zajedničke karakteristike i svi elementi grupe se obrađuju na isti način.
* Izrađuju se testni primjeri za svaku grupu.
  + - ***Testiranje bazirano na*** *uputama* (eng. guideline)– odabir testova se radi prema uputama.
* Upute su nastale kao rezultat prethodnih iskustava s greškama koji se javljaju kod razvoja sustava.

## Testiranje particija

Vrste ulaznih podataka i rezultata aplikacije se često mogu podijeliti u određen broj grupa/klasa čiji su članovi slični.

* Grupe mogu biti: pozitivni brojevi, odabir iz izbornika, …
* Bitno je da se program najčešće slično ponaša za sve elemente određene grupe.
* *Npr. ukoliko program za proračun treba dva pozitivna broja, tada se očekuje da će se program ponašati isto za sve pozitivne brojeve.*

Zbog sličnog ponašanja ove se grupe nazivaju još i ekvivalentne particije ili domene u kojima se program ponaša na sličan način za sve članove grupe.

Kod sistematičnog pristupa razvoja testnih primjera potrebno je identificirati sve ulazne i izlazne particije za pojedinačne komponente i cijeli sustav i testovi se onda rade za svaku particiju.

##### Ekvivalentne particije



Primjer: Prema specifikaciji program prihvaća 4-10 ulaznih vrijednosti a svaka od njih ima 5 znamenki i veća je od 10000.

Ekvivalentne particije se mogu kreirati iz:

* Specifikacije sustava – tzv. "black-box testing" – ne treba imati nikakvo znanje o tome kako sustav radi.
* Koda sustava – tzv. "white-box testing" – testovi se stvaraju na osnovu koda.
* *Npr. ukoliko kod sadrži upravljanje iznimkama u slučaju neispravnog unosa, kreira se particija iznimki koja sadrži sve ulazne vrijednosti koje će izazvati pokretanje te iznimke.*

Korištenje ekvivalentnih particija je učinkovit pristup testiranju jer pomaže otkrivanje grešaka koje se često javljaju kod obrade vrijednosti na granicama particije.

## Testiranje bazirano na uputama

Upute uključuju informacije o tome koje su vrste testova učinkovite za otkrivanje grešaka.

Kod testiranja programa koji uključuju nizove ili liste upute bi bile:

1. Testiraj program s sljedovima koji imaju jednu vrijednost

* Programeri najčešće podrazumijevaju da se slijed sastoji od različitih vrijednosti, pa ako se unosi ista program ne mora dobro raditi.

1. Koristiti različite sljedove, različitih vrijednosti u različitim testovima

* Umanjuje mogućnost da će neispravan program slučajno generirati ispravan izlaz zbog neke slučajne karakteristike ulaza.

1. Testove napraviti tako da se pristupa prvom, srednjem i zadnjem elementu slijeda.

* Otkriva greške na granicama particije.

1. Testirati program sa slijedom duljine 0.

Neke od najopćenitijih uputa su:

* Odabrati ulazne vrijednosti tako da generiraju poruke o grešci.
* Odabrati ulazne vrijednosti tako da izazovu preljev ulaznih spremnika.
* Ponavljanje istih ulaz ili serija ulaza više puta.
* Isforsirati generiranje neispravnih izlaznih vrijednosti.
* Isforsirati rezultate proračuna da budu preveliki ili premali.

## *Testiranje komponenti*

Softverska komponenta se najčešće sastoji od nekoliko objekata u interakciji.

* + Npr. kod meteorološke stanice komponenta za rekonfiguraciju sadrži sve objekte koji pokrivaju bilo koji aspekt rekonfiguracije.

Funkcionalnostima tih objekata se pristupa preko definiranog sučelja komponente, pa se testiranje komponenti u biti svodi na testiranje ponaša li se sučelje komponente prema specifikaciji.

* + Pretpostavka je da su svi jedinični testovi za objekte unutar komponente uspješno obavljeni.



## Testiranje sučelja

Cilj je otkriti greške koje nastaju kao rezultat grešaka sučelja ili

krivih pretpostavki o sučeljima.

Moguće grešaka sučelja razlikuju se ovisno o vrstama sučelja:

1. ***Parametarska sučelja***– sučelja kod kojih se podaci

prebacuju iz jedne komponente u drugu. (Metode

objekta imaju parametarsko sučelje.)

1. ***Sučelja zajedničke/dijeljene memorije***– sučelja u kojima

komponente dijele dio memorije. Jedan podsustav

smješta podatke u memoriju, a drugi im pristupa na toj

memorijskoj lokaciji. Ovo sučelje se često koristi kod ugradbenih uređaja, gdje senzori stvaraju podatke, a zatim im pristupaju druge komponente zadužene za dohvat i obradu.

1. ***Sučelja procedura***– jedan podsustav obuhvaća niz procedura, koje onda mogu pozivati drugi podsustavi. Objekti i komponente namijenjene za ponovno korištenje imaju ovakvu vrstu sučelja.
2. ***Sučelja za prosljeđivanje poruka***– sučelje kod kojeg jedna komponenta traži servis od druge na način da joj proslijedi poruku. Povratna poruka sadrži rezultat izvršavanja servisa. Neki objektno orijentirani sustavi koriste ovaj oblik sučelja, kao i klijent-server sustavi.

## Moguće greške sučelja

Greške sučelja spadaju u najčešći oblik grešaka kompleksnih sustava, a mogu se podijeliti u 3 klase:

* ***Zloupotreba sučelja*** – Korištena komponenta poziva drugu komponentu i pri tome koristi sučelje na krivi način. Česta je kod parametarskih sučelja (šalju se parametri neispravnog tipa, u neispravnom rasporedu, neispravan broj parametara).
* ***Nesporazum kod korištenja sučelja***– Komponenta koja poziva sučelje nije dobro shvatila specifikaciju sučelja komponente koju poziva pa pretpostavlja kako se komponenta ponaša. Pozvana komponenta se ne ponaša na očekivani način, što uzrokuje nepredvidivo ponašanje komponente koja ju je pozvala. Npr. metoda za binarno pretraživanje očekuje da će dobiti sortirani niz, ukoliko niz nije sortiran algoritam daje neočekivane rezultate.
* ***Greške zbog vremenske neusklađenosti******komponenti***– Komponenta koja poziva i pozvana komponenta rade različitom brzinom zbog čega neka od njih pristupa zastarjelim informacijama. Događaju se kod sustava za rad u realnom vremenu.

## Upute za testiranje sučelja

Neke od općenitih uputa za testiranje sučelja su:

* Provjeriti kod koji treba testirati i eksplicitno navesti svaki poziv vanjskim komponentama. Zatim dizajnirati testove tako da parametri pozvanih vanjskih procedura su na ekstremnim krajevima opsega. Najveća je vjerojatnost da će takve ekstremne vrijednosti otkriti nekonzistentnost sučelja.
* Uvijek treba testirati pokazivačke parametre s NULL pokazivačima
* Dizajnirati testove koji će uzrokovati grešku. Različite pretpostavke o greškama su neki od najčešćih nesporazuma specifikacije.
* Korištenje stres testova kod sustava koji izmjenjuju poruke. Dizajnirati testove tako da generiraju mnogo više poruka među komponentama nego što će to biti slučaj u praksi. To je efikasan način otkrivanja problema s vremenskom neusklađenosti među komponentama.
* U sustavima koji koriste zajedničku memoriju testove je potrebno dizajnirati tako da se komponente koje koriste tu memoriju aktiviraju različitim redoslijedom. Ovo može otkriti implicitne pretpostavke programera o rasporedu kojim se stvaraju i koriste ti podaci.

## *Testiranje sustava*

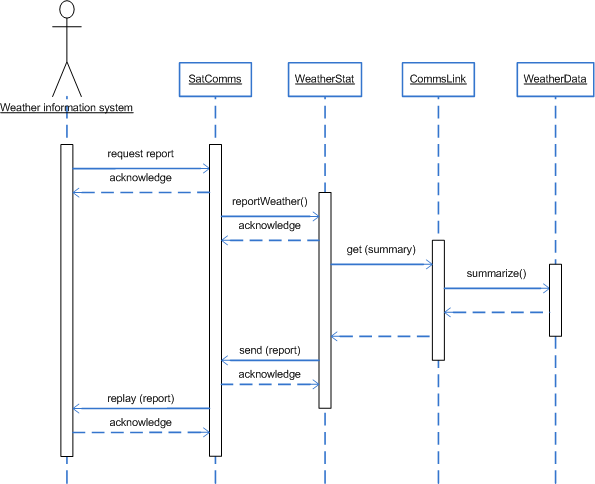
Testiranje sustava za vrijeme razvoja uključuje integraciju komponenti u neku od izvršnih verzija sustava i zatim testiranje integriranog sustava.

Fokus je na testiranju interakcije među komponentama i provjerava se:

* jesu li komponente kompatibilne;
* surađuju li međusobno na ispravan način;
* izmjenjuju li ispravne podatke u pravom vremenu preko sučelja.

Testiranje sustava provjerava izranjajuća svojstva sustava.

## "Use-case" testiranje

* "Use case" scenariji razvijeni u fazi specifikacije zahtjeva i dizajna identificiraju interakcije među dijelovima sustava i mogu se koristiti kao osnova za testiranje.
* Svaki "use case" obično uključuje nekoliko komponenti sustava pa testiranje nekog "use case-a" forsira sve te interakcije.
* Slijedni dijagram koji je dio "use case-a" daje vezu između komponenti i interakcija koje se testiraju.

## Primjer – Meteorološka stanica

Iz slijedni dijagrama prikupljanja podataka o vremenu se vidi da zahtjev za izvještaj o vremenski prilikama nekoj meteorološkoj stanici rezultira izvršavanjem slijedećih metoda:   
SatComms:request-> WeatherStation:reportWeather-> Commslink:Get(summary)-> WeatherData:summarize

## Testne politike

* Za većinu sustava je teško procijeniti koliko je testova potrebno i kada prestati s testiranjem.
* Opsežno testiranje sustava (testiraju se svi mogući sljedovi izvršavanja) nije moguće, pa je potrebno izabrati nekoliko mogućih testnih primjera.
* U idealnom slučaju treba postojati politika odabira testnih paketa.
* Ova politika se može biti zasnovana na:
* Osnovnim pravilima testiranja, npr. da se sve programske linije moraju izvršiti bar jednom;
* Saznanjima o korištenju sustava i fokus je na testiranju svih funkcionalnosti:
* Potrebno je testirati sve funkcionalnosti sustava kojima se može pristupiti preko izbornika (menija);
* Kombinacija funkcija (npr. formatiranje teksta) kojima se pristupa preko istog menija isto treba testirati;
* U slučaju da aplikacija koristi korisnički unos potrebno ju je testirati s ispravnim i neispravnim vrijednostima.

## Razvoj upravljan testiranjem (TDD)

* Razvoj upravljan testiranjem (eng. Test Driven Development) je pristup razvoja programa u kojem su testiranje i programiranje usko povezani.
* TDD se pojavio kao dio Agilnih metoda (XP), ali može se koristiti i kao dio razvojnog procesa baziranog na planiranju.



* Testovi se pišu prije koda i prolazak

testa je cilj razvoja.

* Kod se razvija u inkrementima, zajedno

s testovima za taj inkrement. Novi

inkrement ne može početi prije nego

što kod ne prođe sve testove.

## Procesne aktivnosti TDD-a

1. U proces se kreće identifikacijom inkrementa potrebnih funkcionalnosti. Pri tome svaki inkrement bi trebao biti dovoljno mal da se može napisati s nekoliko linija koda.
2. Piše se test za tu funkcionalnost i implementira u obliku automatiziranog testa.
3. Pokreče se svi testovi, pri tome pošto funkcionalnost još nije implementirana testovi ne prolaze.
4. Implementira se funkcionalnost i ponovno pokreće test. Ovaj korak može uključivati procjenjivanje postojećeg koda s ciljem poboljšanja i dodavanje novog koda.
5. Jednom kada se svi testovi uspješno izvrše, ide se na implementaciju slijedećeg djelića funkcionalnosti.

## Prednosti TDD razvoja

Pokrivenost koda – svaki napisani djelić koda bi trebao imati bar jedan pridruženi test.

Regresijsko testiranje – testni paket se razvija inkrementalno s programom i uvijek se provode svi testovi kako bi se provjerilo da neke izmjene u programu nisu uzrokovale greške.

* "Ručno" regresijsko testiranje troši dosta vremena.
* Korištenje automatskih testnih okolina omogućava regresijsko testiranje na vrlo jednostavan način. Svi testovi se ponovno pokreću kad god se napravi promjena u programu.
* Jedna od najvećih prednosti TDD-a je što uvelike smanjuje cijenu regresijskog testiranja.

Pojednostavljeno uklanjanje grešaka – kada test javi grešku, jasno je gdje je greška nastala .

Dokumentacija sustava – testovi su sami po sebi oblik dokumentacije koji opisuju što bi kod trebao raditi.

## TDD u praksi

Korištenje TDD ima najviše smisla kod razvoja novog softvera i to kod malih i srednje velikih projekata. TDD nema smisla kod:

* Razvoja sustava koji koriste već gotove komponente ili naslijeđene dijelove softvera. Tada bi trebalo napisati testove tako da provjeravaju rad tih dijelova kao cjeline.
* Višenitnih sustava – Različite niti se mogu ispreplesti u različitim trenutcima i testovima što može uzrokovati različite rezultate.

## *TESTIRANJE VERZIJE ZA OBJAVU*

Testiranje verzije za objavu (eng. release testing) je proces testiranja verzije programa koja se namjerava koristiti van razvojnog tima.

* Najčešće se tu radi o verziji za korisnike, ali ona može biti namijenjena drugim timovima koji razvijaju povezane sustave.

Primarni cilj testiranja verzije za objavu je uvjeriti naručioca da je sustav dovoljno dobar za upotrebu.

* Treba dokazati da sustav pruža tražene funkcionalnosti, performanse, pouzdanost i da je dovoljno dobar za korištenje.

Najčešće se radi o "black-box" testiranju gdje se testovi izvode iz specifikacije sustava.

## Testiranje verzije za objavu i testiranje sustava

Testiranje verzije za objavu je dio testiranja sustava, a važne razlike su:

* Odvojeni tim koji nije bio uključen u razvoj sustava bi trebao provesti testiranje za objavu.
* Testiranje sustava koje provodi razvojni tim bi se trebalo fokusirati na pronalaženje grešaka, dok testiranje verzije za objavu provjerava da sustav odgovara zahtjevima i dovoljno je dobar za upotrebu (validacijsko testiranje).

Testiranje verzije za objavu se baviti testiranjem:

* Funkcionalnosti (bazirano na zahtjevima i scenarijima);
* Performansi.

## Testiranje bazirano na zahtjevima

Zahtjevi bi trebali biti napisani tako da ih je lako testirati, tj. da je lako osmisliti test koji provjerava je li zahtjev realiziran ili ne.

Testiranje bazirano na zahtjevima uključuje razmatranje svih zahtjeva i pisanje jednog ili više testova koji provjeravaju je li zahtjev zadovoljen ili ne.

* Ovdje se radi više o validaciji sustava, a ne uklanjanju grešaka.

INFORMACIJSKI SUSTAV PACIJENATA KLINIKE

Primjer zahtjeva:

* + Ukoliko se zna da je pacijent alergičan na neki lijek, tada kod izdavanja recepta za taj lijek aplikacija rezultira upozorenjem liječniku koji izdaje recept.
  + Ukoliko liječnik odabere ignorirati upozorenje, treba dati detaljno objašnjenje.

Primjer testa baziranog na zahtjevima:

* Postaviti zapis za pacijenta koji nije alergičan na niti jedan lijek. Prepisati mu lijek i provjeriti da nema upozorenja.
* Postaviti zapis za pacijenta koji je alergičan na jedan lijek. Prepisati mu taj lijek i provjeriti da se javlja upozorenje.
* Postaviti zapis za pacijenta koji je alergičan na dva ili više lijekova.
* Prepisati mu te lijekove odvojeno i provjeriti da se svaki put javlja upozorenje.
* Prepisati mu odjednom neka od ta dva lijeka i provjeriti javljaju li se dva upozorenja.
* Prepisati mu jedan od tih lijekova i kada se javi upozorenje ignorirati ga kako bi provjeri li traži li sustav objašnjenje.

## Testiranje bazirano na scenarijima

* Testiranje na osnovu scenarija je pristup testiranju verzije sustava za objavu kod koje se smišlja tipičan scenarij korištenja sustava i on se koristi za izradu testnih primjera.
* Primjer scenarija korištenja inf. sustava pacijenata klinike:

Kate is a nurse who specializes in mental health care. One of her responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side -effects.

On a day for home visits, Kate **logs into the MHC-PMS** and uses it to **print her schedule of home visits for that day, along with summary information about the patients to be visited**. She requests that the records for these patients **be downloaded to her laptop**. She is **prompted for her key phrase to encrypt the records on the laptop**.

One of the patients that she visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side -effect of keeping him awake at night. Kate looks up Jim’s record and is **prompted for her key phrase to decrypt the record**. She **checks the drug prescribed and queries its side effects**. Sleeplessness is a known side effect so she **notes the problem in Jim’s record** and suggests that he visits the clinic to have his medication changed. He agrees so Kate **enters a prompt to call him** when she gets back to the clinic to make an appointment with a physician. She ends the consultation and the system re-encrypts Jim’s record.

After, finishing her consultations, Kate returns to the clinic and **uploads the records of patients** visited to the database.

Karakteristike sustava koje se testiraju iz tog scenarija:

* + Identifikacija korisnika prijavom u sustav.
  + Prebacivanje podataka o određenim pacijentima na prijenosno računalo.
  + Ispis rasporeda kućnih posjeta
  + Kodiranje i odkodiranje kartona pacijenata na mobilnom uređaju.
  + Dohvat i izmjena podataka u kartonu.
  + Poveznice na baze podataka lijekova koje sadrže informacije o nuspojavama.

## Testiranje performansi

* Kada se sustav u potpunosti integrira moguće je testirati izranjajuća svojstva poput performansi i pouzdanosti.
* Testovi performansi moraju biti dizajnirani tako da pokažu da sustav radi zadatke u predviđenom vremenu.
  + Najčešće se pokreće više testova s tim da se opterećenje sustava u svakom slijedećem testu povećava, sve dok performanse postanu neprihvatljive.
  + Testno opterećenje sustava bi trebalo što više nalikovati stvarnom (vrste korisnika sustava).
  + “Stress testing” je vrsta testiranja performansi kada se sustav namjerno preoptereti kako bi se testiralo ponašanje u slučaju greške.

## *KORISNIČKO TESTIRANJE*

**Korisničko testiranje** je faza u testnim procesima u kojoj korisnik sudjeluje i daje savjete o testiranju sustava.

* + Može uključivati formalno testiranje sustava naručenog od vanjskog dobavljača, ili može biti neformalni proces u kojem korisnici eksperimentiraju s novim softverom.

Korisničko testiranje je neophodno čak i kada se provede opsežno testiranje sustava i verzije za objavu.

* + Utjecaji stvarne radne okoline korisnika imaju velik efekt na pouzdanost, performanse, iskoristivost i robusnost sustava a to se ne može replicirati u testnoj okolini.

## Vrste korisničkog testiranja

U praksi postoje tri vrste korisničkog testiranja

* + ***Alfa testiranje***– Korisnici softvera rade s razvojnim timom kako bi testirali sustav tamo gdje se on razvija.
    - Korisnici upućuju na probleme koji su promakli razvojnom timu.
    - Koristi se kod razvoja “shrink-wrapped” softvera.
  + ***Beta testiranje***– Korisnici softvera eksperimentiraju s objavljenom verzijom i prijavljuju probleme koji se pojave razvojnom timu.
  + Najčešće se koristi za produkte koji se koriste u različitim okolinama, jer razvojni tim ne može testirati sve moguće slučajeve.
  + Važan je za otkrivanje interakcijskih problema softvera i okoline u kojoj se koristi.
  + ***Test prihvaćanja***– Korisnici testiraju sustav kako bi vidjeli zadovoljava li njihove potrebe i mogu li ga primijeniti u svojoj okolini. Uglavnom se radi za sustave po narudžbi.

## Proces testa prihvaćanja

Postoji šest faza u procesu testa prihvaćanja:

* 1. ***Definiranje kriterija za prihvat***– Ova faza bi se trebala dogoditi rano u procesu prije nego se potpiše ugovor. Trebaju biti dio ugovora, što nije uvijek moguće u praksi jer možda ne postoji detaljna specifikacija zahtjeva ili je došlo do velike promjene zahtjeva u toku razvoja.
  2. ***Planiranje testa prihvaćanja***– Donosi se odluka o resursima, vremenu i budžetu namijenjenom za test. Pravi se raspored testiranja. Definiraju se rizici u procesu testiranja (pad sustava, loše performanse) i raspravlja kako ih umanjiti.
  3. ***Stvaranje testa prihvaćanja***– Kada se definiraju kriteriji za prihvat rade se testovi koji ih provjeravaju. Cilj je provjera svih funkcionalnih i ne-funkcionalnih ali je u praksi teško definirati objektivne kriterije prolaznosti testa.
  4. ***Pokretanje testa prihvaćanja*** – Dogovoreni testovi prihvaćanja se pokreću na sustavu (idealno u stvarnoj radnoj okolini). Najčešće se ne može automatizirati jer u testiranju sudjeluju i krajnji korisnici (može biti potrebna njihova obuka).
  5. ***Pregovor o rezultatima testa***– Malo je vjerojatno da će svi definirani testovi proći i da neće biti nikakvih problema i tada razvojni tim i kupac trebaju donijeti odluku je li sustav dovoljno dobar da se krene s korištenjem i što razvojni tim treba napraviti u vezi otkrivenih problema.
  6. ***Odbijanje/prihvat sustava***– Sastanak razvojnog tima i kupca na kojem se donosi odluka prihvaća li se sustav ili ne. Ukoliko sustav nije dovoljno dobar, razvojni tim ispravlja identificirane probleme. Test prihvaćanja se u tom slučaju ponovo ponavlja.



## Agilne metode i testiranje prihvaćanja

Kod Agilnih metoda ko što je XP test prihvaćanja ima potpuno drugo značenje.

* + Korisnik je dio razvojnog tima (radi kao alfa tester) i kreira zahtjeve u obliku korisničkih priča, definira testove na osnovu kojih se donosi odluka podržava li sustav korisničke priče. Testovi su automatizirani i razvoj ide dalje tek kada prođu svi testovi.
  + Ne postoji odvojeni proces testa prihvaćanja.

Glavni problem je koliko kvalitetno obavlja posao predstavnik klijenata koji pomaže u radu razvojnom timu.