UNIVERZITET
Union Nikola
Tesla

**POSLOVNI I PRAVNI
FAKULTET - BEOGRAD**

RAZVOJ SOFTVERA

- PREDAVANJA -

Predavač: Prof. dr Borivoje M. Milošević

2020

Osnovni koncepti i metodologije razvoja informacionih sistema

Projektovanje informacionih sistema:

Prilikom projektovanja informacionih sistema je potreban metodološki pristup. Informacioni sistem koji nastaje treba da korisniku pruži informacije koje su:

• **Relevantne** – Pre nego što se krene sa razvojem informacionog sistema treba izvršiti analizu procesa upotrebe informacija i doneti odluku koje su informacije relevantne za korisnika, a koje to nisu. Informacija je relevantna u toku procesa donošenja odluke, ako njen sadržaj može da utiče na samu odluku.

• **Tačne** – Informacija treba da bude onoliko tačna koliko je to potrebno. Ponekad je prihvatljivo da informacija bude u određenoj meri netačna, posebno ako je tačnost informacije u direktnoj vezi sa poskupljenjem dobijanja informacije, bilo u pogledu vremena bilo u pogledu novca.

• **Došle na vreme** – Informacija mora da do korisnika stigne u definisanom vremenskom intervalu, u kome će biti od koristi. Zakasnele informacije nisu korisne. Ponekad je bolje brže dobiti informaciju, nego čekati da ona bude apsolutno tačna.

• **Usmerena** – Informacija mora da stigne do pravog korisnika.

• **U pravom obliku** – Način na koji se informacija prikazuje korisniku utiče na njegovu efikasnost. Umesto da se prave tradicionalni tekstualni izveštaji, treba težiti grafičkom prikazu informacija.

• **Interaktivne prirode** – Korisnik treba da dobije onoliko informacija koliko mu je potrebno. Tek kada zatraži dodatnu informaciju treba mu je pružiti. Ne treba ga unapred bombardovati svim informacijama.

• **Konstrolisane** – Neke informacije su osetljive i mogu biti od koristi konkurenciji. U skladu sa time moraju se preduzeti odgovarajući koraci koji će obezbediti sigurnost informacija.

Pristupi projektovanju informacionih sistema

Prilikom projektovanja informacionih sistema mogu da postoje različiti pristupi. Neki od njih su:

• **Kupovina softverskog proizvoda** – Ovo je pristup koji se koristio na početku priče o informacionim sistemima. Težište kod ovakvog pristupa je razvoj računarskog sistema koji će zameniti svu papirologiju, koja je ranije obavljana ručno. Kod ovakvog pristupa obično nema velike analize zahteva sistema. Informacije koje ovakav sistem daje su obično u obliku izveštaja, sa puno podataka, u kojima je teško snaći se i u kojima je teško izdvojiti pravu informaciju.

• **Iz početka** – Ovo je pristup nastao kao odgovor na nedostatke prethodnog pristupa. Kao što i ime pokazuje kod ovog pristupa se razvoju informacionog sistema pristupa od nule.

• **Ključne informacije** – Kod ovog pristupa se pretpostavlja da su neke informacije od ključnog značaja za uspešno funkcionisanje sistema. To na primer, mogu biti ukupna gotovina kojom preduzeće raspolaže, nivo pražnjenja zaliha ili odnos dobijenog i uloženog. Razvoj informacionog sistema se odvija tako što se pronađu te ključne informacije, a onda se informacioni sistem pravi tako da se te informacije mogu uvek dobiti.

• **Kompletna studija** – Ovaj pristup se zasniva na kompletnom proučavanju procesa rada i poređenju trenutne situacije sa onom koja će nastati posle uvođenja informacionog sistema. Postupak se obično sastoji u razgovoru sa svima koji učestvuju u procesu i određivanju informacija koje su za njih bitne. Ovakav način rada može biti razumljiv i koristan u otkrivanju nedostataka postojećih sistema. Sa druge strane, može biti jako skupo intervjuisati sve ljude, a sve to može dovesti do prikupljanja velike količine podataka, koje nije lako analizirati.

Softversko inženjerstvo

Razvoj novih programa se ubrzao krajem 60-ih godina prošlog veka. Osnovna pokretačka snaga je bila sve veća primena u vojsci i pojava novih generacija računara. U to vreme nisu postojale jasne smernice za pisanje softvera, definisanje zahteva i uspešno vođenje složenih projekata.

Sistemi koji su u to vreme razvijani su generalno bili vrlo jednostavni.
Kreiranje sistema bez ikakvih ograničenja i pisanje programa po principu „samo neka radi“ doveli su do neuspeha velikog broja softverskih projekata. Pojavila se softverska kriza.

Na konferenciji NATO iz 1968 su pokušali da nađu rešenje za nastale probleme, pa se tada prvi put javio termin softversko inženjerstvo. Rešenje za haos koji se javio u projektima vezanim za razvoj softvera se tražilo u primeni dokazanih metoda i principa iz oblasti mašinstva i građevine. Tu se odmah postavlja pitanje da li se razvoj softvera može upoređivati sa drugim inženjerskim disciplinama. Razlog je što se stanje softvera ne može naučno verifikovati ili izvesti na osnovu neke jednačine.

Zaključak je da ne postoje čvrsti principi za razvoj softvera. Najviše što se može dobiti su praktične preporuke (best practices). Praktične preporuke predstavljaju skup metoda, praksi, alata i procesa za koje se generalno smatra da su industrijski dokazani i optimalni.

Softversko inženjerstvo obuhvata važna područja kao što su:

- vođenje poslovanja i IT,
- razvoj softverskih metodoligija i okvira,
- troškovi razvoja
- trajanje razvoja,
- rizici u razvoju softvera,
- ugrađivanje kvaliteta razmišljanje u procesu razvoja softvera,
- testiranje,
- upravljanje razvojnim timovima,
- project management,
- projekt izveštavanje,
- projekt brzine razvoja projekta,
- komunikacija svih učesnika

Procesu programiranja predhode izvesne pripremene radnje i one zahtevaju:
- potpuno razmatranje i precizno formulisanje problema koji se rešava,
- izbor matematičkih metoda i postavljanje matematičkog modela,
- izbor i analiza numeričkih postupaka sa tačke gledišta efikasnog rešavanja datog problema na računaru.

PROGRAMIRANJE je proces zadavanja skupa naredbi u nekom programskom jeziku kako bi se izvršila neka aktivnost, odnosno, rešio određeni problem.

Proces programiranja obuhvata sledeće faze:

IDENTIFIKACIJA PROBLEMA	Svaki problem može imati mnogo različitih rešenja, ali sva ona moraju imati nešto zajedničko. Znači, ovde pokušavamo da uradimo tačno ono šta se od našeg programa zahteva da uradi.
PROJEKTOVANJE I RAZRADA ALGORITAMA	Ako je potrebno, ili to zahteva upotreba određenog programskog jezika, vršimo projektovanje rešenja na određenoj naučnoj platformi i opisujemo ga izradom algoritma.
DIZAJNIRANJE REŠENJA	Jednom kada definišemo stvari potrebne za rešavanje problema, i specificiramo koju će formu zauzeti rešenje problema, potrebno je izaći na kraj sa pitanjem kako uvesti takvu specifikaciju u program na osnovu kojeg će se odraditi zadatak.
PISANJE PROGRAMA	Programiranje je zadatak kojim ćemo predstaviti naše dizajnirano rešenje računaru, učeći ga da po našem putu izvrši rešavanje problema Obično špostoje tri nivoa pisanja programa: n Kodiranje – (izbor EDITORA) n Kompilacija n Debugging
TESTIRANJE PROGRAMA	Testiranje programa je ispitivanje da li program radi ono šta smo predvideli da radi. Ovo je neizostavan proces, jer iako je kompajler proverio da li je program korektno napisan, on ne može proveriti da li ono šta smo napisali aktuelno rešava naš problem.
IZRADA DOKUMENTACIJE	Izrada dokumentacije je takođe neophodna jer ona spada u sisteme kvalitetnog pisanja softvera.

Izazovi pri izradi softvera

Svaki problem se može rešiti na više načina. Oni se razlikuju po efikasnosti, preciznosti, mogućnosti modifikovanja, korisnosti, razumljivosti i drugim osobinama. Stoga pisanje softvera zahteva znanje, ali i domišljatost i veštinu.

Haker  ⇨ Ume da napiše kod koji nešto radi.

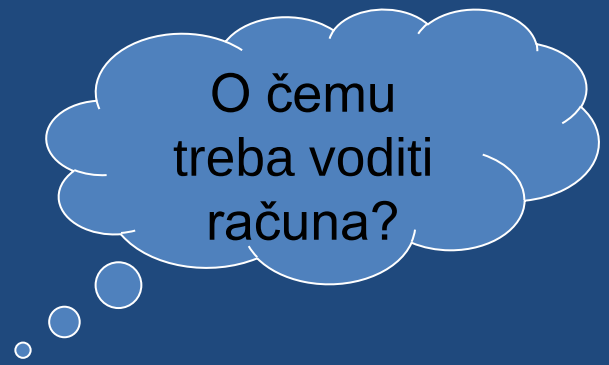
Ume da proizvede sveobuhvatan stabilan i razumljiv kod koji se lako održava i koji efikasno radi ono zbog čega je napravljen. Taj kod predstavlja visoko kvalitetan softver.

⇦  Softverski inženjer

Izazovi pri izradi softvera

I pored toga što proizvođači teže softveru bez mana, u stvarnosti mnogi softverski proizvodi sadrže nedostatke.

<u>Neočekivana upotreba sistema</u>
- usled zloupotrebe
- usled nestručnog rukovanja



<u>Tržište diktira brz razvoj softvera</u>
- brza isporuka ostavlja malo vremena za testiranje, pa su greške češće
- ponekad je teže ispraviti uočene nedostatke, nego napisati kompletan novi softver

<u>Kvalitet je neophodan</u>
- što je nedostatak duže neotkriven, njegovo otklanjanje više košta (troškovi ispravljanja greške u fazi analize su 10 puta manji od troškova nakon isporuke).

Šta je dobar softver?

Kvalitet softvera zavisi od konteksta posmatranja. Na primer, nedostaci koji se tolerišu u softveru za obradu teskta, ne mogu se prihvatiti u sistemima gde je faktor bezbednosti izuzetno bitan.

Kvalitet softvera mora se posmatrati na više načina:

- Kvalitet proizvoda
- Kvalitet postupka izrade proizvoda
- Kvalitet proizvoda u kontekstu poslovnog okruženja u kome će se koristiti

Kvalitet proizvoda

Karakteristike proizvoda koje određuju kvalitet zavise od toga ko analizira softver

Korisnik

smatra da je softver kvalitetan ako radi na odgovarajući način, lako se uči i koristi.

Korisnik meri: tip i broj otkaza.

Projektant

razmatra interne karakteristike proizvoda i procenjuje nedostatke.

Projektant meri: broj nedostataka u zahtevima, projektu i kodu.

Modeli kvaliteta

dovode u vezu spoljni pogled korisnika i unutrašnji pogled programera na softver.

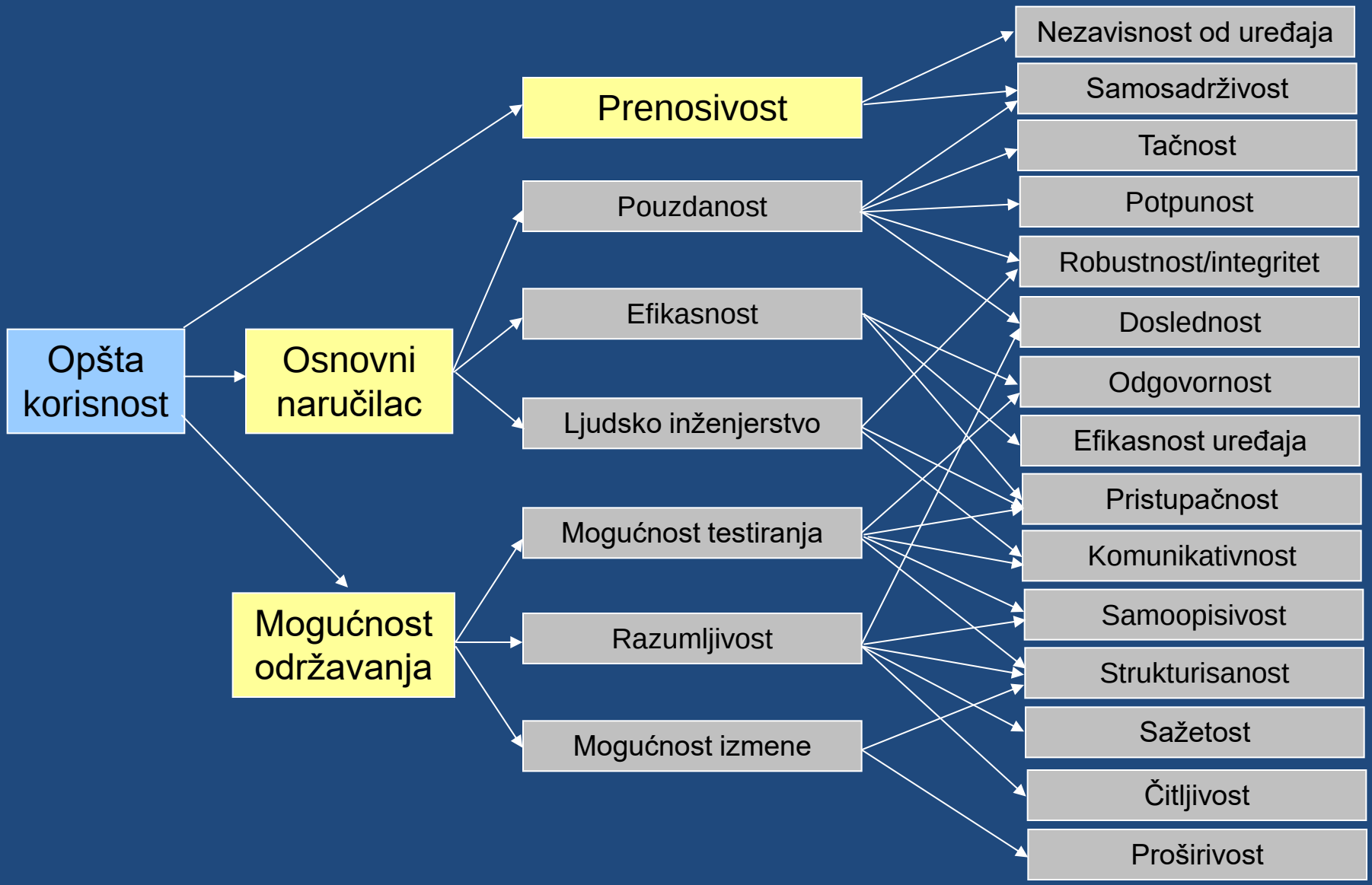
Boehm-ov model kvaliteta

Boehm-ov model kvaliteta (Boehm i dr., 1978.) je jedan od najpoznatijih modela. On predstavlja hijerarhiju karakteristika od kojih svaka doprinosi ukupnom kvalitetu. U model su uključena očekivanja kako korisnika, tako i programera.

Po ovom modelu, kvalitetan softver je onaj koji:

- radi ono što korisnik od njega traži,
- ispravno i efikasno koristi računarske resurse,
- jednostavan je za učenje i korišćenje,
- dobro je projektovan, dobro kodiran i lako se testira i održava.

Boehm-ov model kvaliteta



Kvalitet procesa

Kvalitet procesa razvoja softvera je jednako važan kao i kvalitet proizvoda. Ako neka od aktivnosti krene u pogrešnom smeru, to može da pogorša kvalitet proizvoda. Zbog toga se radi modelovanje postupka koje omogućava lakšu analizu postupka i nalaženje načina za njegovo poboljšanje.

Pitanja koja treba postaviti u procesu razvoja:

- Gde i kada ćemo verovatno naći neku vrstu nedostatka?
- Kako možemo što ranije da pronađemo nedostatke u procesu razvoja?
- Kako možemo da ugradimo toleranciju na greške, da bi smanjili verovatnoću da nedostatak pređe u otkaz?
- Da li postoje alternativne aktivnosti koje mogu da načine proces efikasnijim, uz osiguranje kvaliteta?

Kvalitet procesa

Kvalitet procesa razvoja softvera je jednako važan kao i kvalitet proizvoda. Ako neka od aktivnosti krene u pogrešnom smeru, to može da pogorša kvalitet proizvoda. Zbog toga se radi modelovanje postupka koje omogućava lakšu analizu postupka i nalaženje načina za njegovo poboljšanje.

Pitanja koja treba postaviti u procesu razvoja:

- Gde i kada ćemo verovatno naći neku vrstu nedostatka?
- Kako možemo što ranije da pronađemo nedostatke u procesu razvoja?
- Kako možemo da ugradimo toleranciju na greške, da bi smanjili verovatnoću da nedostatak pređe u otkaz?
- Da li postoje alternativne aktivnosti koje mogu da načine proces efikasnijim, uz osiguranje kvaliteta?

Kvalitet u kontekstu poslovanja

Kvalitet sa aspekta poslovanja se posmatra u zavisnosti od proizvoda i usluga koje pruža poslovni sistem čiji je softver sastavni deo. Pri tome se analizira poslovna vrednost proizvoda.

Učesnici u razvoju softvera

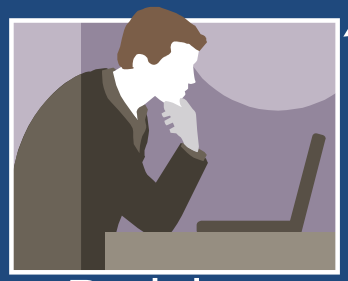
Kupac

Kupac je kompanija, organizacija ili pojedinac koji finansira razvoj softverskog sistema.

Projektant je kompanija, organizacija ili pojedinac koji pravi softverski sistem za kupca.

Ugovorna obaveza

Ima potrebu



Projektant

Softverski sistem

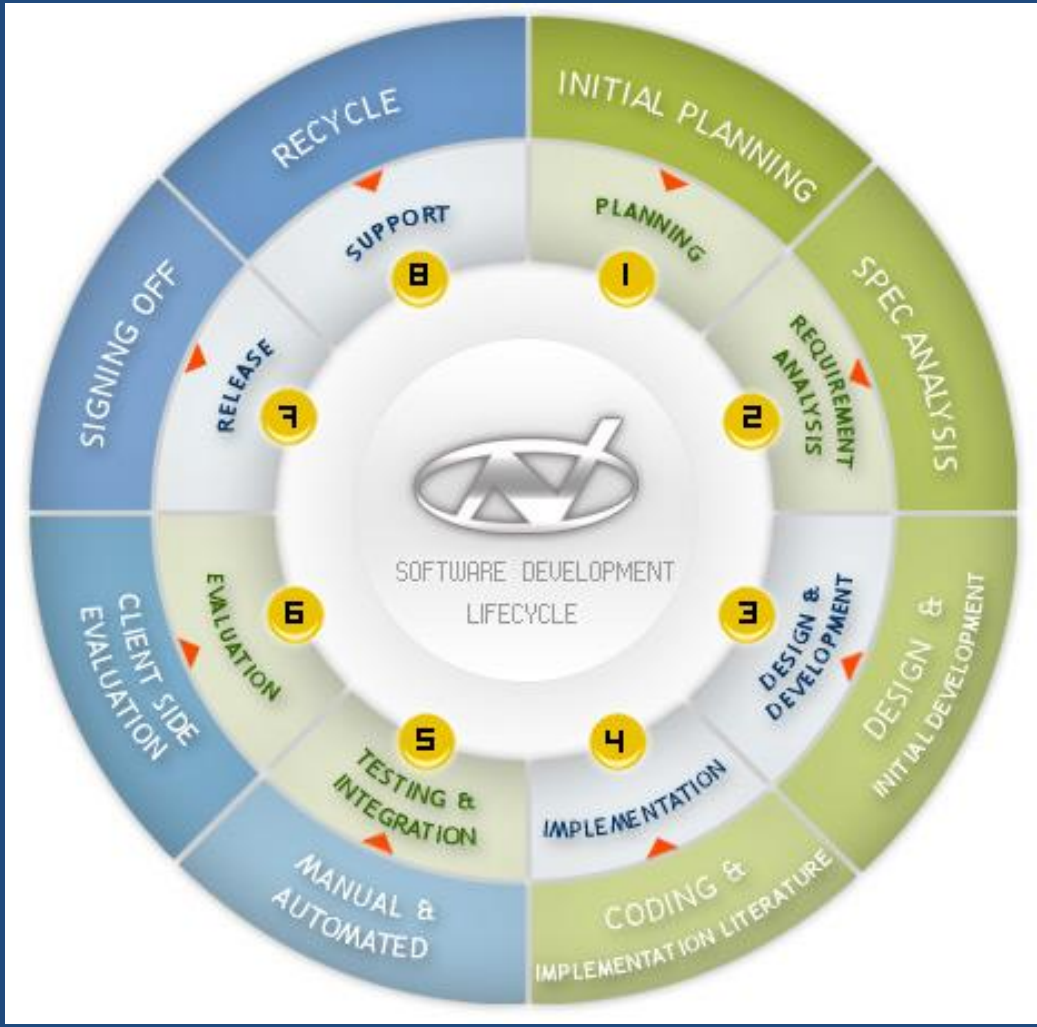
Ima potrebu

Kupac

Korisnik je jedan ili više pojedinaca koji će stvarno koristiti sistem.

Učesnici u projektu mogu istovremeno da imaju više uloga. Na primer, ako neki sektor u kompaniji razvija sam softver za svoje potrebe, on je istovremeno i kupac i korisnik i projektant.

Životni ciklus razvoja softvera



Metodologije razvoja softvera

Metodologija se može definisati kao "skup međusobno povezanih metoda i tehnika", gde se pod metodama podrazumevaju „sistematične procedure" slične tehnikama. Metodologije za razvoj softvera bave se „metodama i tehnikama" za razne elemente koji se pojavljuju u procesu razvoja softvera. Na sledećoj slici prikazani su najvažniji elemenati razvoja softvera od kojih zavisi uspešnost svakog softverskog poduhvata.

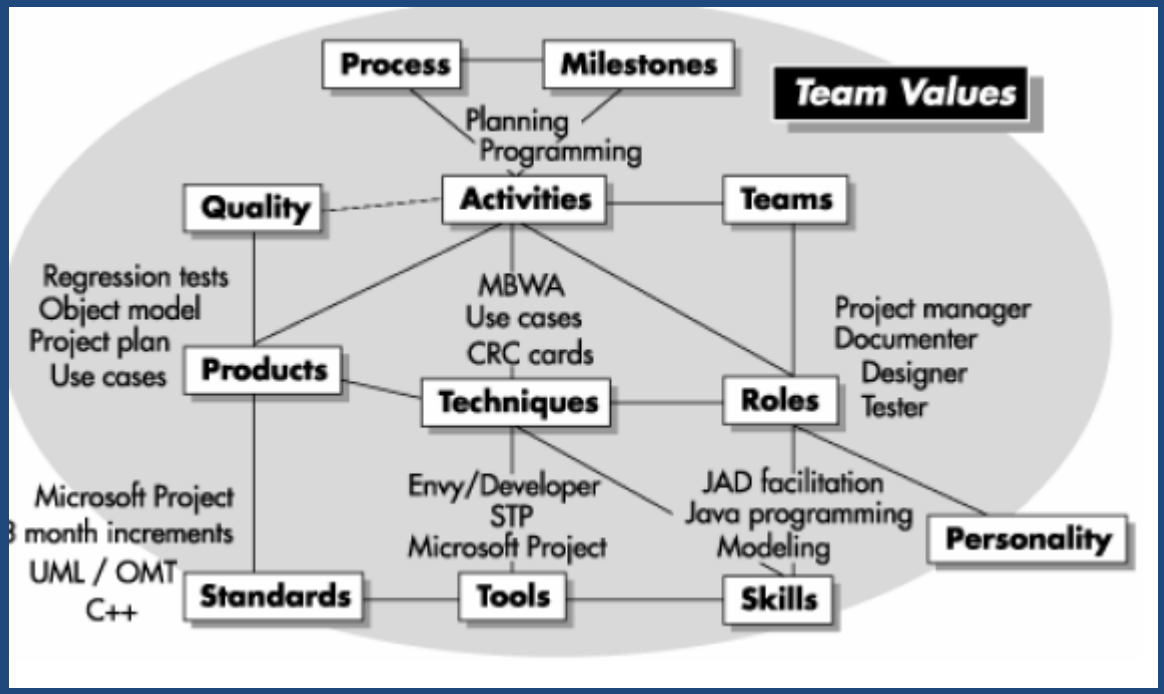
Modeli razvoja se pojavljuju od vremena kada su se projektima razvijali veliki softverski sistemi i prikazuju različite poglede na proces razvoja softvera**. Osnovni razlog njihove pojave** je bila želja da se obezbedi **uopštena šema razvoja softvera**, **koja bi služila kao osnova u planiranju, organizovanju, snabdevanju, koordinaciji, finansiranju i upravljanju aktivnostima razvoja softvera.**

Modeli su apstrakcije koje pomažu u procesu razvoja softvera. Model softvera predstavlja komponente razvoja softvera i razvijen je na osnovu ideja konstruktora i njegove predstavke šta je najznačajnije u tom razvoju.

Model može predstavljati:

- ✓ model procesa razvoja,
- ✓ model softvera ili
- ✓ model načina upravljanja softverom.

Metodologije razvoja softvera



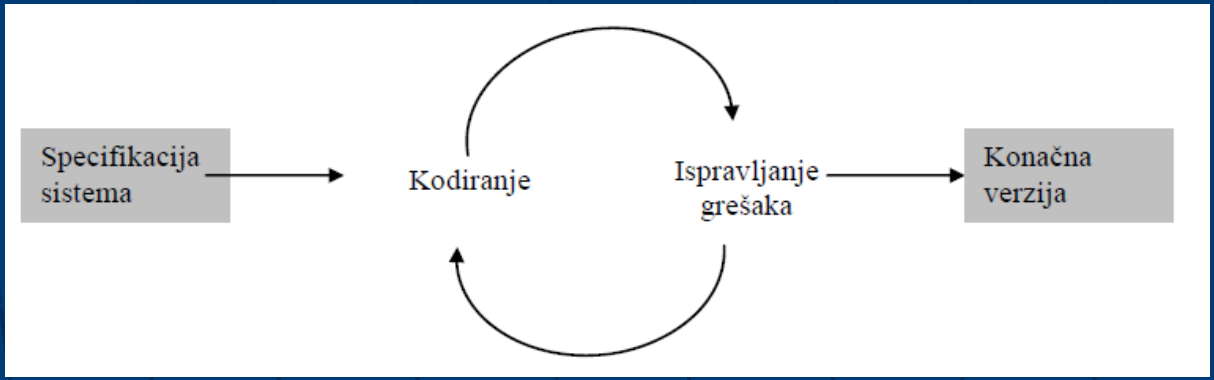
Softverski timovi (Teams) koji se sastoje od menadžera, dizajnera, testera, dokumentalista i sl. (Roles) koji poseduju potrebna znanja i veštine (Skills) , svojim svakodnevnim aktivnostima (Activities) kao što su planiranje i programiranje kroz procese (Process) i ključne rezultate (Milestones), a korišćenjem različitih tehnika (Techniques) i alata (Tools) stvaraju nove oftverske proizvode (Products) određenog kavliteta (Quality) i po „de fakto“ ili „de jure“ standardima (Standards). Naravno, ne treba zaboraviti da su timovi sastavljeni od ljudi koji imaju svoje osobenosti (Personality) o kojima takođe treba voditi računa.

Kada se proces razvoja (proizvodnje) softvera shvati na prikazani način, onda postaje jasnije zašto je potrebno da se taj razvoj odvija prema nekoj unapred usvojenoj metodologiji. Odsustvo metodologije bi za slučaj svakog malo složenijeg softverskog poduhvata vodilo ka haosu sa negativnim posledicama.

Metodologije razvoja softvera

- ✓ **Kodiraj pa popravljaj (Code-fix)**

Ovaj pristup i nije metodologija u pravom smislu reči, ali ga pominjemo pošto neki i danas rade na taj način. Ovakav pristup ponovo uspostavlja princip „samo neka radi“. Inicijalno, kupac može da navede šta je za njega najvažnije, ali to nije najvažnije kod razvoja. Ti njegovi navodi mogu biti u formi skice, elektronske poruke ili neke druge vrste slabe specifikacije.
Razvoj se može zasnivati i na programerovom lokalnom ili ekspertskom znanju o poslu kupca i načinu njegovog rada. S vremena na vreme programer svoju aplikaciju demonstrira kupcima i dobija povratnu informaciju, nakon čega nastavlja rad. Sa sledeće slike se vidi da programeri najviše svog vremena provode u programiranju i ispravljanju grešaka.



Ovakav pristup ima negativne efekte, pa ga ne bi trebalo koristiti. One su: Kvalitet proizvoda je mali, sistem često završava sa nekoordinisanom i zamrešnom zbrkom od koda, sistemi koji se ovako razvijaju se teško proširuju i održavaju, komplikovani sistemi obično imaju malu fleksibilnost. Specifikacija zahteva klijenata može da postoji, a može da se zasniva i na iskustvu i lokalnom, ili ekspertskom znanju programera. **Razvoj počinje brzim ciklusima kodiranja iza kojih sledi popravka.** Ovde iskusniji i veštiji programeri imaju više šanse za uspeh zato što prave manje grešaka. Sa vremena na vreme programer demonstrira svoju aplikaciju kupcima i dobija povratnu informaciju, nakon čega nastavlja razvoj.

Metodologije razvoja softvera

- ✓ **Kaskadni pristup (Waterfall model)**

Kaskadni model je primer modela kod kojeg se razvoj softvera zasniva na ideji o životnom ciklusu softvera. Ideja sa životnim ciklusom softvera je preuzeta od inženjera, koji su mnogo ranije uvideli da svi proizvodi imaju konačan vek upotrebe, koji počinje kreiranjem koncepta, nakon čega slede specifikacija, projektovanje, implementacija, održavanje i zastarelost.

Njegov fazni pristup problemu kroz cikluse planiranja, realizacije i kontrole, omogućio je ostvarenje komplikovanih multidisciplinarnih različitih poduhvata na organizovan i u velikoj meri predvidljiv način. Ovaj koncept nameće potrebu za detaljnim planiranjem svih aktivnosti pre početka projekta preko različitih elaborata i studija. Malo je mesta za kasnije improvizacije i podešavanja.

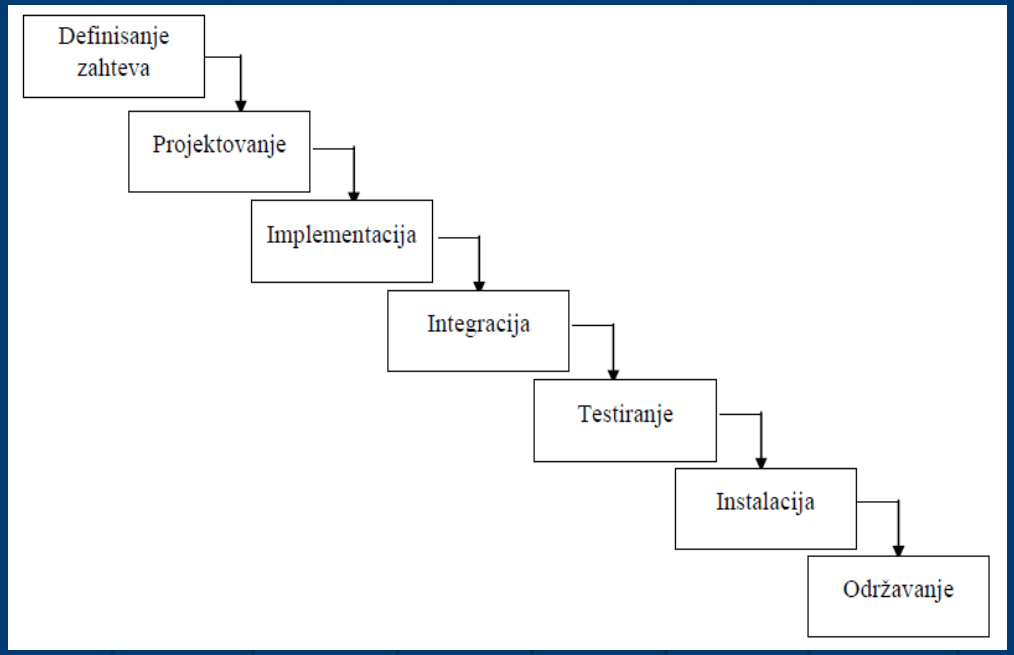
Kaskasdni model je sekvencijalni model razvoja softvera kod kojeg razvoj postojano teče naniže (kao vodopad – waterfall). Smatra se da ovaj termin proizilazi iz članka Rojsa, iz 1970-e godine, kojeg svi navode i kao tvorca ovog modela, mada je on primer ovakvog modela dao kao nešto što ne treba koristiti. Za vreme krize u softverskom inženjerstvu ovakav pristup se nametnuo kao jedino dostupno rješenje nastalih i nagomilanih problema.

Originalan kaskadni model je doživeo mnoge izmene i poboljšanja, ali ćemo ovde predstaviti kako je izgledao na početku. U originalnom modelu u razvoju softvera postoje sledeće faze.

Metodologije razvoja softvera

Kaskadni pristup (Waterfall model

U čistom kaskadnom modelu faze se odvijaju potpuno sekvencijalno. To znači da, dok se ne završi jedna faza, sledeća ne može da počne. Sve počinje definisanjem zahteva. Tek kad se zahtevi kompletiraju, prelazi se na projektovanje. Projektuju se odgovori na sva pitanja koja se postavljaju pred programere i pravi se uputstvo koje oni treba da slede. Projekat je plan za implementaciju zahteva.



Nakon što se završi projektovanje, programeri počinju sa implementacijom, odnosno pisanjem programskog koda. Nakon programiranja, se vrši integracija različitih softverskih komponente, koje su napravili različiti delovi tima. Tek nakon implementacije i integracije se vrši testiranje softvera. Ovde se ispravljaju sve greške koje su proistekle iz prethodnih faza. Nakon toga se program instalira i kasnije održava (u smislu dodavanja novih funkcija i popravke grešaka uočenih tokom eksploatacije).

Kao što se vidi sa slike kaskadni model prelazi na sledeću fazu tek ako je prethodna faza u potpunosti završena i dovedena do savršenstva. Faze su ovde diskretne i nema preklapanja između njih, niti povratka unazad.

Metodologije razvoja softvera

Prednosti kaskadnog modela

Vreme koje se potroši u početnoj fazi kreiranja softvera može kasnije da donese velike uštede. Mnogo puta je dokazano je mnogo jeftinije ako se greška pronađe u početnim fazama (definisanje zahteva i projektovanje). Uštede se odnose kako na novac, tako i na vreme i rad koji treba uložiti u ispravljanje grešaka.

Ako na primer, postane nemoguće implementirati program zbog grešaka u projektu, mnogo je jeftinije popraviti projekat, nego da mesecima kasnije, nakon uloženog velikog rada i potrošenog novca, prilikom integracije komponenti ispravljamo ono što nije dobro.

Ovo je centralna ideja koja stoji iza kaskadnog modela i modela Projektovanja unapred. Vreme koje se provede na tome da se osigura da su zahtevi i projekat apsolutno tačni kasnije donosi uštede u vremenu i radu.

Još jedan argument koji se ističe kao dobra strana kaskadnog modela je njegovo insistiranje na dokumentaciji (dokumenti sa specifikacijom zahteva i dokumenti projekta), kao i na izvornom, programskom kodu.

Kaskadni model, generalno, može biti primenljiv za softverske projekte koji su stabilni (posebno kod projekata kod kojih se zahtevi ne menjaju), kao i tamo gde je moguće i verovatno da projektanti u potpunosti predvide rad sistema i naprave ispravan projekat, pre nego što se počne sa implementacijom.

Metodologije razvoja softvera

Nedostaci kaskadnog modela

Za kaskadni model se tvrdi da nije dobra ideja uglavnom zato što se misli da, za bilo koji ozbiljniji projekat, nije moguće u potpunosti završiti jednu fazu projekta, pre nego što se pređe na sledeću. Na primer, možda klijenti nisu u potpunosti svesni toga šta tačno žele, pre nego što vide radni prototip, koji mogu da komentarišu. Može se desiti da oni stalno menjaju svoje zahteve i nad time programeri i projektanti imaju malo ili nimalo kontrole. Ako klijent svoje zahteve promeni posle završetka faze projektovanja, onda se i projekat mora prilagoditi tome. Na taj način vreme provedeno u prthodnom projektovanju predstavlja uludo potrošeno vreme. Možda tek u trenutku implementacije postane jasno da je neku funkciju programa izuzetno teško ostvariti. U takvim slučajevima je bolje promeniti projekat, nego tvrdoglavno nastaviti sa prethodnim projektom, u pokušaju da se geške u projektu reše u fazi implementacije.

Vrlo je teško u napred proceniti vreme i cenu svake faze procesa razvoja softvera, bez „prljanja ruku“, osim ako su oni koji prave te procene izuzetno iskusni. Samo pojedini članovi tima su u potpunosti kvalifikovani za pojedine faze. Ako u timu imate ljude koji samo kodiraju i ne koriste se kod projektovanja i obrnuto, onda to predstavlja čist gubitak, jer programeri sede dok projektanti rade i obrnuto.

Usled nedostataka koji su uočeni kod kaskadnog modela, predložena su različita proširenja ovog modela, koja pokušavaju da prevaziđu nedostatke, a istovremeno zadrže dobre strane. Jedan od takvih modela razvoja softvera je tzv. V model.

Metodologije razvoja softvera

V model

V model je koji predstavlja proširenje kaskadnog modela. Umesto da se naniže, u nove faze, ide linearno, pojedine faze teku tako da daju oblik slova V. Za V model se može reći da je nastao kao rezultat evolucije u testiranju softvera. Definisane su različite tehnike testiranja, koje su jasno odvojene jedna od druge, što zajedno sa kaskadnim modelom vodi ka V modelu.

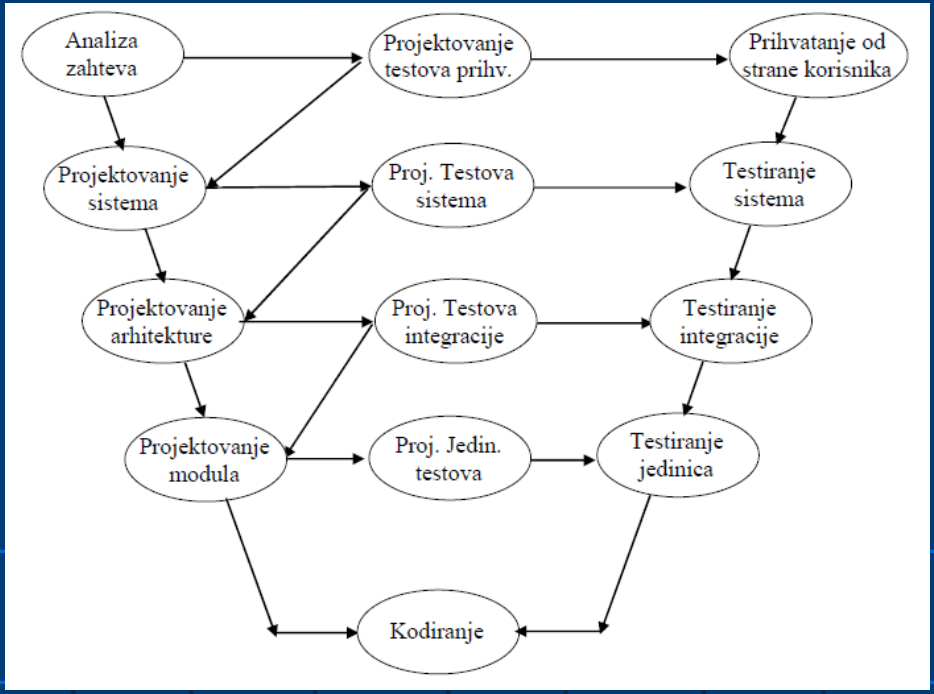
Prva faza kod V modela je **analiza zahteva**. U ovoj fazi se definiše kako bi trebalo da izgleda idealan sistem. U ovoj fazi se ne definiše kako softver treba da se napravi. Obično se rade intervjui korisnika i prave se odgovarajući dokumenti. Ovi dokumenti opisuju rad sistema sa aspekta korisnika. Korisnici treba da pažljivo prouče ove dokumente, jer će oni služiti kao vodič za projektante. U ovoj fazi se definišu i testovi za prihvatanje softvera od strane korisnika.

Druga faza je **projektovanje sistema**. U ovoj fazi projektanti treba da izvrše analizu funkcija predloženog sistema, na osnovu zahteva iz prethodne faze. Istražuju se mogućnosti i tehnike putem kojih će se implementirati zahtevi korisnika. Ako neki od zahteva nije izvodljiv, o tome treba informisati korisnika. Pronalazi se odgovarajuće rešenje i u skladu sa tim se ažuriraju dokumenti. U ovoj fazi se pravi dokument sa specifikacijm softvera, koji služi kao vodič kasnije prilikom kodiranja. Ovaj dokument sadrži organizaciju sistema, strukturu menija, strukturu podataka itd. U ovoj fazi se pripremaju i dokumenti za testiranje sistema.

Metodologije razvoja softvera

V model

Projektovanje arhitekture sistema je sledeća faza. Ova faza predstavlja projektovanje na najvišem nivou. Tu treba izabrati arhitekturu koja je u stanju da ostvari zahteve u zadatom roku, sa zadatom cenom i resursima. Obično se bira između arhitekture u dva nivoa, tri nivoa ili više nivoa. Ovi modeli se obično sastoje od baze podataka, korisničkog interfejsa i poslovne logike.



Sledeća faza je **projektovanje modula**. Ovo je projektovanje na nižem nivou. Sada se projektovani sistem deli na manje jedinice i svaka od njih se detaljno objašnjava tako da programeri mogu da počnu da kodiraju.

Metodologije razvoja softvera

V model

Testiranje počinje **testiranjem pojedinih softverskih jedinica**. Greške koje se ovde otkriju se mnogo jeftinije ispravljaju nego ako se otkriju u toku upotrebe sistema. Ova faza obuhvata analizu napisanog koda i otklanjanje grešaka.

Tu se proverava i da li je kod efikasan i da li je u skladu sa prihvaćenim standardima kodiranja. Testiranje se radi na osnovu testova koji su pripremljeni u fazi projektovanja modula. Testove izvršavaju posebni ljudi ili programeri.

U fazi testiranja integracije se proverava interfejs pojedinih modula i komunikacija između pojedinih komponenti. Ovo se radi na osnovu testova pripremljenih u fazi projektovanja arhitekture.

U fazi testiranja sistema se vrši poređenje specifikacije sistema i realizovanog sistema.
Na kraju se vrši i testiranje od strane korisnika. Ovde se testiranje vrši sa realnim podacima, rade ga ljudi koji će kasnije i koristiti sistem. Izvršavaju se testovi koji su napravljeni ranije. U ovoj fazi se pregleda i korisnička dokumentacija.

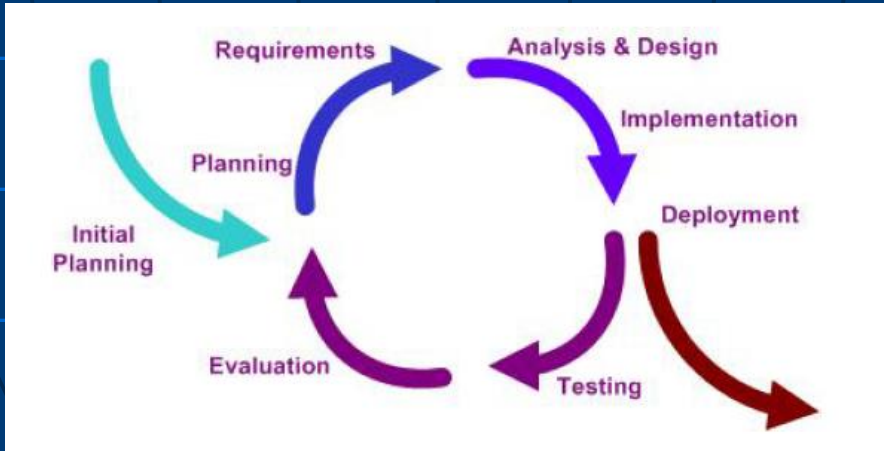
Metodologije razvoja softvera

Iterativni model

Iterativni i inkrementalni razvoj je ciklični model razvoja softvera, nastao kao odgovor na slabosti kaskadnog modela. Inkrementalni razvoj je strategija definisanja termina i faza kod koje se različiti delovi sistema razvijaju u različito vreme i integrišu se nakon završetka.

Iterativni razvoj je je strategija prepravki kod koje se odvaja posebno vreme za pregled i poboljšanje delova sistema. Nije obavezno da se iterativni razvoj koristi zajedno sa inkrementalnim razvojem, ali se najčešće koriste zajedno. Osnovna razlika je u tome da se izlaz iz inkrementa šalje korisniku, dok se izlaz iz iteracije ispituje radi poboljšanja.

Osnovna ideja koja stoji iza iterativnog razvoja je da se softver razvoja malo pomalo, tako da programeri mogu da iskoriste ono što su naučili tokom razvoja prethodne verzije. Informacije dolaze kako iz procesa razvoja, tako i iz upotrebe sistema, ako je to moguće. Osnovni koraci su početak projekta sa jednostavnom dokumentacijom i samo podskupom zahteva, nakon čega se to malo pomalo proširuje, sve dok se ne implementira ceo sistem. U svakoj iteraciji se menja i dopunjuje projekat, odnosno dodaju se nove funkcije.



Metodologije razvoja softvera

Iterativni model

Prilikom projektovanja iterativnog modela na umu treba imati sledeće:

• Sve poteškoće u projektovanju, kodiranju i testiranju signaliziraju da je potrebno popravljati projekat.
• Sve promene treba da se lako ubacuju u module koji su odvojeni i koje je lako pronaći. Ako to nije slučaj, potrebno je menjati projekat.
• Promene u tabelama u bazi podataka treba da je posebno lako napraviti. Ako to nije slučaj, treba menjati bazu.
• Promene treba da budu sve lakše kako proces iteracije napreduje.
• Zakrpe su dozvoljene samo za jednu ili dve iteracije. Može biti da su one neophodne da bi se tokom implementacije izbeglo ponovno projektovanje.
• Postojeća implementacija treba da se stalno analizira da bi se odredilo koliko dobro zadovoljava ciljeve projekta.
• Treba analizirati i reakciju korisnika na trenutnu imlpementaciju.

Iterativni razvoj predlaže prilagodljivo planiranje zasnovano na 2 vrste planova: fazni plan i serija iterativnih planova. Fazni plan je jedan a iterativnih ima mnogo više. U svakom trenutku definisana su 2 iterativna plana, za tekuću i narednu iteraciju. Kako se korisnički zahtjevi i arhitektura stabilizuju, procjenjuje brzina razvoja i tim sazrijeva, moguće je planirati 2-3 iteracije unapred.

Metodologije razvoja softvera

Iterativni model

Iterativni model propisuje **ko, kako, šta i kada radi** da bi se postigao određeni cilj. Koristi 5 osnovnih elemenat:

- **Uloga (role) – ko**
- **Aktivnost (activity) – kako**
- **Artefakt (artifact) – šta**
- **Tok posla (workflow) – kada**
- **Disciplina (discipline) – skladište 4 predhodno pomenuta elementa**

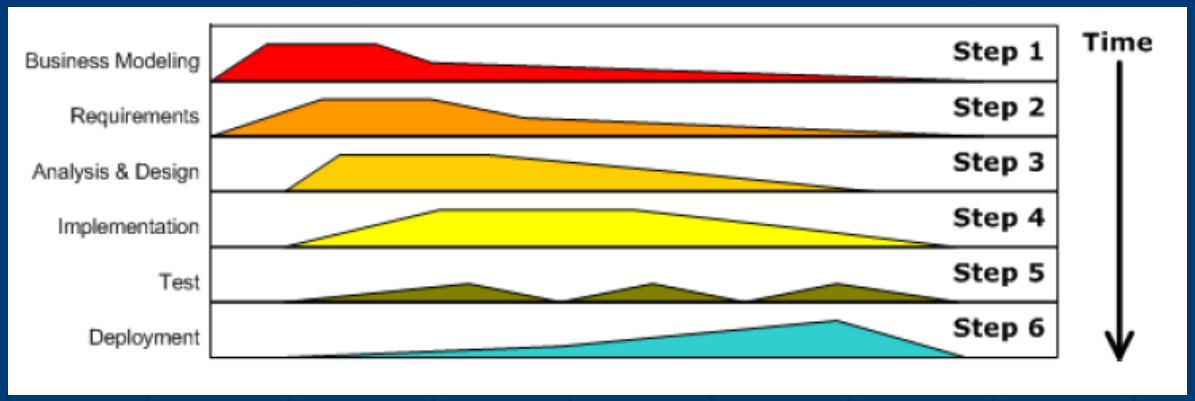
Organizovan je u **4** sekvencijalne faze:

- Početak, ili uvodna, pripremna, inicijalna faza (***inception***)
- Elaboracija, ili faza razrade (***elaboration***)
- Konstrukcija, ili faza izgradnje (***construction***)
- Tranzicija, ili prelazna faza (***transition***)

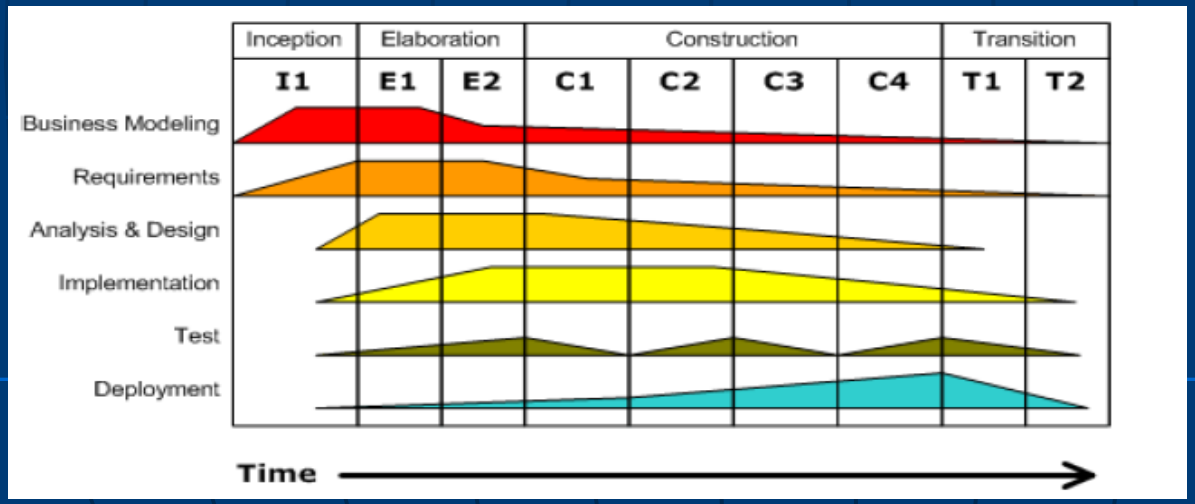
Cilj svake iteracije je implementacija, integracija i testiranje odabranog skupa korisničkih zahteva (funkcionalnih i/ili nefunkcionalnih), kao i kreiranje pratećih artifakata. Svaka iteracija se završava mini prekretnicom u kojoj se na bazi kriterijuma uspeha ocenjuje uspešnost iteracije.

Metodologije razvoja softvera

Upoređenje kaskadnog i iterativnog pristupa razvoju softvera:



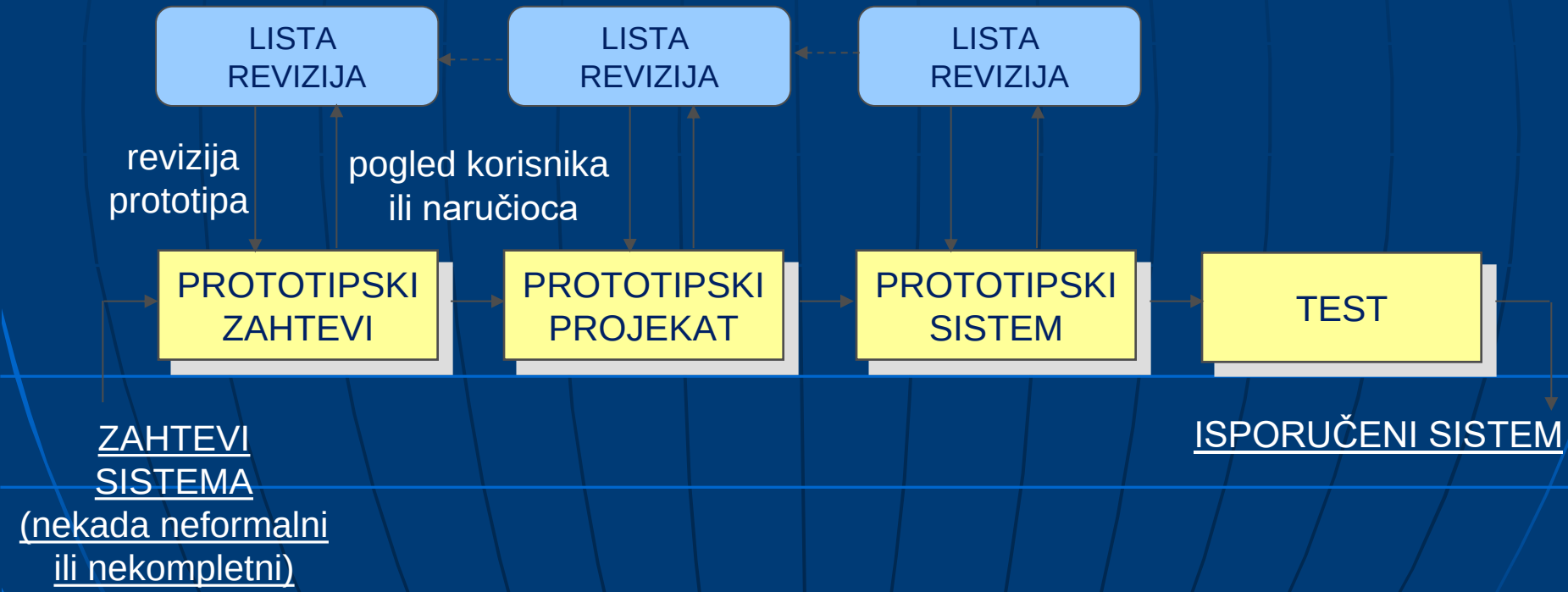
Kod kaskadnog pristupa se svaka faza završava u potpunosti, pa se tek onda prelazi na sledeću. Isporuka se vrši odjednom, vrlo blizu kraju celog projekta.



Kod iterativnog razvoja se funkcije sistema koje se isporučuju dele u iteracije. U svakoj iteraciji se isporučuje po neki deo funkcija. Kod unificiranog procesa se iteracije grupišu u faze početka, razrade, konstruisanja i prelaza.

Metodologije razvoja softvera

Model prototipskog razvoja se koristi da bi se za potrebe korisnika razvio inicijalni model budućeg softvera koji simulira njegove stvarne funkcije sa ciljem da korisnik da svoje mišljenje i odluči koji i kakvi su njegovi zahtevi prototip može biti takođe iskorišćen kao koncept unutar
omogućuje da se brzo izgrade primitivne verzije softvera



Metodologije razvoja softvera

Model prototipskog razvoja se najčešće upotrebljava i daje solidne rezultate u situacijama:

kada su od strane korisnika samo *uopšteno definisani ciljevi* razvoja softverskog proizvoda, ali ne i detalji u pogledu ulaza, procedura i izlaza,
kada je *moguće simulirati rad softvera* da bi korisnik mogao videti kako će budući softverski proizvod funkcionisati i
kada same razvojne organizacije žele proveriti efikasnost algoritama ili adaptibilnost sistema

Model uobičajeno može imati tri oblika:

- ✓ *prototip u obliku papira* koji opisuje vezu čoveka i mašine na način da korisniku omogući razumevanje tog odnosa,
- ✓ *radni prototip* koji implementira neke od funkcija postavljenih kao zahtevi softverskom proizvodu ili
- ✓ *realni program* koji izvršava deo ili celinu zahtevanih funkcija

Metodologije razvoja softvera

Model prototipskog razvoja

Korisnik uočava radnu verziju softvera neznajući na koji su način delovi softvera međusobno povezani, *neznajući da u brzini realizacije nisu razmatrani aspekti kvaliteta ili održavanja u dužem vremenskom periodu*

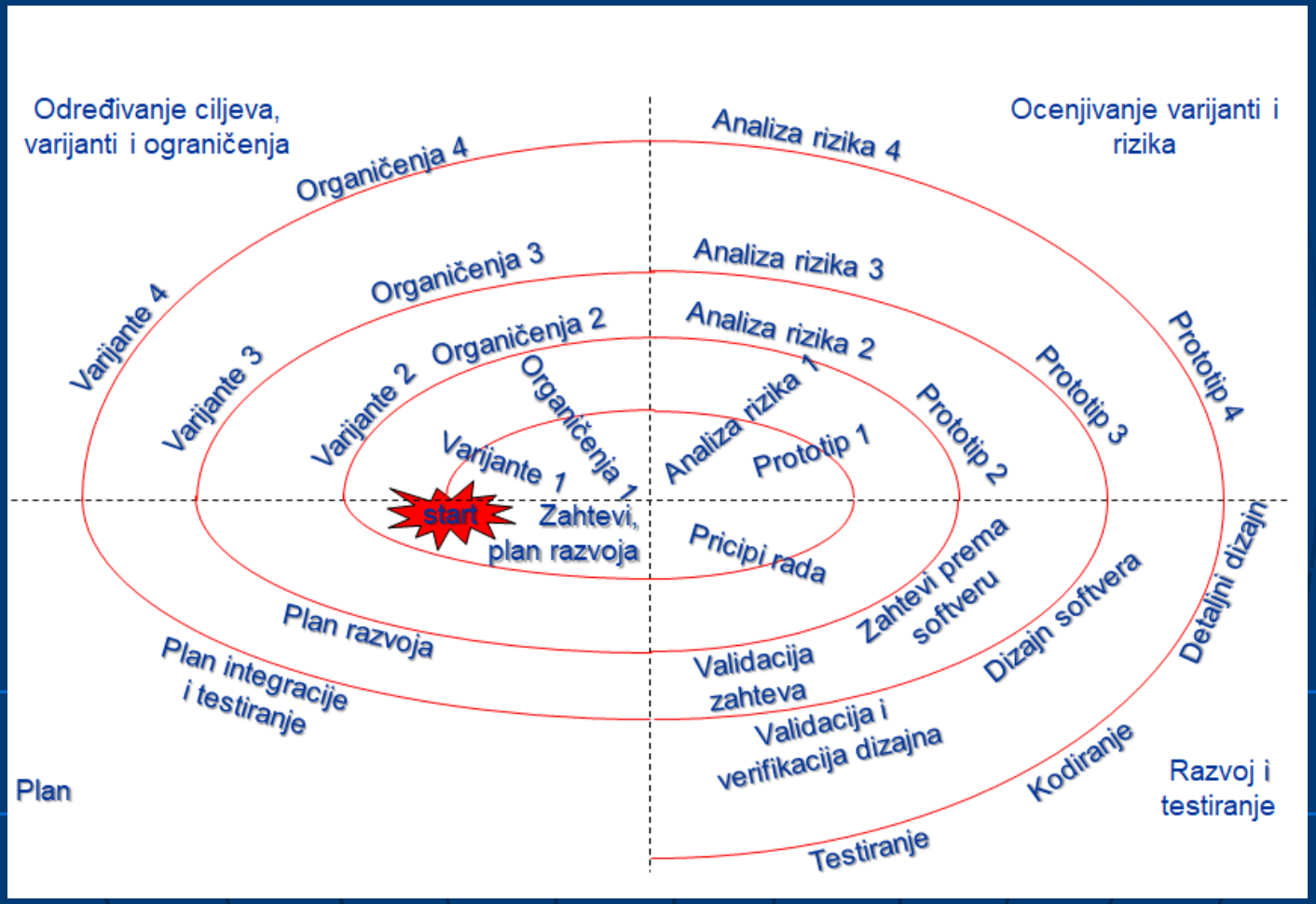
Kada dođe do informacija da je potrebno izvršiti "remont" ili dalju dogradnju još ne uvedenog softverskog proizvoda, *korisnik se oseća prevarenim* i insistira da se putem izvesnih intervencija brzo realizuje njemu potreban proizvod. Upravljanje razvojem softvera u ovakvim situacijama postaje nekontrolisano.

Projektant često čini kompromise u implementaciji sa ciljem da izgrađeni prototip što pre stavi u funkciju.

Neadekvatan operativni sistem ili programski jezik se jednostavno upotrebljavaju samo zato što su raspoloživi ili poznati; neefikasan algoritam se primenjuje samo da bi se demonstrirala sposobnost softvera.
Nakon izvesnog vremena, zaboravlja se na način odabira i njihove uzroke, pa manje idealna rešenja ili bolje rečeno *manje kvalitetna rešenja ostaju integralni deo konačnog softverskog rešenja.*

Metodologije razvoja softvera

Spiralni model



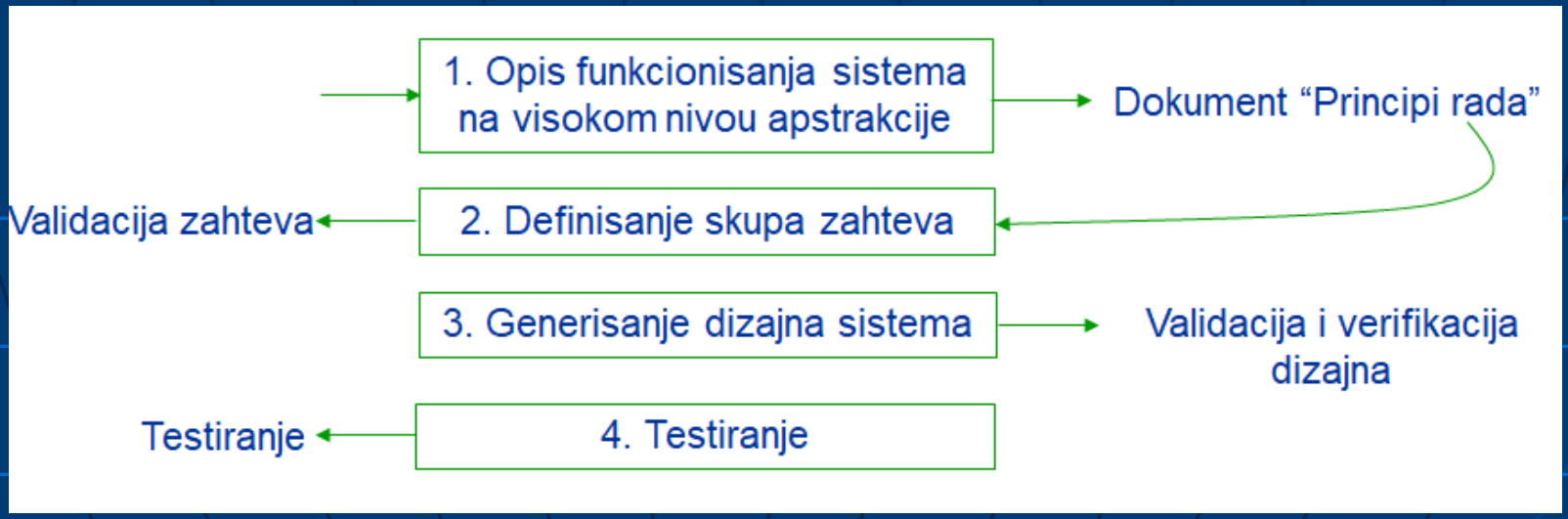
Metodologije razvoja softvera

Spiralni model se predstavlja spiralom na kojoj su definisane četiri faze razvoja:

planiranje – faza koju čine aktivnosti odrenivanja ciljeva, alternativa i ograničenja,
analiza rizika – faza koju čine aktivnosti analize alternativa i identifikovanja rizika,
inženjering – faza razvoja novih nivoa proizvoda i
razvoj i ocenjivanje – faza procene rezultata inženjeringa.

Posmatranjem spirale, svakom iteracijom se progresivno razvijaju kompletnije i potpunije verzije softvera. Tokom prvog ciklusa kretanja spiralom, prikupljaju se zahtevi i planira projekat razvoja, da bi se izvršila analiza rizika inicijalnih zahteva

Zahtevi i plan razvoja



Metodologije razvoja softvera

Model zasnovan na komponentama

Osnovni pristup u ovom modelu je konfigurisati i specijalizovati već postojeće komponente softvera u novi aplikativni sistem

Višestruko korišćenje softvera je proces uključivanja u novi proizvod pojedinih komponenti:

prethodno testiranog koda,
prethodno proverenog dizajna,
pretnodno razvijene i korišćene specifikacije zahteva i
prethodno korišćenih procedura za testiranje

Koristi koje sobom donosi ponovno korišćenje komponenti razvijenog softvera su sledeće:

- ✓ podiže robustnost softvera,
- ✓ povećava produktivnost izrade softvera,
- ✓ povećava kvalitet softvera,
- ✓ smanjuje troškove razvoja softvera,
- ✓ štedi odnosno skraćuje vreme izrade,
- ✓ zadovoljava ciljeve softverskog inženjeringa,
- ✓ širi korišćenje softvera,
- ✓ obezbenuje adekvatnu dokumentaciju,
- ✓ olakšava održavanje softvera,
- ✓ modelira sistem za lakše razumevanje i dr.

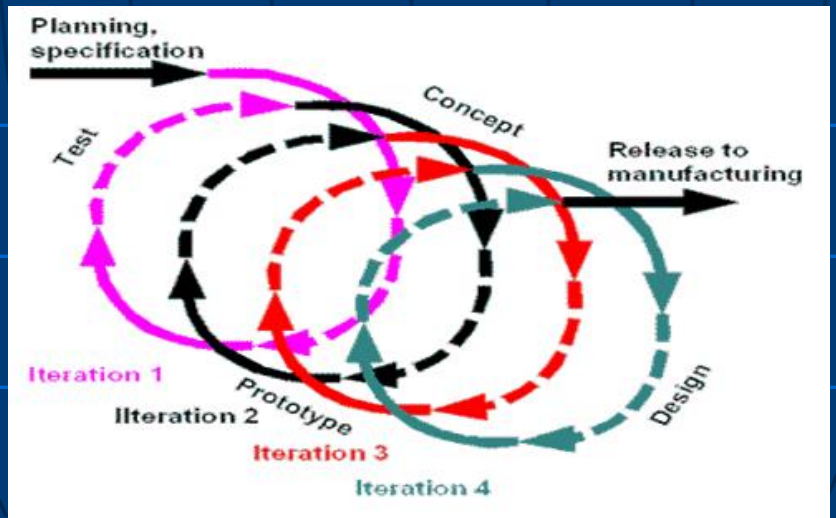
Metodologije razvoja softvera

Model unificiranog procesa razvoja

Ovaj model opisuje proces razvoja korišćenjem UML - objedinjenog jezika za modelovanje u objektno-orijentisanom razvoju

Prema autorima, model se može opisati kao proces klasificiranja iteracija, koje se mogu podeliti u 4 grupe:

- ✓ u prvoj grupi se nalaze početne iteracije interakcija sa stekholderima, tj. značajnim učesnicima u razvoju softvera,
- ✓ drugu grupu sačinjavaju razranene iteracije želja i potreba korisnika,
- ✓ iteracije konstruisanja inicijalnih operacionih mogućnosti sačinjavaju treću grupu i
- ✓ prelazne iteracije kompletiranja proizvoda su četvrta, konačna iteracija razvoja softvera prema ovom modelu.

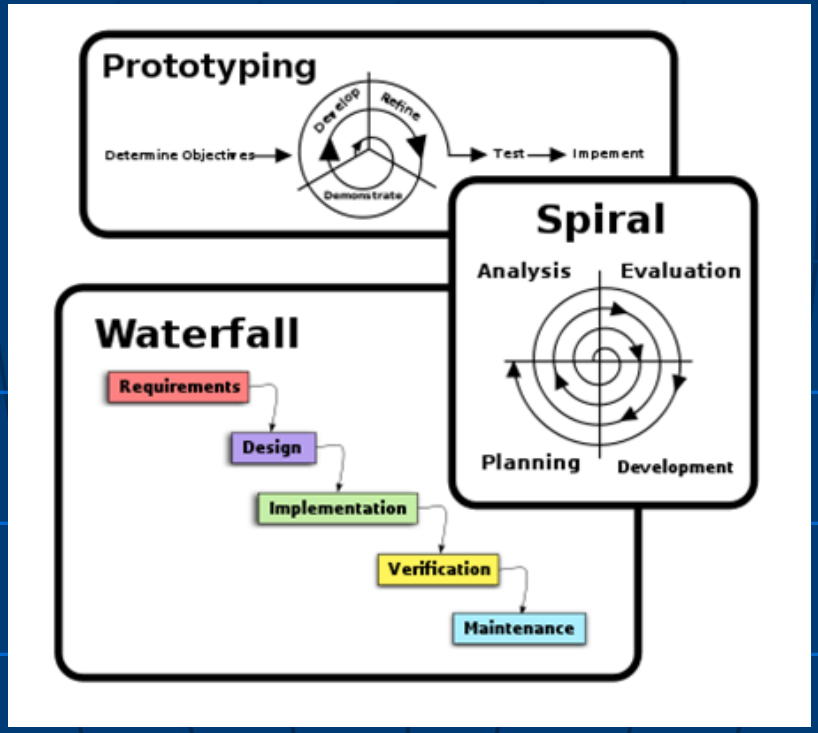


Metodologije razvoja softvera

Kombinovani model

U mnogim situacijama modeli se mogu kombinovati tako da se postignu prednosti od svih na samo jednom projektu.

Spiralni model je i sam primer dobre kombinacije dva modela, ali i drugi modeli mogu poslužiti kao osnova na koju će se integrisati neki modeli.



Upravljanje softverskim projektima

Dat je opis do sada korišćenih metoda upravljanja softverskim projektima. Njihove prednosti i kritika proizišla iz teorije i prakse. Odnosi se na klasične, tradicionalno korišćene metode kroz reprezentativne predstavnike **code-fix, waterfall i rational unified prosess (RUP).**

Naglim razvojem industrije bio je praktično nemoguć uz metode i alate koji su se do tada koristili.

Najveći broj razvijenog softvera potiče od haotiččne aktivnosti, često označene frazom kodiraj i popravi (**code-fix**).

Proceduralni jezici i code-fix metode razvoja dale su svoj maksimum. Morale su se stvoriti metode koje je investitor sam mogao upravljati i kontrolisati.To su metode poznate pod nazivom: **plansko-orijentisane i planskocentrične.**

METODE AGILNOG RAZVOJA SOFTVERA

Opis i istorijat agilnih metoda razvoja softvera, pre svega njihovo poređenje sa klasičnim metodama opisanim u predhodnom poglavlju, zatim prevod tzv. “manifesta agilne metodologije” sačinjen od njenih tvoraca i kratak komentar o svim pojedinačnim, do sada razvijenim, metodama agilnog programiranja uz poseban osvrt na one metode koje su posebno važne za ekstremno programiranje.

Agilni razvoj softvera potiče iz 90-tih godina i to kao negativne reakcije na ,,robusne”metode prije svega na **„Vaterfall”** model. Agilni naziv usvojen je 2001.godine.Najpoznatije metode agilnog softverskog razvoja danas su:ekstremno programiranje, industrijsko ekstremno programiranje,scrum, agilno modeliranje, adaptivni razvoj softvera, crystal clear metode, metod dinamičkog razvoja sistema, razvoj vođen karakteristikama, ,,suvi razvoj”, agile unified process.

Agilne metode akcentuju realnu komunikaciju i to **preko komunikacije licem u lice** i **„živu reč”** u **odnosu na pisanu dokumentaciju.**

METODE AGILNOG RAZVOJA SOFTVERA

Agilne metode predstavljaju koncept u razvoju softvera koji propagira iteracije u razvoju. Postoji više agilnih metoda. Zajedničko za sve njih je da se smanjuje rizik kod razvoja softvera, time što se skraćuje vreme razvoja. Ovde se pod iteracijom podrazumeva softver proizveden u jednom vremenskom intervalu. Vremenski interval može biti od jedne do četiri nedelje. Svaka iteracija predstavlja celokupan softverski projekat, što znači da sadrži planiranje, analizu zahteva, projektovanje, kodiranje, testiranje i dokumentovanje. Jedna iteracija možnda neće dati dovoljno funkcija da se proizvod iznese na tržište, ali je cilj da se na kraju svake iteracije napravi nešto što može da se implementira kod korisnika (bez bagova). Na kraju svake iteracije tim ponovo definiše prioritete u projektu.

Agilne metode daju prednost komunikaciji sa korisnicima, nad pisanim dokumentima. Većina timova se nalaze u jednoj kancelariji. Tu se nalaze i programeri i njihovu „kupci“ (kupci su ti koji definišu proizvod i mogu biti menadžeri, poslovni analitičari ili klijenti). U kancelariji se mogu još nalaziti oni koji testiraju programe, ljudi zaduženi za integraciju itd.

Agilne metode takođe ističu da je osnovna mera napretka u projektu softver koji se koristi. U kombinaciji sa komunikacijom lice u lice, agilne metode proizvode vrlo malo pisane dokumentacije, u odnosu na druge metode. Iz ovog i proizilaze kritike agilnih metoda da tu nema discipline. Savremena definicija agilnih metoda je nastala 1990-ih godina, kao odgovor na tadašnje metode razvoja softvera (kaskadni model). Proces je nastao kao reakcija na probleme sa kaskadnim modelom i činjenicu da taj model nije pratio način na koji softverski inženjeri zaista rade.

METODE AGILNOG RAZVOJA SOFTVERA

2001-e godine su se sastale vodeće ličnosti u ovoj oblasti i napravile manifet agilnih metodologija. U ovom manifestu su definisani osnovni principi razvoja na ovaj način. Neki od najvažnijih principa su:

- ✓ • Kupac mora biti zadovoljan. To se obezbeđuje time što se softver isporučuje brzo i stalno.
- ✓ • Često se isporučuje softver koji radi (nedelje su u pitanju, a ne meseci)
- ✓ • Softver koji radi je mera napretka.
- ✓ • Naknadne promene u zahtevima su čak i dobrodšle.
- ✓ • Svakodnevna, bliska saradnja između ljudi iz preduzeća i programera.
- ✓ • Najbolji oblik komunikacija je licem u lice.
- ✓ • U projekte su uključeni motivisani ljudi, kojima treba verovati.
- ✓ • Stalna pažnja se obraća na dobar dizajn i upotrebu najnovijih tehnologija.
- ✓ • Jednostavnost
- ✓ • Timovi koji se sami organizuju

Agilne metode

Agilne metode (Agile Alliance, 2001.) su nastale kao otpor mnogim ranijim modelima razvojnog procesa koji su pokušavali da nametnu neki oblik discipline vezane za osmišljavanje softvera, dokumentovanje, razvoj i testiranje. Ideja je da se naglasi uloga fleksibilnosti u spretnom brzom razvoju softvera.

4 principa agilnog razvoja

Za uspešnost projekta najvažniji su **kvalitet ljudi** koji na njemu rade i kvalitet njihove saradnje. Postupci i alati imaju sporedni značaj.

Umesto planiranja i praćenja plana, važnije je **odgovarati na promene**, jer se ne mogu svi zahtevi predvideti na početku razvoja.

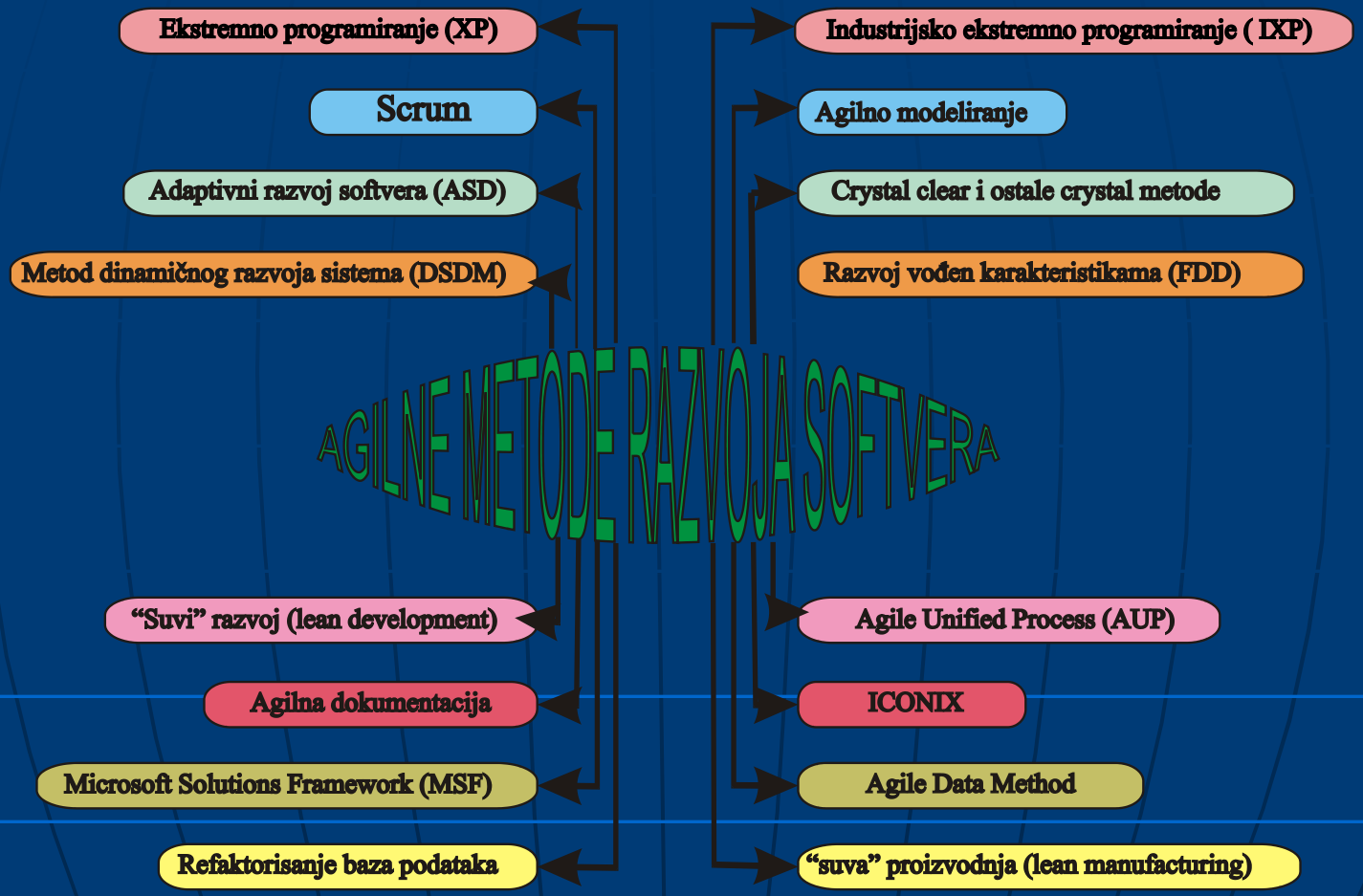
Bolje je uložiti **vreme u izradu softvera** koji radi, nego u izradu sveobuhvatne dokumentacije.

Zajednički **rad sa naručiocem** je vrlo važan, jer se tako naručilac uključuje u ključne aspekte razvoja.

METODE AGILNOG RAZVOJA SOFTVERA

- Kasnih devedesetih godina prošlog veka, grupa naučnika (Agile Alliance 2001) je pokušala da promeni do tada čvrsto ustaljeni način i disciplinu na koji se softver osmišljava, dokumentuje, razvija i testira. Oni su, naglašavajući ulogu koju fleksibilnost može da odigra u spretnom i brzom razvoju softvera, formulisali sopstvene principe ogledavajući ih u takozvanom ” agilnom manifestu ” :
  - Osobe i interakcije umesto procesa i alata
  - Softver koji radi umesto opširne dokumentacije
  - Zajednički rad s kupcem umesto pregovaranja preko ugovora
  - Reagovanje na promene umesto striktnog pridržavanja plana
- Na početku razvoja ove metode su se nazivale lakim metodama, da bi tek kasnije dobile ime koje i danas poseduju – AGILNE METODE RAZVOJA SOFTVERA

METODE AGILNOG RAZVOJA SOFTVERA



METODE AGILNOG RAZVOJA SOFTVERA

- Agilne metode u razvoju softvera javljaju se kao odgovor na opsežne (tradicionalne) metodologije. Tako iz sledeće tabele možemo utvrditi osnovne razlike između ”starog ” načina razvoja softvera, npr. poznatog tipa vodopada (Waterfall) i agilnih metoda.

Waterfall	Agilne metode
Puno dokumentacije	Verbalna komunikacija
Planiranje unapred	Postepeno planiranje Evolutivni dizajn
Mala fleksibilnost	Velika povratna sprega
Isporuka u jednom trenutku	Fleksibilnost isporuke
Kontrola	Automatizovano testiranje

METODE AGILNOG RAZVOJA SOFTVERA

- Agilne metode, označavaju spremnost na pokret, aktivnost, žestinu, brzinu razvoja softvera pokušavajući da ponude odgovor na želju klijenata za manje robusnim metodologijama razvoja softvera, koje sa sobom donose brže, žešće i kvalitetnije procese razvoja.
- Agilne metode razvoja softvera u softverskom inženjerstvu su manje opsežne metode prihvatajući činjenicu da je softver teško kontrolisati. Zato su, inženjeri koji razvijaju softver fokusirani na male jedinice posla, minimizirajući rizik za pravljenje grešaka. To je naročito važno sa današnjim rapidnim rastom industrije vezane za telekomunikacije, Internet i sa okolinom vezanom za mobilne aplikacije sa distribuiranim razvojem.
- Agilne metode imaju značajne razlike u odnosu na ranije planskocentrične inženjerske metode.Najuočljivija razlika je insistiranje na manje obimnoj dokumentaciji za dati problem.Umjesto dokumentacionoorjentisane agilne metode su prije orijentisane ka izvornom kodu kao ključnom dijelu dokumentacije.

METODE AGILNOG RAZVOJA SOFTVERA - karakteristike:

- Mogu se definisati sledeće karakteristike agilnih softverskih procesa sa aspekta brze isporuke, koje omogućuju skraćivanje životnog ciklusa softverskih projekata:

  - ✓ modularnost stepena razvojnog procesa
  - ✓ kratke iteracije koje omogućavaju brze verifikacije i korekcije
  - ✓ vremenski okvir iteracija je od jedne do šest nedelja
  - ✓ uklanjanje svih nepotrebnih aktivnosti
  - ✓ prilagodljivost s mogućim nenadanim rizicima
  - ✓ inkrementalni pristup procesu koji omogućava funkcionalnu izgradnju softverskog produkta u malim koracima
  - ✓ konvergentni i inkrementalni pristup minimizira rizike
  - ✓ ljudski-orijentisani agilni proces favorizuje ljude iznad procesa i tehnologija
  - ✓ kolaborativni i komunikativni radni stil.

Ekstremno programiranje (XP)

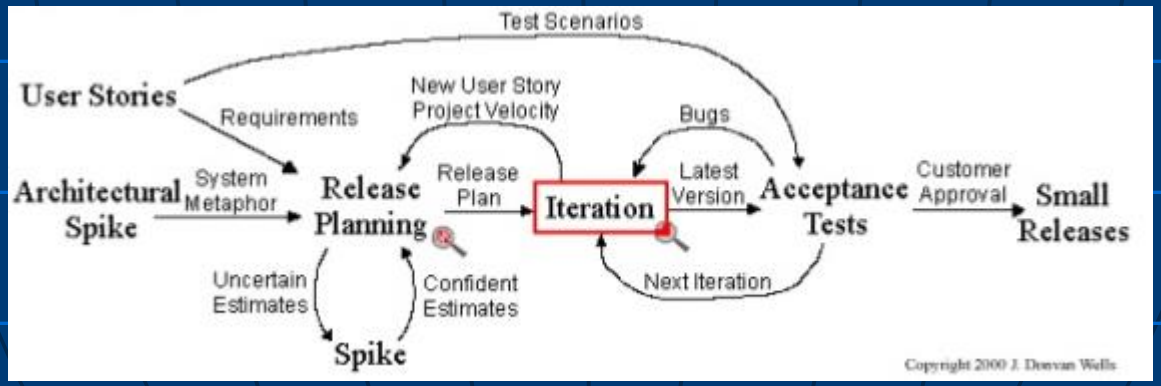
- Ekstremno programiranje se počelo koristiti u prvoj polovini devedesetih godina prošlog veka sa vrlo disciplinovanim pristupom razvoju softvera.
- Ekstremno programiranje je metodologija softverskog inženjerstva, zasnovana na vrednostima povratne sprege, komunikacije, jednostavnosti, odvažnosti i poslovanja.
- Pokretač ove nove metodologije je bio *Kent Beck*, koristeći nove koncepte razvoja softvera. Rezultat je bio metodologija ekstremnog programiranja razvoja softvera koja primenjuje skup 12 veština koje se mogu izdvojiti u 4 oblasti:

Ekstremno programiranje (XP)

U ovom poglavlju se detaljnije osvrćemo na metodu ekstremnog programiranja kao, kako se sada čini, najznačajniju od svih agilnih metoda, na njen nastanak i detalje koji je definišu. Naučno gledano, ova metoda se može posmatrati i kao jezik uzora (*pattern lenguage*) sa čitavim nizom projektnih uzora koje uklapa u jednu celinu što isključuje upotrebu pojma "metoda".

Međutim, ovde će se koristiti ovaj pojam i zanemariti viđenje kroz uzore što je za sada takođe validan pristup. Poseban akcenat je stavljen na obavezne prakse ekstremnog programiranja koje čine njegove osnovne gradivne jedinice, u smislu upravljanja razvojem softvera, koje je potrebno praktikovati ekstremno disciplinovano.

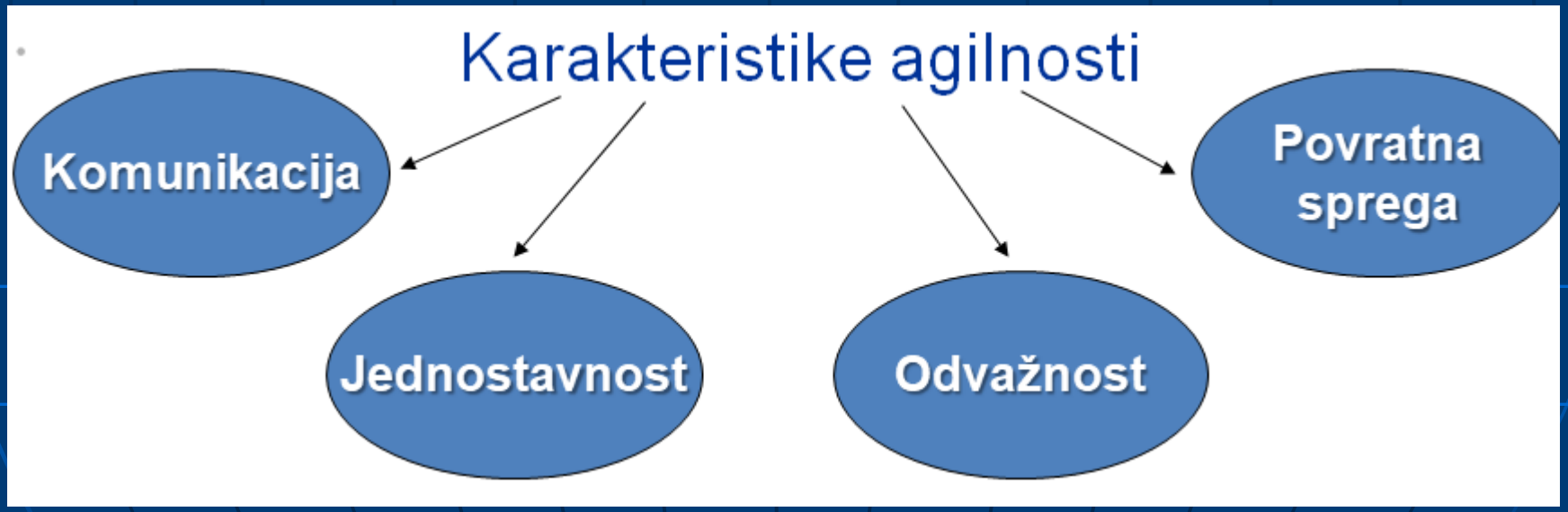
Zatim sagledavamo ponovo **XP** kao cjelinu uz posebnu analizu evolutivnog dizajna i njegovog uticaja na shvatanje dizajna uopšte, primjenu UML i softverskih uzora. Nakon toga sagledavamo pogodne organizacione modele koji mogu biti pogodni za primjenu u ovakvim uslovima rada, kao i posebno sagledavanje važnosti tima i timskog rada u **XP**.



Ekstremno programiranje (XP)

Ekstremno programiranje, XP (eXtreme Programming), (Beck, 1999.) predstavlja skup tehnika za omogućavanje kreativnosti projektnog tima uz minimizovanje prekomernog administriranja

- ✓ **Zasnovana je na** jednostavnosti, komunikaciji, feedbacku i hrabrosti.
- ✓ **Timovi koriste jednostavnu formu** planiranja i praćenja kako bi odlučili šta sledeće da rade.
- ✓ **Timovi kreiraju softver u seriji malih**, potpuno integrisanih izdanja koja su prošla sve testove koje je korisnik definisao.



Ekstremno programiranje (XP)

Ekstremno programiranje (XP)

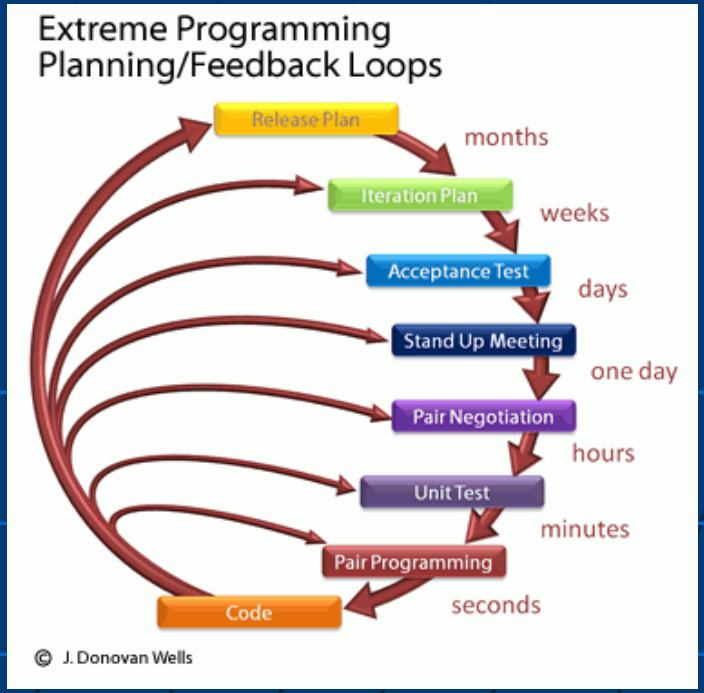
- **POVRATNA SPREGA**
- ✓ Programiranje u paru (engl. *Pair Programming*). Ovaj princip znači da uvek dva čoveka pišu određeni kod.
- ✓ Igra planiranja (engl. *Planning game*). Korisnička interakcija u programerskom (implementacionom) timu između programera i korisnika oko procena implementacije pojedinih funkcionalnosti projekta.
- ✓ Testiranje (engl. *Testing*). Kontinualno, često ponavljajuće automatizovano testiranje jedinice (engl. *unit*) i regresiono (engl. *regression*) testiranje.

- **KONTINUALNOST PROCESA**
- ✓ Kontinualna integracija (engl. *Continuous integration*). Novi kod se integriše u sistem čim je spreman (implementiran i testiran).
- ✓ Korišćenje tehnike refaktorisanja (engl. *Refactoring*). Uklanjanje dvostrukog (redundantnog) koda i održavanje koda jednostavnim.
- ✓ Male česte isporuke (engl. *Small/short releases*). Sistem se brzo i često isporučuje, najmanje svaka 2 do 3 meseca. Ovaj pristup se temelji na praksi iterativnog i inkrementalnog razvoja.
- ✓ Povratna informacija od kupca (korisnika) (engl. *On-site customer*).Korisnik je stalno na raspolaganju programerima.

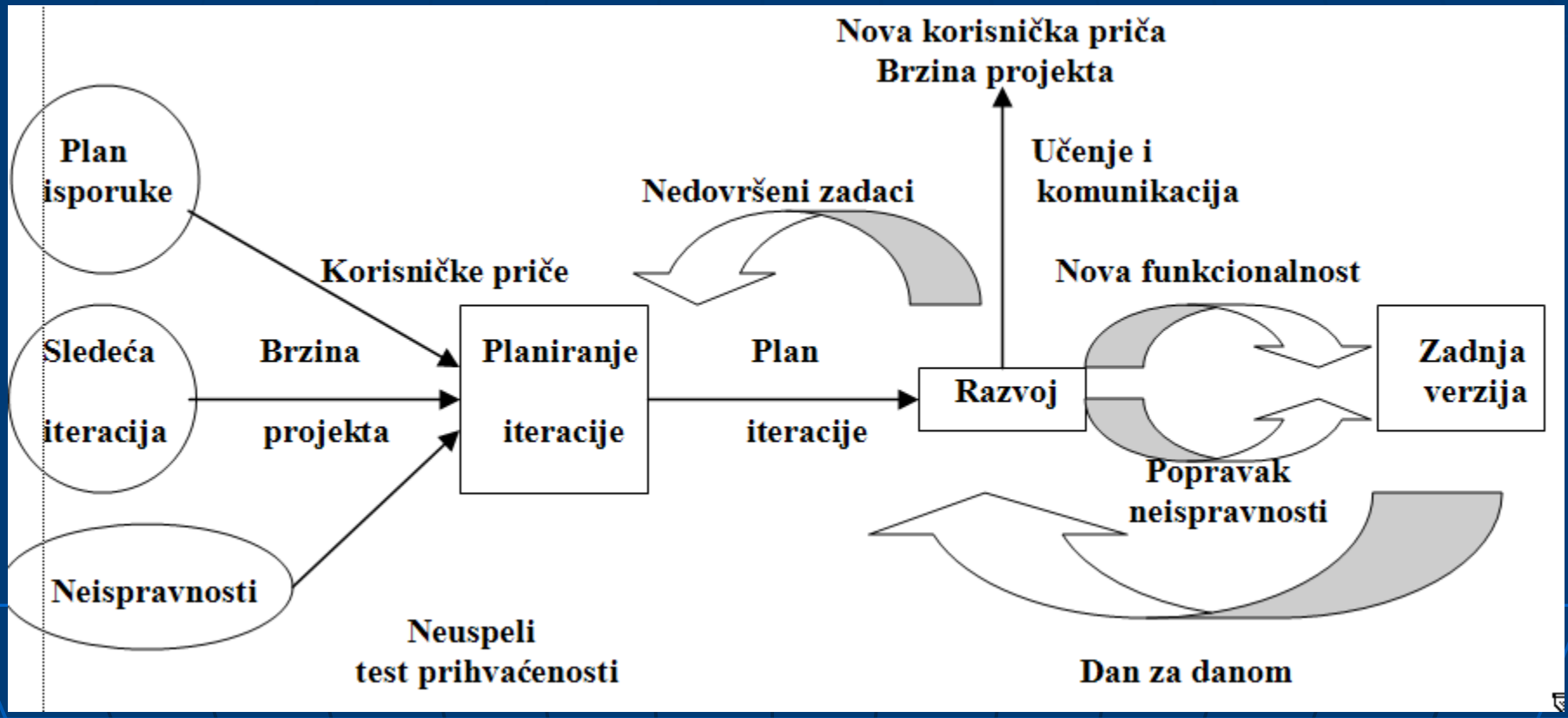
Ekstremno programiranje (XP)

- **DELJENO RAZUMEVANJE**
  - ✓ Standardi kodiranja (engl. *Coding Standards*). Postoje standardi kodiranja i programeri ih slede kako bi kod na kojem se trebaju napraviti bilo kakve izmene, a koga je pisao neko drugi u timu, bio razumljiviji.
  - ✓ Zajedničko deljenje koda (pristup kodu) (engl. *Collective Ownership*). Bilo ko iz tima sme mijenjati bilo čiji kod.
  - ✓ Jednostavan dizajn (engl. *Simple Design*). Naglasak je na dizajnu najjednostavnijeg rešenja koji je zahtevan u tom trenutku, bez dodatnog koda i viška funkcionalnosti.
  - ✓ Organizacija sistema sa metaforama (engl. *Methaphor*). Metafora je pojednostavljena slika sistema u razvoju.

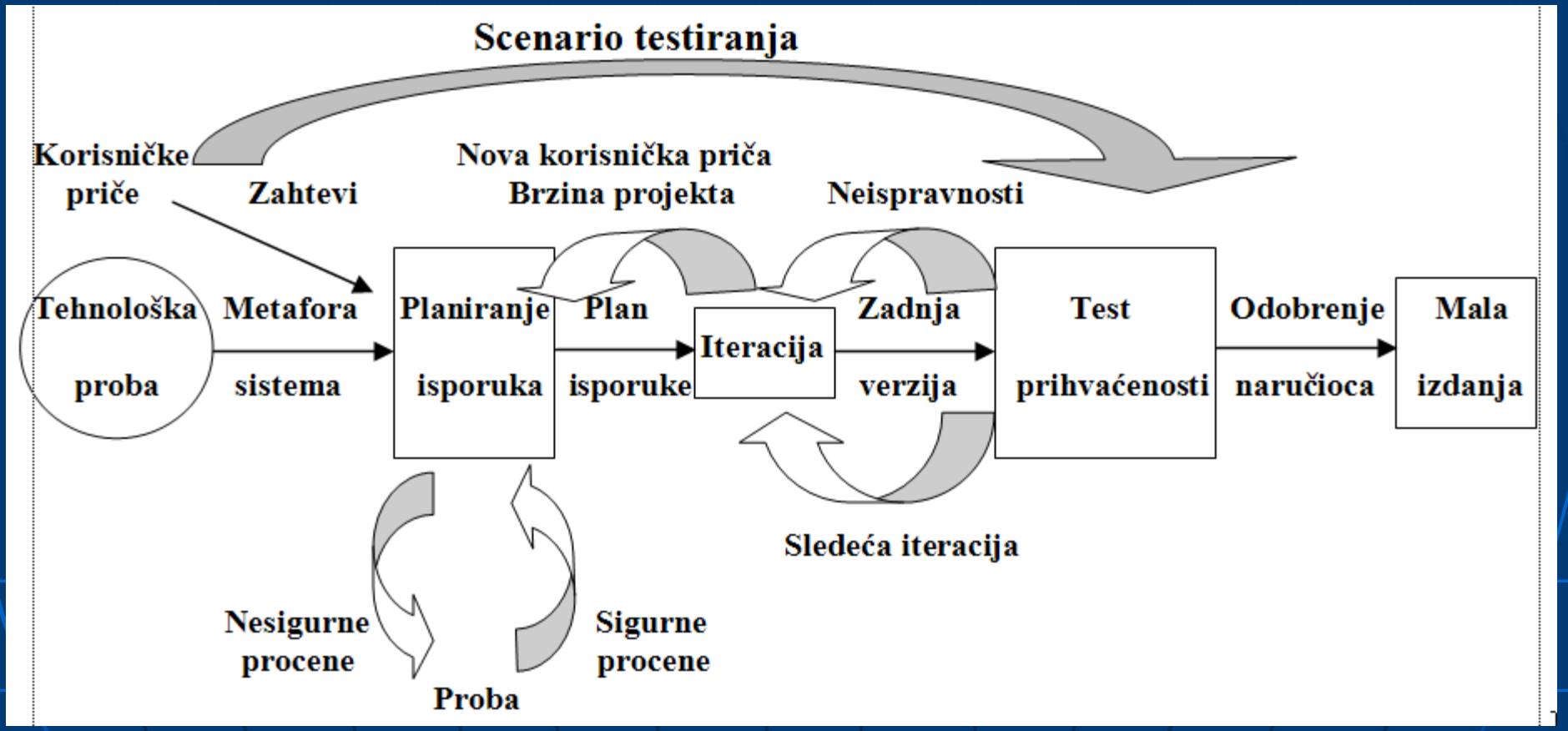
- **DOBROBIT PROGRAMERA**
  - ✓ Održivi korak (engl. Sustainable Pace). Četrdeset - satna radna nedelja (engl. *40-hours week*). Maksimum je 40-satna radna nedelja. Nisu poželjni uzastopni prekovremeni dani zbog slamanja timskog duha.



Iterativni postupak u procesu razvoja projekta



Proces razvoja projekta



Životni ciklus XP projekta

Ekstremno programiranje (XP)

Kao i vrednosti, i principi ekstremnog programiranja proizilaze iz manifesta agilne metodologije proširujući i konkretizujući njihovo značenje u svakom pojedinačnom projektu. Vrednosti nam pružaju osnovne kriterijume po kojima se sagledava uspešnost posla, ali su suviše široke da bi se na osnovu njih mogli definisati praktični mehanizmi za osiguranje uspešnosti. Sledeći konkretni principi dekomponuju definisane vrednosti kako bi se ovo postiglo. Oni otkrivaju različite alternative koje služe odlučivanju, sve vreme integrišući vrednosti u njih.

Principi	*Objašnjenje*
Brza povratna sprega	Umjesto da se na odgovor od klijenata čeka mjesecima, ili godinama, oni informaciju vraćaju u roku od nekoliko dana, ili nedjelja. Programeri u roku od nekoliko minuta, ili sekundi, saznaju da li je dizajn dobar i da li program radi, umjesto da na to čekaju danima, ili nedjeljama. To pojačava razumjevanje i učenje.
Pretpostavljena jednostavnost	Pretpostaviti da je svaki problem trivijalno jednostavan u rešavanju. Vrijeme koje se tako uštedi, daleko prevazilazi vrijeme koje se izgubi u rijetkim slučajevima kada ova premisa nije istinita.
Postepene promjene	Svaka velika promena se razbija u niz minimalnih koraka. Sve vrste promjena treba sprovoditi u malim pomacima–inkrementima.
Prihvatanje promjena	Najbolja strategija je da se rešavaju prvo gorući problemi, pa tek onda oni sitniji.
Kvalitetan rad	Niko ne voli da radi aljkavo, ili loše. Jedine moguće vrijednosti varijable kvaliteta su odlično i perfektno, ništa manje. U suprotnom se ne uživa u poslu, što izaziva pad performansi.

Ekstremno programiranje (XP)

Na kraju se sve svodi na konkretne aktivnosti koje se sprovode u timu i koje vode visokom nivou kvaliteta u što kraćem vremenu i uz minimalne troškove ostajući pritom u regiji cilja, odnosno u domenu rešenja. Aktivnosti su ono što proizilazi iz vrednosti i principa i važne su za definisanje ponašanja tima u određenim radnim situacijama i imaju smisla jedino kada odražavaju te vrijednosti i principe, a takođe moraju biti sprovedene na način striktno propisan obaveznim praksama. Aktivnosti ekstremnog programiranja su sledeće:

Aktivnosti	*Objašnjenje*
Kodiranje	Ovo je oblik komuniciranja i predstavlja preslikavanje ideja u softver. Izvorni kod je osnovni element kompjuterskog programa. Preko njega se materijalizuju ideje, ali i vrši čista i nedvosmislena komunikacija; kako sa mašinom, tako i sa kolegama. On može izražavati ideje za rješenje problema, ali i testove, takođe se može koristiti i za opis algoritama i ukazivati na moguća mjesta daljeg razvoja sistema. Najbolja mjera isravnosti je kôd koji radi.
Testiranje	Gotovi junit testovi bi trebalo da budu potreban i dovoljan uslov za početak kodiranja, jer se bez testa ne može znati kada je kodiranje završeno, niti koliko tačno je sprovedeno. Takođe, na osnovu funkcionalnih testova, klijent određuje pravac daljeg rada. Na taj način se poboljšava i komunikacija među klijentima i programerima.
Slušanje	Polazi se od tvrdnje da programeri neznaju ništa o poslovnoj logici, te oni moraju aktivno da slušaju klijenta. Aktivno slušanje se odnosi na ukazivanje klijentu šta je teže, a šta lakše izvesti. Iz *priče klijenta* mora se izvući što više korisnih informacija, koje će kasnije pomoći u razumijevanju sistema. Potrebna je česta komunikacija sa klijentom, kome se postavljaju pitanja i pažljivo slušaju odgovori.
Dizajn	U **XP** dizajn je evolutivne prirode. On teži jednostavnosti i smanjenju redudantnosti koda. Dijagrami i planovi nisu centralni elementi dizajna, već je to izvorni kod. Ovo je još jedna revolucionarna promjena u načinu razmišljanja programera.

Ekstremno programiranje (XP)

Obavezna praksa u ekstremnom programiranju predstavlja skup praktičnih mehanizama pomoću kojih se sprovode aktivnosti i realizuje razvoj softvera metodom ekstremnog programiranja.

Postoji dvanaest ovih mehanizama koji se sprovode ekstremno disciplinovano.

Mehanizmi	*Objašnjenje*
Programiranje u paru	U **XP** Programeri rade u paru. Dva programera sjede za jednim računarom radeći na istom zadatku.
Planiranje igre	Planiranje unapred je brzo i grubo. Ne treba se predugo zaustavljati oko plana koji treba dati osnovne smjernice.
Razvoj vođen testovima	Ne treba pisati ni jednu liniju koda pre nego što se napiše test koji taj kod treba da zadovolji. Cilj kodiranja se svodi samo na zadovoljavanje testova. Klijent takođe testira rješenja i pomoću rezultata testa upravlja razvojem.
Cjelokupnost tima	U timu bi trebao da se nađe i predstavnik klijenata koji bi u timu igrao ulogu stručnog savjetodavca. On piše priče korisnika, vrši funkcionalno testiranje i razjašnjava nejasnoće u hodu.
Stalna integracija	Kako se komponenta završi, sistem se integriše. Mora se imati odgovarajući način za praćenje promjena u kodu koji dozvoljava poništavanje izmjena, ako je potrebno.
Poboljšanje dizajna (refaktorisanje)	Svaki put kada se unapređuje neka komponenta, to se vrši pomoću refaktorisanja. Povremeno pregledati cio kod i refaktorisati ga kako bi se kod održao čitljivim.
Male verzije	Razvoj se vrši u malim iteracijama. To smanjuje cijenu promjena i odbacivanja loših rješenja. Svaka nova verzija se daje klijentu na testiranje i korišćenje što ga uključuje u proces razvoja.
Standardi kodiranja	Da bi tim funkcionisao bez zastoja zbog prilagođavanja na tuđi stil programiranja uspostavljaju se praktični standardi kojih se pridržavaju svi članovi tima.
Kolektivno vlasništvo koda	Svi su vlasnici svega, pošto svi rade sve. Tako se izbegava posvećenost pogrešnim idejama i rješenjima, otpor promjenama i olakšava homogenizacija znanja i vještina u timu.

Scrum

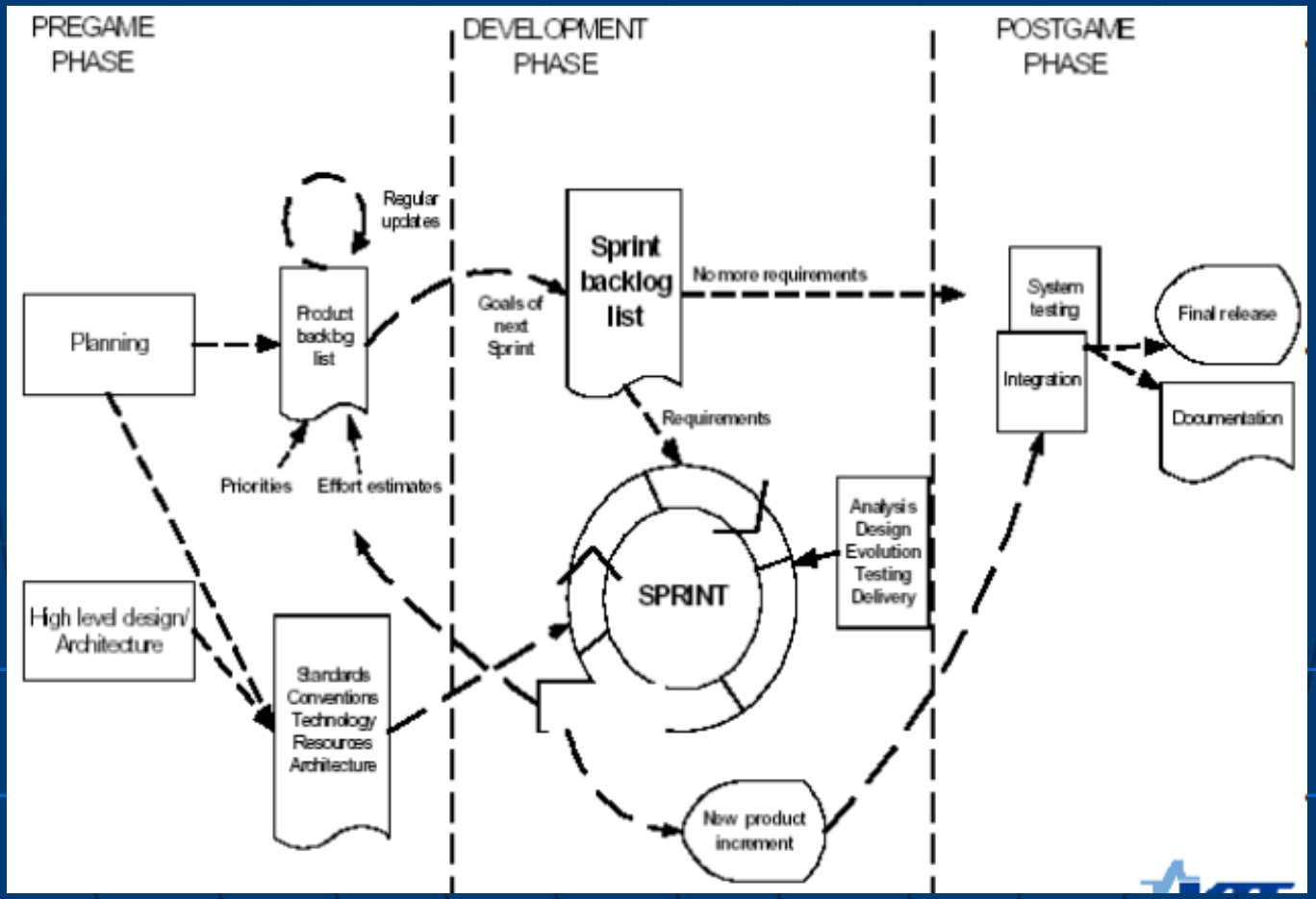
- Termin *Scrum* (*Schwaber* 1995. godine, *Schwaber* i *Beedle* 2002. godine) se u kontekstu novih principa i teorije programiranja prvi put opisuje, (1986. Hirotaka Takeuchi i Ikujiro Nonaka), kao pristup koji povećava adaptivnost, fleksibilnost, brzinu i samoorganizaciju procesa razvoja softvera.

- Odgovornosti i uloge u procesu razvoja softvera mogu se definisati, slika:

Scrum

- Može se reći da se Scrum sastoji od:
  - tehnika i alata u cilju izbegavanja haosa koji nastaje zbog kompleksnosti i neizvesnosti u dosadašnjim metodama razvoja softvera,
  - uloga i procedura koje u tri faze:
    - ✓ Pre-game - Pre igre *(planiranje, dizajn / arhitektura visok nivo apstrakcije)*
    - ✓ Development – Razvojna *igra (razvoj, sprintovi – iterativni ciklusi, poboljšanja, nove verzije)*
    - ✓ Post-game - *Posle igre (nema novih zaheva, sistem spreman za produkciju).*

Scrum


PREGAME PHASE
DEVELOPMENT PHASE
POSTGAME PHASE
Regular updates
Planning
Product backlog list
Goals of next Sprint
Sprint backlog list
No more requirements
System testing
Integration
Final release
Documentation
Requirements
Priorities
Effort estimates
Analysis
Design
Evolution
Testing
Delivery
SPRINT
High level design/ Architecture
Standards
Conventions
Technology
Resources
Architecture
New product increment

Scrum

Ova metoda je više vezana za agilno upravljanje softverskim projektom, nego za agilno projektovanje softvera. Ona propisuje načine upravljanja zahtevima, formiranja iteracija (planiranje sprinta), kontrole implementacije i isporuke klijentu.

Često se upotrebljava kao način vođenja **XP**, ili drugih projekata koji ne moraju obavezno da se projektuju nekom agilnom metodom.

Osnovu predstavljaju tri ključna pitanja koja se postavljaju na svakodnevnim, jutarnjim "stojećim" petnaestominutnim sastancima, a to su:

1. Šta je urađeno juče?
2. Šta će se raditi danas?
3. Kakve nas danas prepreke očekuju?

Ova pitanja se odnose na:

1. Kontrolu izvršenog
2. Planiranje budućeg dizajna
3. Identifikaciju rizika i nalaženje rešenja

Scrum

Jezgro scrum metodologije čine određeni elementi i prakse, a to su:

- **Sagledavanje delova proizvoda**
- **Uloge i odgovornosti**
- **Zalihe proizvoda i planiranje isporuka**
- **Sprint zalihe i planiranje sprinta**
- **Sprint**
- **Dnevni stojeći sastanci**
- **Karte dogorjevanja (*burndown charts*) i izvještavanje o projektu**
- **Pregled sprinta i retrospektiva • 59-minutni scrum**

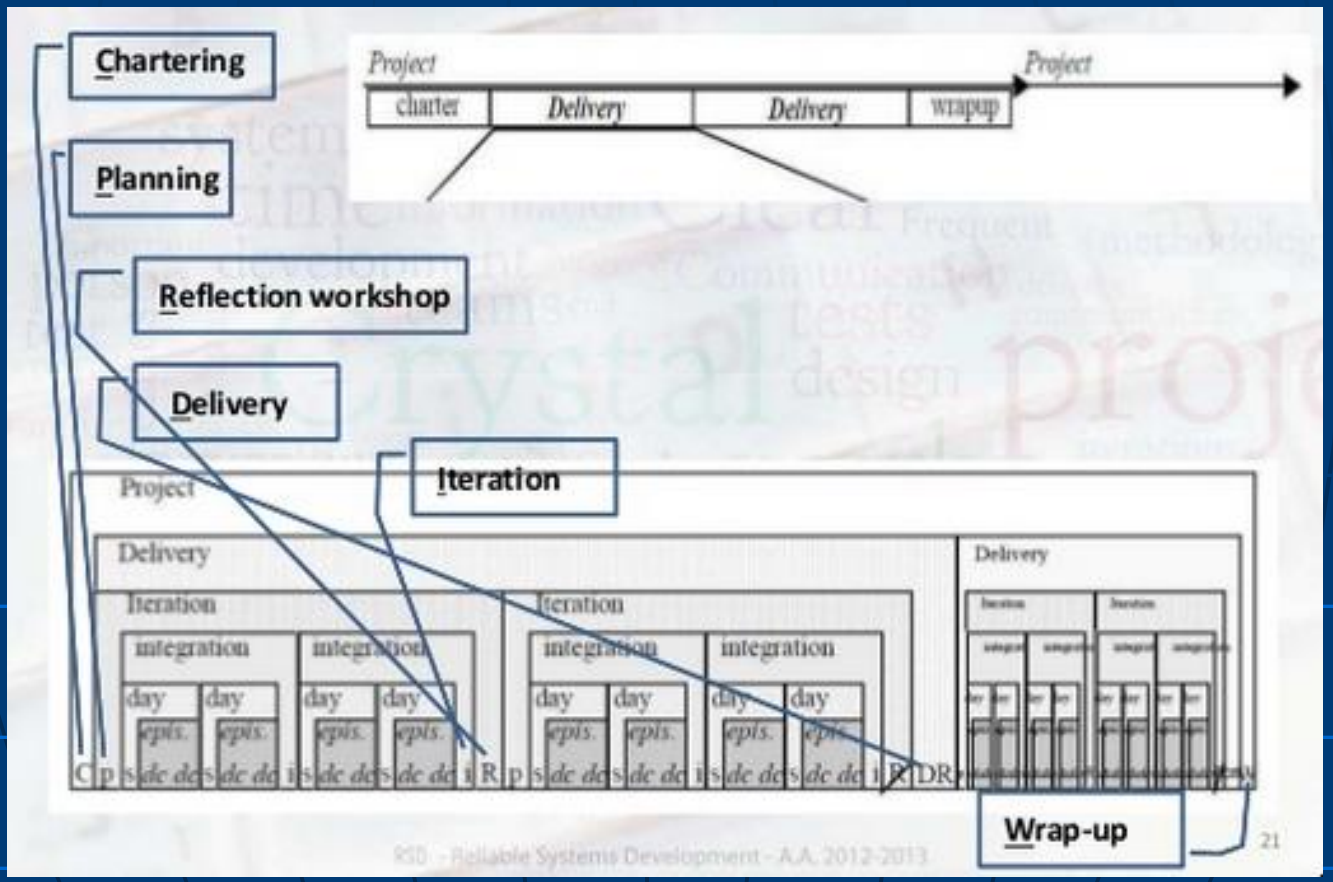
Scrum tim broji 5-10 članova od kojih su obaavezni **jedan vlasnik** proizvoda (product owner) koji je predstavnik klijenta, **scrum master** koji je vođa tima i ostali članovi tima koji mogu biti **specijalisti** za pojedine oblasti razvoja

Scrum metodologija

- Omogućava bolje uočavanje i menadžersku kontrolu svih potencijalnih defekata ili poteškoca u razvojnom procesu.

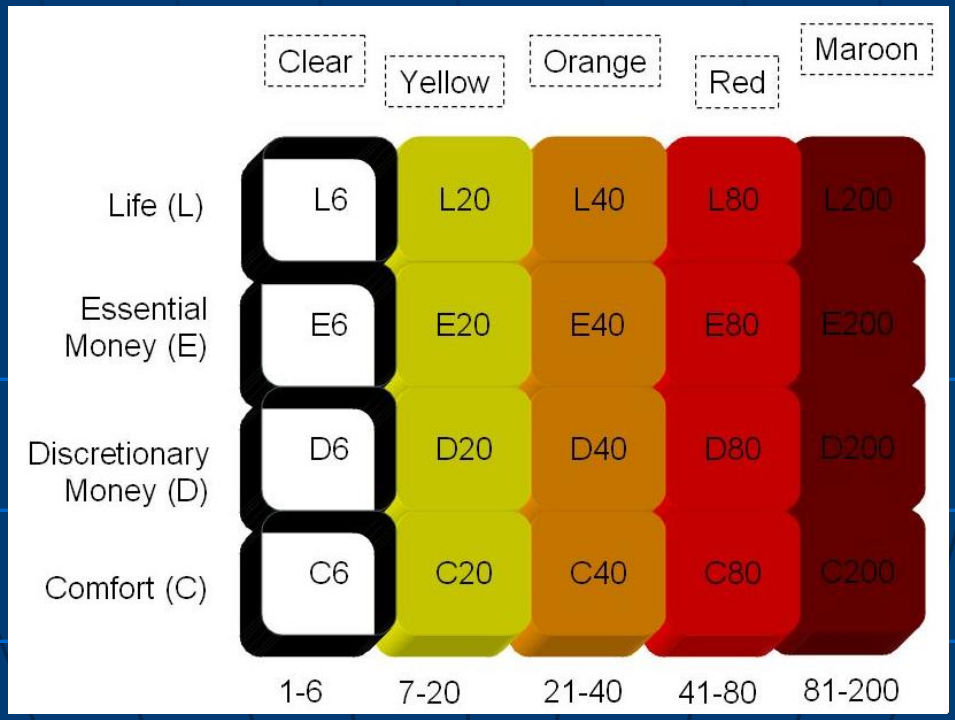
Crystal metodologija

- Crystal familija za razvoj softvera se uvodi kao skup metoda koje su više ili manje primjenjive na neke konkretne razvojne situacije i koja na osnovu povratnih veza najviše odgovara, a onda se i dodatno adaptira da odgovori izazovima konkretne situacije.



Crystal metodologija

- Metodologije u kojima centralno mesto predstavljaju ljudi - *"people-centric methodologies", su* bolje od *"process-centric methodologies",* u kojima centralno mesto zauzimaju procesi.
- One, u okviru Cockburn koncepta, pridružuju različite boje pojedinim metodama, da bi naglasile njihovu adekvatnost pojedinim tipičnim razvojnim situacijama, po principu: što tamnija boja, to "teža" metoda.
- Metodologije imaju svoja specifična imena, nazvana prema geološkim kristalima: čisti (clear), žuti, narandžasti, crveni, kestenjasti (maroon). Najpoznatije metodologije su cystal clear, crystal orange i crystal orange web.



Crystal metodologija

- **Učestala isporuka** – svakih nekoliko meseci, korisnici upoznati sa međuverzijama, daju povratne informacije.
- **Kontinualne povratne informacije** – projektni tim raspravlja o projektnim aktivnostima, vrši se validacija projekata sa korisnicima, rasprava o mogućim problemima.
- **Stalna komunikacija** – mali tim (svi u jednoj sobi), veći timovi lokacijski povezani
- **Sigurnost** – predstavljena na dva načina
  - sigurna zona - članovi tima komuniciraju i rade bez represije;
  - svi projekti nisu kritično isti
- **Fokus** – članove tima upoznati sa prioritetnim ciljevima, dati im mogućnost da ih ostvare
- **Raspoloživost korisnika** – članovima tima omogućiti saradnju sa korisnicima tokom celog projekta
- **Automatski testovi i integracija** – različiti oblici verifikacije funkcionalnosti projekata.

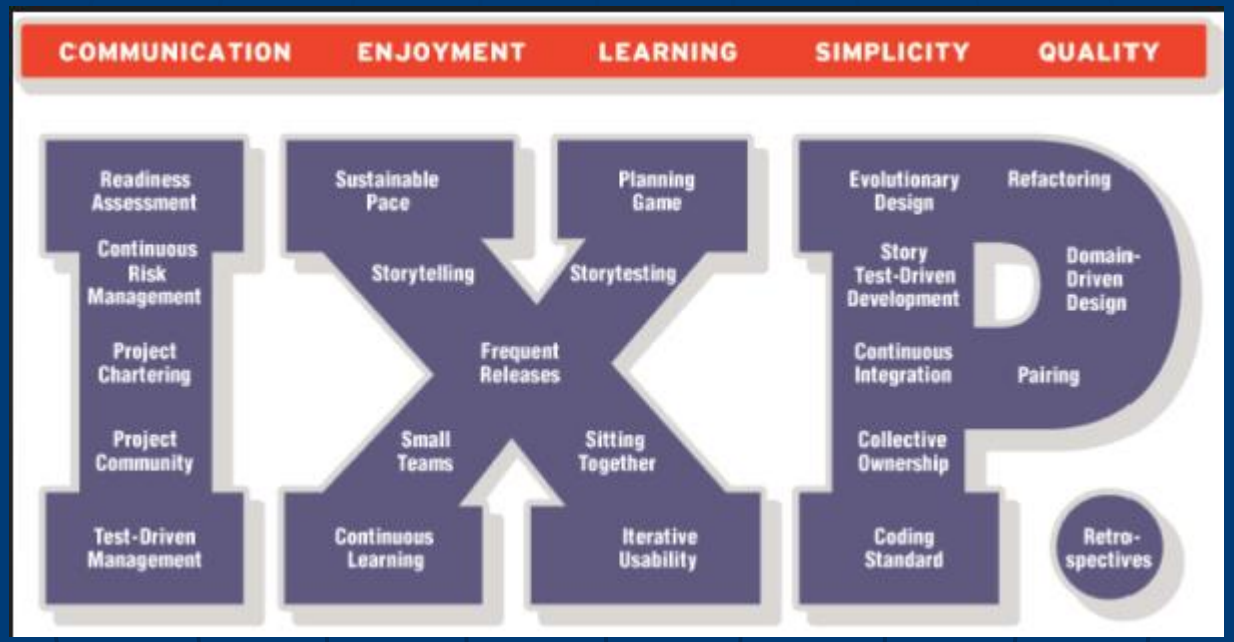
Industrijsko Ekstremno Programmiranje (IXP)

Nastaje 2004. u konsultantskoj firmi Cutter, gde se kao autor navodi Joshua Kerievsky, u vreme izdavanja drugog izdanja knjige Kenta Becka u kojoj je izvršena revizija i retrospektiva u odnosu na prvu verziju **XP**.

Ime dobija po tome što se koristi u velikim i distribuiranim timovima, te tako zadovoljava industrijske standarde velikih firmi.

IXP ide još i korak dalje sa tvrdnjom da osnovne vrednosti ekstremnog programiranja mogu biti fleksibilne, te variraju od tima do tima. Trebalo bi pri organizovanju **IXP,** ili reorganizovanju ka **IXP** upitati zaposlene o osnovnim vrednostima koje pokreću tim. Ovo se zasniva na tvrdnji Kenta Becka u drugom izdanju knjige “Extreme Programming - Explained” da obavezne prakse ne vrede puno, ako nisu motivisane osnovnim vrednostima. Na slici se može videti pet vrednosti koje koristi firma Cutter (komunikacija, zadovoljstvo, učenje, jednostavnost i kvalitet), ali one mogu varirati.

Industrijsko Ekstremno Programmiranje (IXP)



Ovdje se umesto dvanaest obaveznih praksi iz **XP,** pojavljuje čak dvadeset tri. Neke su revidirane, sa promenjenim nazivom, neke su dodate, a neke su ostale iste kao i u ranijoj verziji **XP.** Te prakse su sledeće:

Industrijsko Ekstremno Programmiranje (IXP)

Nove:

1. Pripremljenost za pristup
2. Projektna zajednica
3. Ugovaranje projekta
4. Menadžment vođen testovima (SMART princip)
5. Retrospektiva
6. Stalno učenje

Poboljšane postojeće prakse:

7. Razvoj vođen testovima priče (*storytest-driven development* – SDD) *unapređen* ***TDD***
8. Dizajn vođen domenom (*domain-driven design* – DDD) *unaprijeđenje* ***metafore u sistemu***
9. Uparivanje

unaprijeđeno ***programiranje u paru***

10. Iterativno testiranje korisnosti

unaprijeđeno ***prisustvo klijenta u timu***

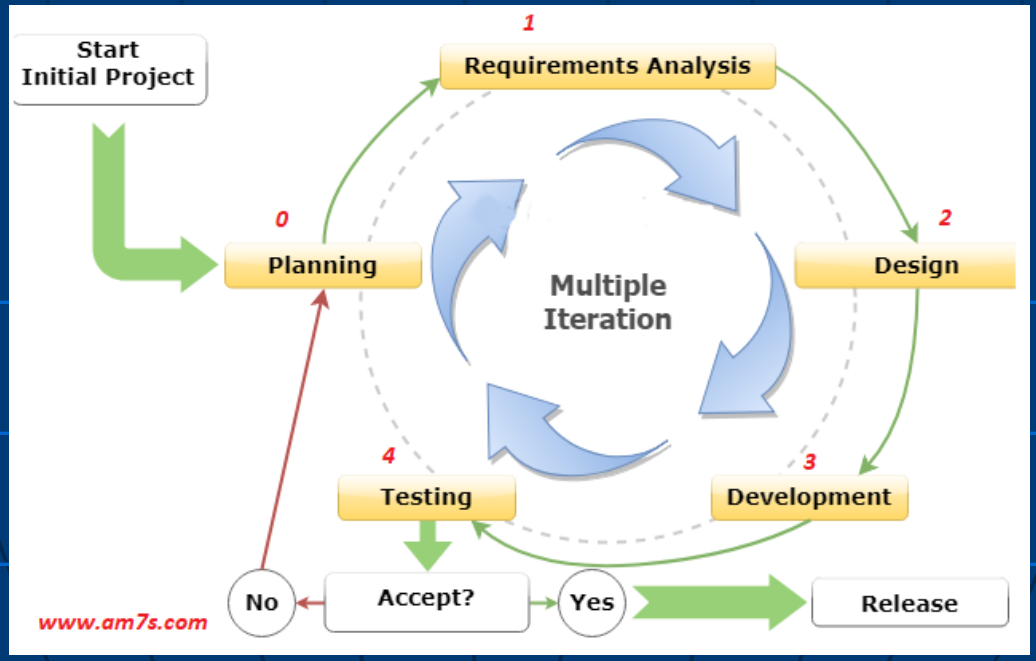
Nepromijenjene obavezne prakse:

11. Refaktorisanje
12. Igra planiranja
13. Stalna integracija
14. Zajedničko vlasništvo
15. Standardi u kôdu
16. Održivi korak
17. Česte isporuke

Agilno Modelovanje (AM)

Agilno modelovanje potiče od Scotta Amblera i njegove knjige "*Agile Modeling: Effective Practices for Extreme Programming and Unified Process*" iz 2002. godine. Ovde se modelovanje tretira kao važan način komuniciranja između članova tima u svim fazama razvoja softvera.

Članovi tima komuniciraju, između ostalog, i preko modela. Često se koristi UML, ali njegova formalna priroda ume da djeluje frustrirajuće na tim. AM je alternativa UML, ali ne isključuje njegovo korišćenje. To nije formalizacija posebnog stila grafičkog prikazivnja modela, nego okvir za korišćenje svih mogućih načina, zavisno od preferencija ljudi koji se modeliranjem i bave.



Agilno Modelovanje (AM)

Osnovne vrednosti AM: **hrabrost, komunikacija, povratna sprega, skromnost i jednostavnost.**

Osnovni principi AM: pretpostavljena jednostavnost, prihvatanje promena, sekundarni cilj je da omoguću dorade i dalji rad, postepene promene, maksimizira iskorišćenost investicija, modeliranje sa ciljem, višestruki modeli, kvalitetan rad, brza povratna sprega, primarni cilj je softver, putovanje sa malo prtljaga.

Sekundarni principi AM: sadržaj je važniji od načina prikazivanja, svako od svakoga može nešto da nauči, poznavanje svojih modela, poznavanje svojih alata, lokalno prilagođavanje, otvorena i poštena komunikacija, osjećaj za rad sa ljudima.

Osnovne obavezne prakse AM: jednostavno oslikavanje modela, javno prikazivanje modela, vrše se iteracije aktivno učešće svih zainteresovanih, primena pravih alata, razmotriti mogućnost testiranja, paralelno kreiranje nekoliko modela, kreiranje jednostavnog sadržaja, do druge vrednosti, modelira se u malim koracima, modelira se zajedno sa drugima, model se dokazuje pomoću koda, upotrebljava se najjednostavniji alat.

Agilno Modelovanje (AM)

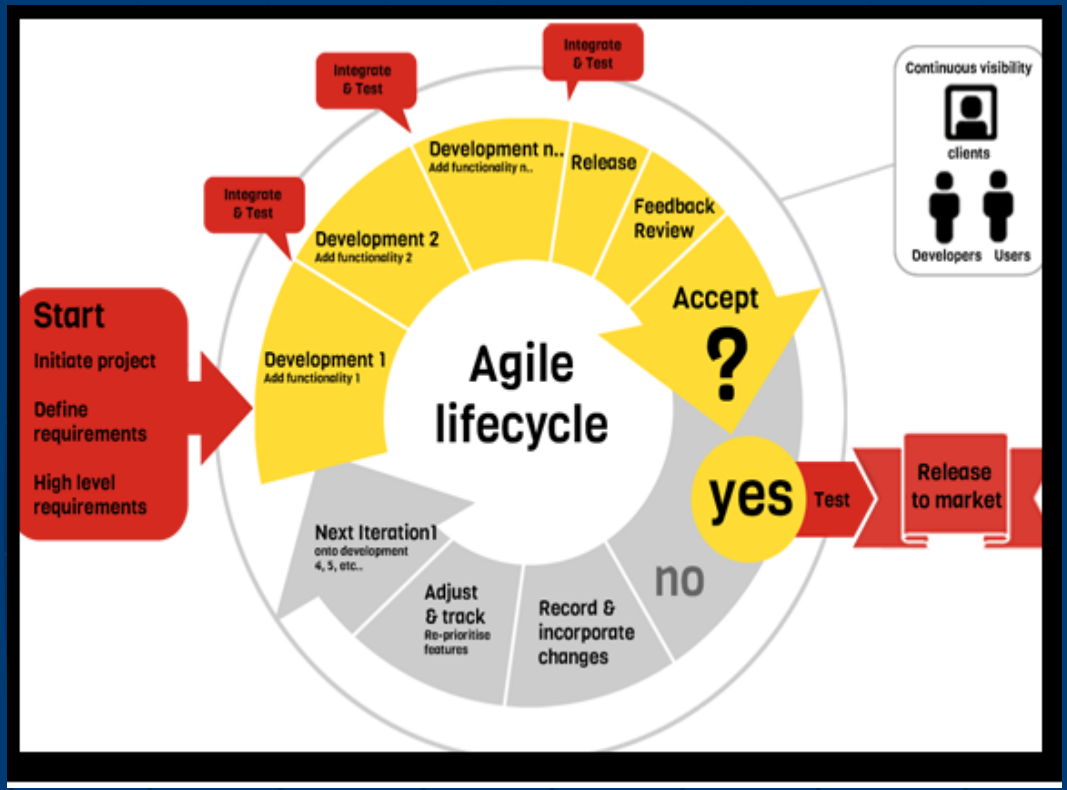
Kad se trebaju provesti nove promene.

Sloboda Agile promena je vrlo važna.

Nove promene mogu se sprovesti po vrlo maloj ceni zbog učestalosti novih inkremenata koji su produkovani.

Da bi implementirali novu pojavu, programeri moraju izgubiti samo nekoliko dana, pa čak i samo nekoliko sati, da bi se povukli i implementirali je u projekt.

Za razliku od modela vodopada u okretnom modelu potrebno je vrlo ograničeno planiranje kako bi se započelo sa projektom.



Agilna metodologija pretpostavlja da se potrebe krajnjih korisnika neprestano menjaju u dinamičnom poslovnom i IT svetu. O promenama se može razgovarati, a mogu biti novoprimećene ili korišćene iz baze sa povratnim informacijama. To kupcu omogučuje sistem dorade koji želi ili treba.
I programeri sistema, kao i korisnici, smatraju da takođe imaju više slobode vremena i mogućnosti.

Softversko inzenjerstvo

- Prvi uzrok softverske krize (ad hoc projektovanje) je najranije uočen i predstavljao je motiv za formiranje posebne discipline u raccunarstvu, nazvane ***softversko inženjerstvo***.

- Ovaj pojam se pojavljuje početkom sedamdesetih godina, pre pojma CASE proizvoda. Ideja softverskog inzenjerstva je bila uvodjenje metodološkog, inženjerskog pristupa pri razvoju programskih proizvoda, sa ciljem da se u zadatim vremenskim rokovima, preciznom primenom odgovarajucih tehnika, dodje kako do dovoljno kvalitetnog projekta programskog proizvoda, tako i do dovoljno kvalitetnog samog programskog proizvoda.

ŠTA SU CASE ALATI?

- **Computer-Aided Software Engineering (CASE)**

- **Sistemski inženjering pomoću računara (CASE)** je jedna aplikacija informacione tehnologije koja je okrenuta ka sistemskom razvoju aktivnosti, tehnika i metodologija.
- CASE alati su programi (softveri) koji automatizuju i podržavaju jednu ili više faza životnog ciklusa razvoja sistema.
- Namera ove tehnologije jeste da ubrza procese razvijanja sistema i poboljša njegov kvalitet.
- Neki ovu tehnologiju nazivaju kao softverski inženjering pomoću računara (*computer-aided software engineering*), međutim treba imati u vidu da je softver samo jedna komponenta informacionog sistema, pa se stoga ovde koristi širi pojam *sistem*.
- CASE nije metodologija niti bilo kakva njena alternativa.
- CASE je tehnologija koja podržava metodologije naročito strategije, tehnike i standarde.
- CASE tehnologija automatizuje celokupnu metodologiju razvoja sistema.

Definicija CASE alata

- CASE alati (Computer-Aided Systems Engineering) su automatizovani softverski alati, koji podržavaju izradu i analizu modela sistema i njegovih specifikacija i nude
  - manje troškove razvoja (brže, lakše održavanje i jednostavije modifikacije)
  - veći kvalitet (automatizacija dokumentovanja, provera konzistentnosti)

- Omogućavaju
  - Forward Engineering – mogućnost CASE alata da direktno generiše inicijalni deo softvera i/ili bazu podataka
  - Reverse Engineering – mogućnost CASE alata da generiše model sistema na osnovu postojećeg softvera i/ili baze podataka
  - Round-trip Engineering – sinhronizovane izmene koda i modela

- Neki CASE alati omogućavaju generisanje prototipa ili celokupnog koda

Vrste CASE alata

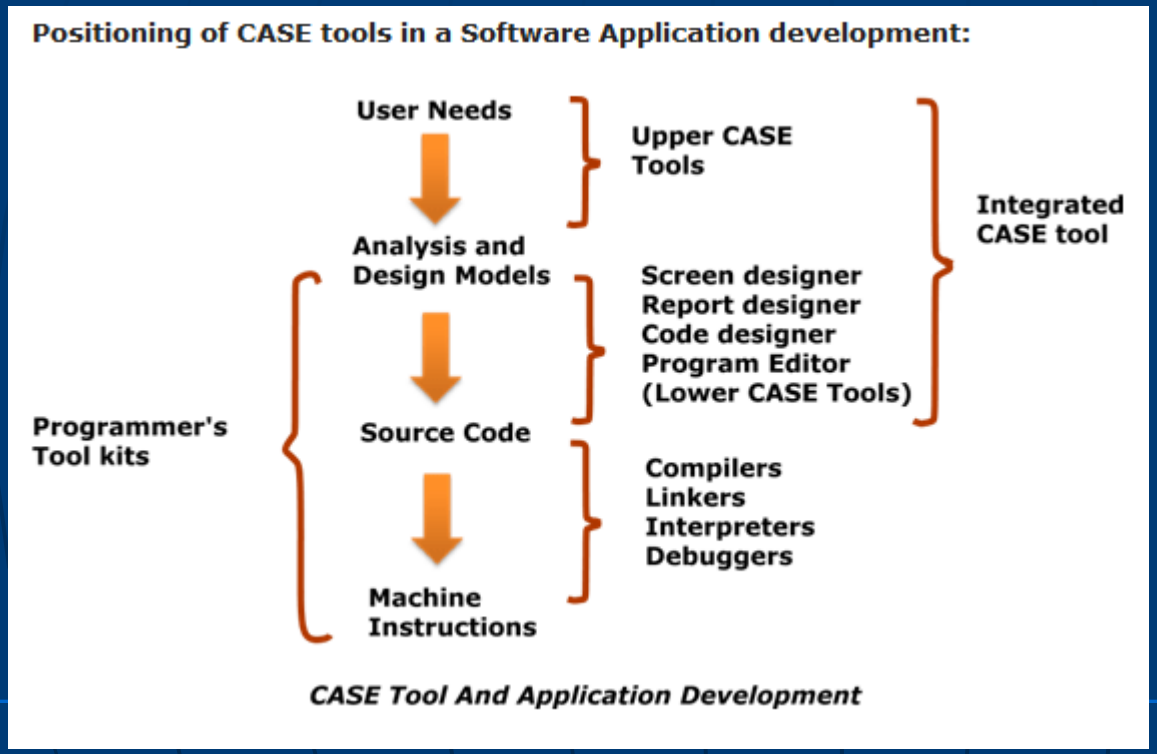
- Integrisani alati za podršku svih faza razvoja
- Alati za analizu i projektovanje (Upper CASE)
  - Alati za izradu dijagrama
  - Generatori formi i izveštaja
  - Alati za analizu
- Alati za implementaciju i održavanje (Lower CASE)
  - Generatori koda
- Alati za podršku (Cross Life-cycle CASE)
  - Alati za dokumentaciju i podršku upravljnja projektom

Kategorije CASE sistema

CASE Tools

- **CASE Tools Categories**
  - Major categories of CASE tools are:
    - Diagram Tools
    - Processing Model Tools
    - Project Management Tools
    - Documentation Tools
    - Analysis Tools
    - Design Tools
    - Configuration Management Tools
    - Change Control Tools
    - Programming Tools
    - Prototyping Tools
    - Web Development Tools
    - Quality Assurance Tools
    - Maintenance Tools

Alati CASE sistema



Kategorije CASE Tools:

CASE alati su klasifikovani u sledeće kategorije preko njihovih aktivnosti :

- ✓ UPPER CASE Tools
- ✓ LOWER CASE Tools
- ✓ INTEGRATED CASE Tools

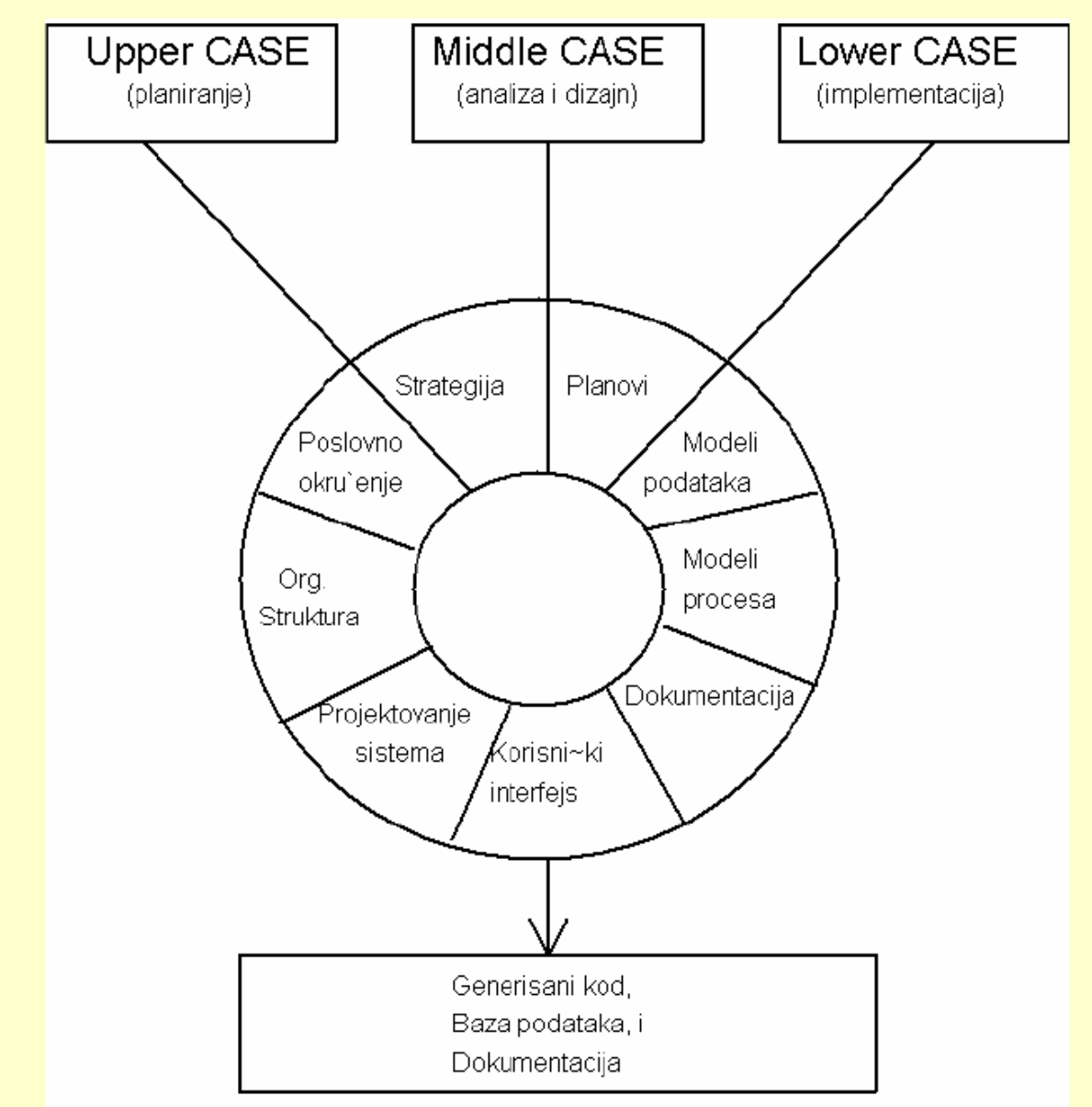
Klasifikacije CASE proizvoda

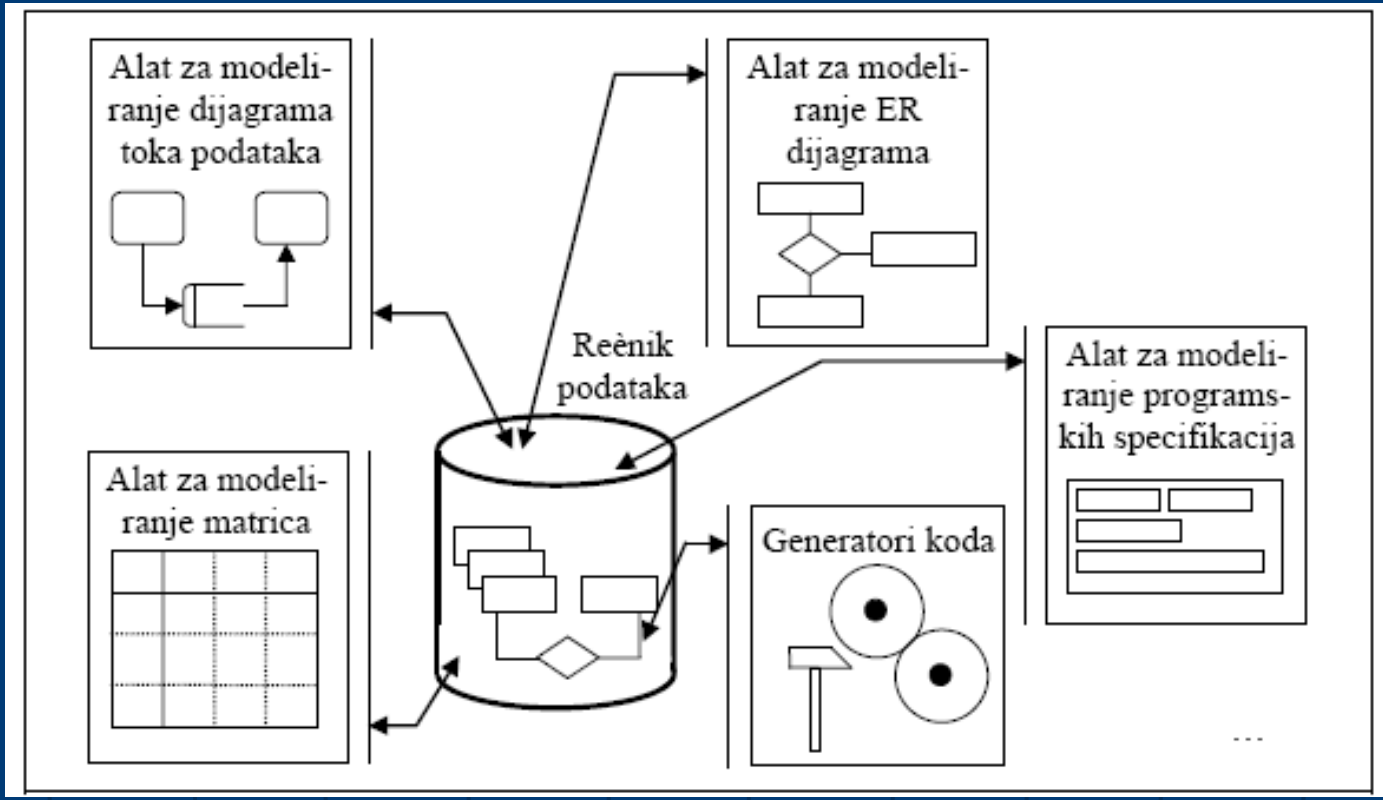
- ***Projektantski CASE (*** Upper CASE) proizvodi - namenjeni za podrsku prve ("gornje") tri faze zivotnog ciklusa (strategija, snimanje i analiza i projektovanje), odnosno za podrsku projektovanju programskog proizvoda,

- ***Programerski CASE (*** Lower CASE) proizvodi - namenjeni za podršku poslednje ("donje") tri faze životnog ciklusa (programiranje, uvodjenje u upotrebu, eksploatacija i održavanje), odnosno za podršku realizacije programskog proizvoda i

- ***Integrisani CASE* (Cross Life Cycle CASE)** proizvodi - integrisani projektantski i programerski CASE proizvodi, namenjeni da podrže svih šest faza životnog ciklusa, odnosno kompletan život programskog proizvoda.

Podela CASE alata po fazama životnog ciklusa





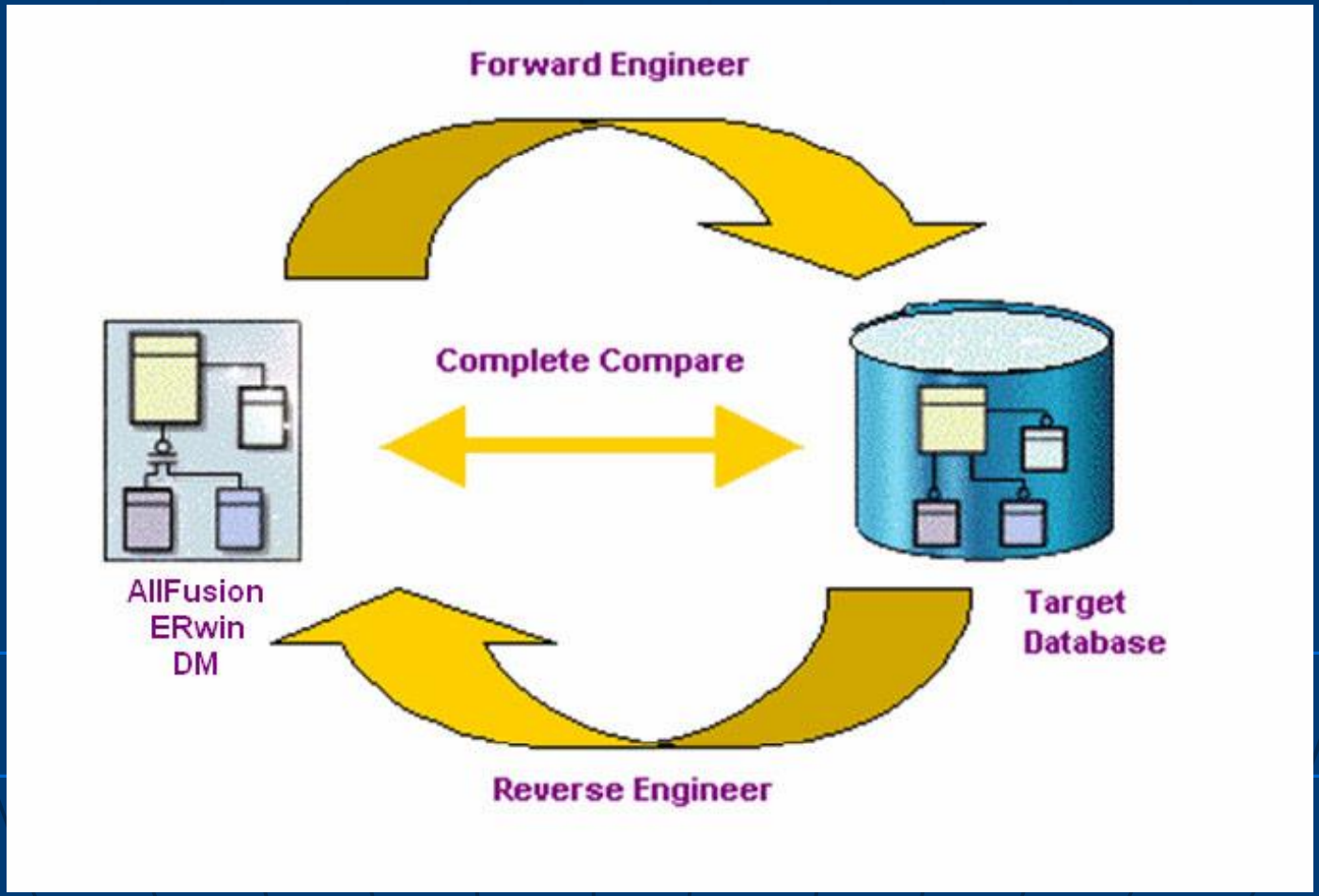
Omogućavaju:

– Forward Engineering: mogućnost CASE alata da direktno generiše inicijalni deo softvera i/ili bazu podataka

– Reverse Engineering: mogućnost CASE alata da generiše model sistema na osnovu postojećeg softvera i/ili baze podataka

– Round-trip Engineering: sinhronizovane izmene koda i modela

Forward i Reverse Engineering

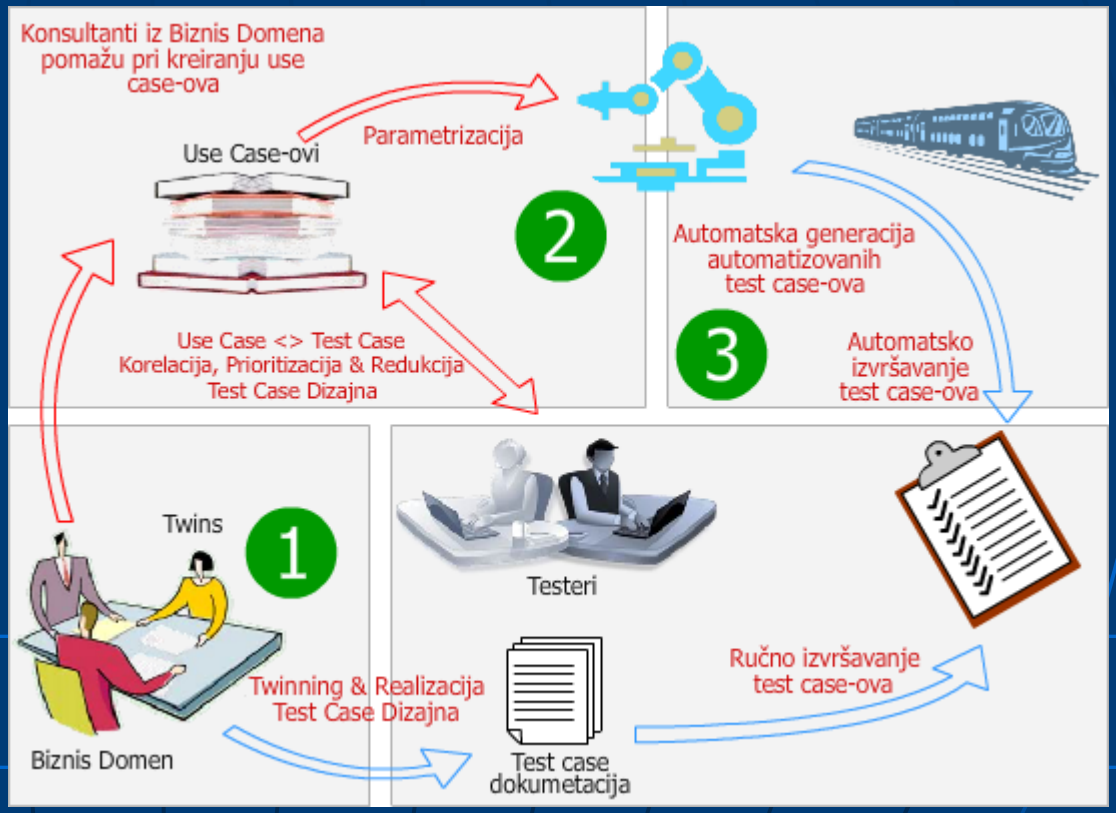


AUTOMATIZOVANI ALATI

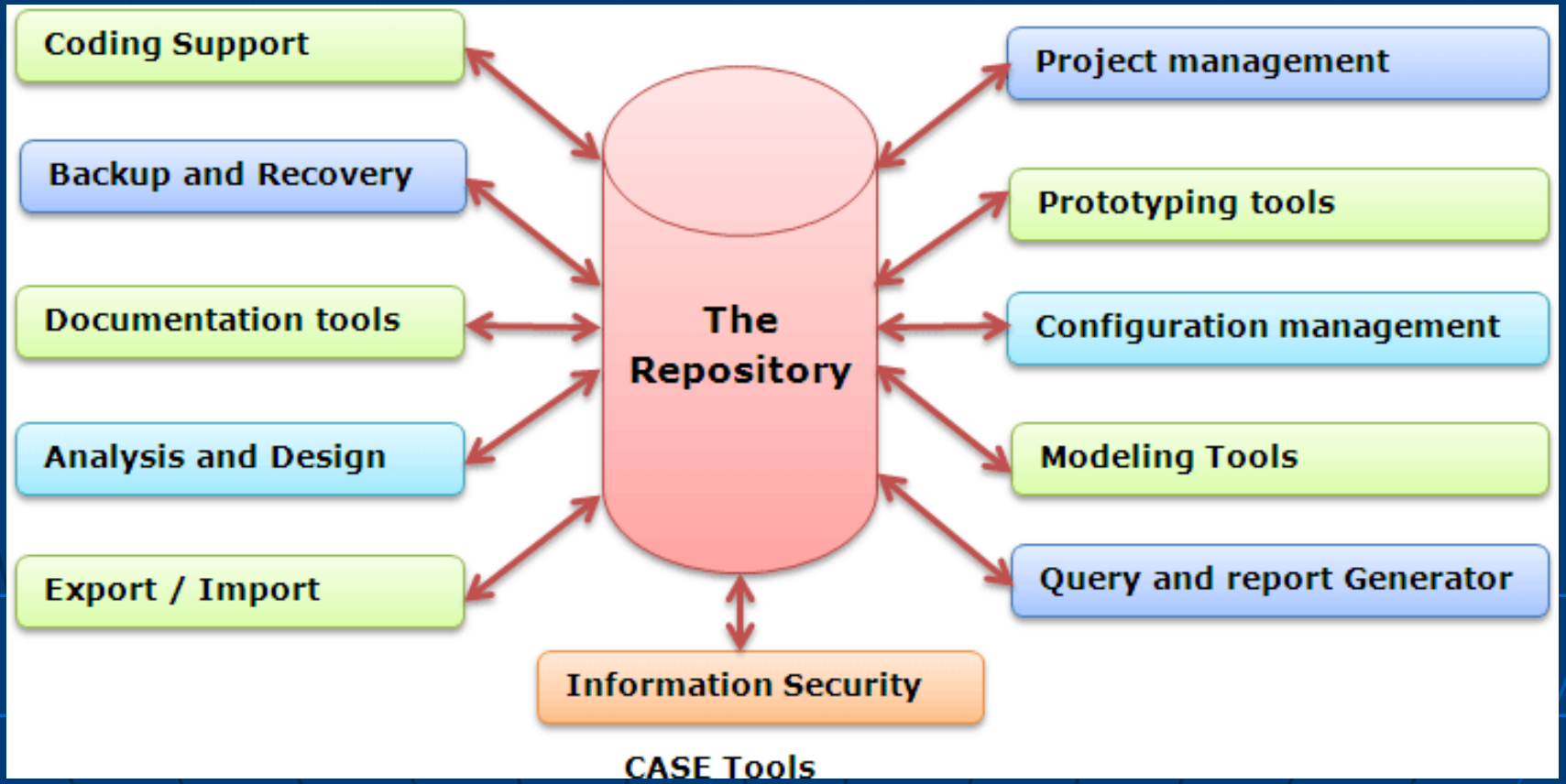
- Postoje 3 klase automatizovanih alata za developere:
  - modeliranje sistema pomoću računara (*computer-aided systems modeling*)
  - okruženje za razvoj aplikacija (*application development environments*)
  - upravljanje projektima i procesima (*project and process management*)

Primeri CASE alata:

BPWin, ERWin, System Architect, Rational Software Architect, DataArchitect, Oracle Designer, SmartDraw, Power Designer i dr.



AUTOMATIZOVANI ALATI



SOFTVERSKI ALATI KAO PODRŠKA MODELIRANJU

- Uspešnim korišćenjem pravilnog CASE alata može se postići sledeće:

  - minimizirati vreme razvoja softvera
  - višestruko povećati produktivnost u pisanju softvera
  - podići nivo kvaliteta softvera
  - povećati pouzdanost
  - standardizovati proizvedeni softver

Terminologija

• CASE – (Computer Aided Software/System Engineering) Računarom podržano programsko inženjerstvo je aplikacija automatizovane tehnologije za procedure softverskog inženjeringa.

• CASE tehnologija – Softverska tehnologija koja omogućuje automatizovanu inženjersku disciplinu za razvoj softvera, održavanje i rukovanje projektom, uključuje metodologije i automatizovane alate.

• CASE alati – Skup integrisanih CASE alata dizajniranih za zajednički rad koji automatizuju (barem delimično) celi životni ciklus softvera, uključujući analizu, projektiranje, generisanje koda i sl. uz podršku rukovanja konfiguracijama.
•Predstavlja nadogradnju jezika četvrte generacije

CASE STUDY

Razlozi za uvođenje naprednih alata

Softverska kriza

- **Niska produktivnost u razvoju informacionih sistema**
- **Visoki proizvodni troškovi**
- **Zaostajanje softvera u odnosu na hardver**

- **Rešenje softverske krize je u iskorišćenju osobina inženjera proverenih u praksi i to pre svega:**
  - **metodičnosti**
  - **operativne discipline**

CASE STUDY

Kao rezultat nastaje **softverski inženjering** koji u sebi sadrži sistematizovane i koordinirane aktivnosti potrebne pri:

- projektovanju
- implementaciji
- eksploataciji
- održavanju softverskih proizvoda

CASE STUDY

- Automatizacija softverskog inženjeringa se izvodi posebnim alatom, čiji je naziv **CASE** (**C**omputer **A**ided **S**oftware **E**ngineering)
- Osnovni cilj CASE alata:
- da razvijanje softvera postane više inženjerska delatnost, a manje individualna umetnost i umeće.

CASE STUDY

Očekivanja od CASE-a

- • Unaprjeđenje kvaliteta softvera
- • Povećanje produktivnosti programera
- • Unaprjeđenje kontrole razvojnog procesa
- • Niži troškovi razvoja
- • Niži troškovi održavanja
- • Smanjenje zaostalog razvoja
- • Povećanje zadovoljstva kupca

CASE STUDY

Uspešnim korišćenjem pravilno odabranog CASE

a**lata može se:**

- minimizirati vreme i trud (koštanje) razvoja softvera,
- višestruko povećati produktivnost u izradi softvera,
- podići nivo kvaliteta,
- povećati pouzdanost,
- standardizovati proizvedeni softver.

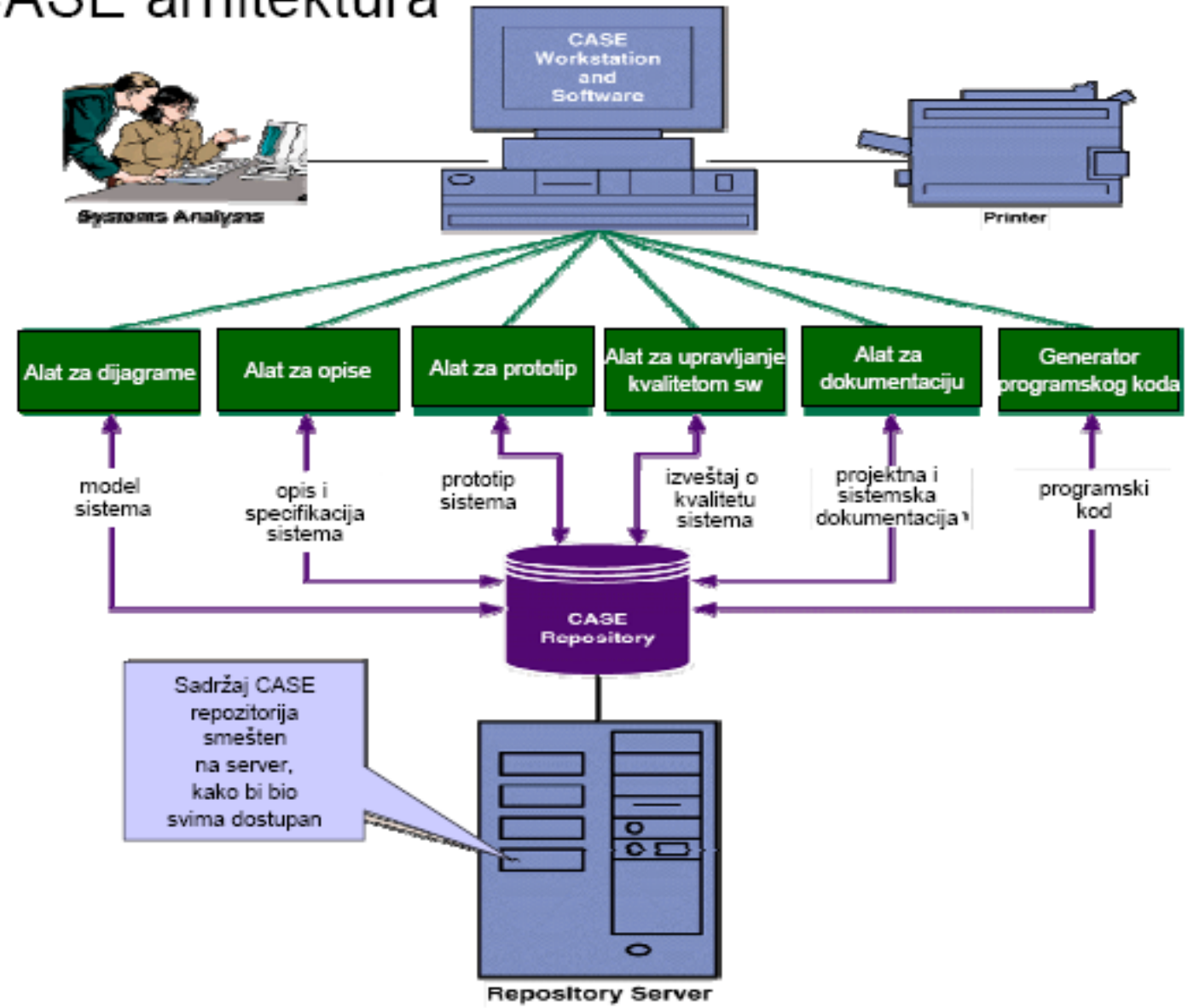
CASE STUDY

- Modeliranje poslovnih podataka i generisanje baze podataka nekog poslovnog sistema, koja će sadržati sve poslovne podatke važne za funkcioniranje poslovnog sistema mora biti integralan i celovit postupak.

- Na temelju podataka, donose se odluke u poslovanju i zbog toga je važno da svi potrebni podaci budu obuhvaćeni u bazi podataka, a odnosi koji postoje u realnom svetu moraju na isti način biti zabeleženi i u logici povezivanja njenih objekata.

- Izrada logičkog relacionog modela podataka i automatsko generisanje relacione baze podataka sprovodi se pomoću savremenih CASE alata. Ti alati omogućuju izdvajanje detaljnog logičkog relacionog modela podataka kao nezavisnog «proizvoda», što poslovnom sistemu otvara mogućnost kontrole razvoja i implementacije poslovnih aplikacija.

CASE arhitektura



CASE ALATI

- **Dijagramski alati (*Diagramming tools*)** se koriste za crtanje sistemskih modela.
- **Rečnik alati (*Dictionary tools*)** se koriste za snimanje, brisanje, izmenu i prikazivanje detaljne dokumentacije i specifi kacije. Opisi su obično pridruženi elementima sistemskih modela koji su prethodno iscrtani dijagramskim alatima.
- **Alati za projektovanje (*Design tools*)** se koriste za projektovanje komponenata sistema uključujući ulaze (*inputs*) i izlaze (*outputs*).

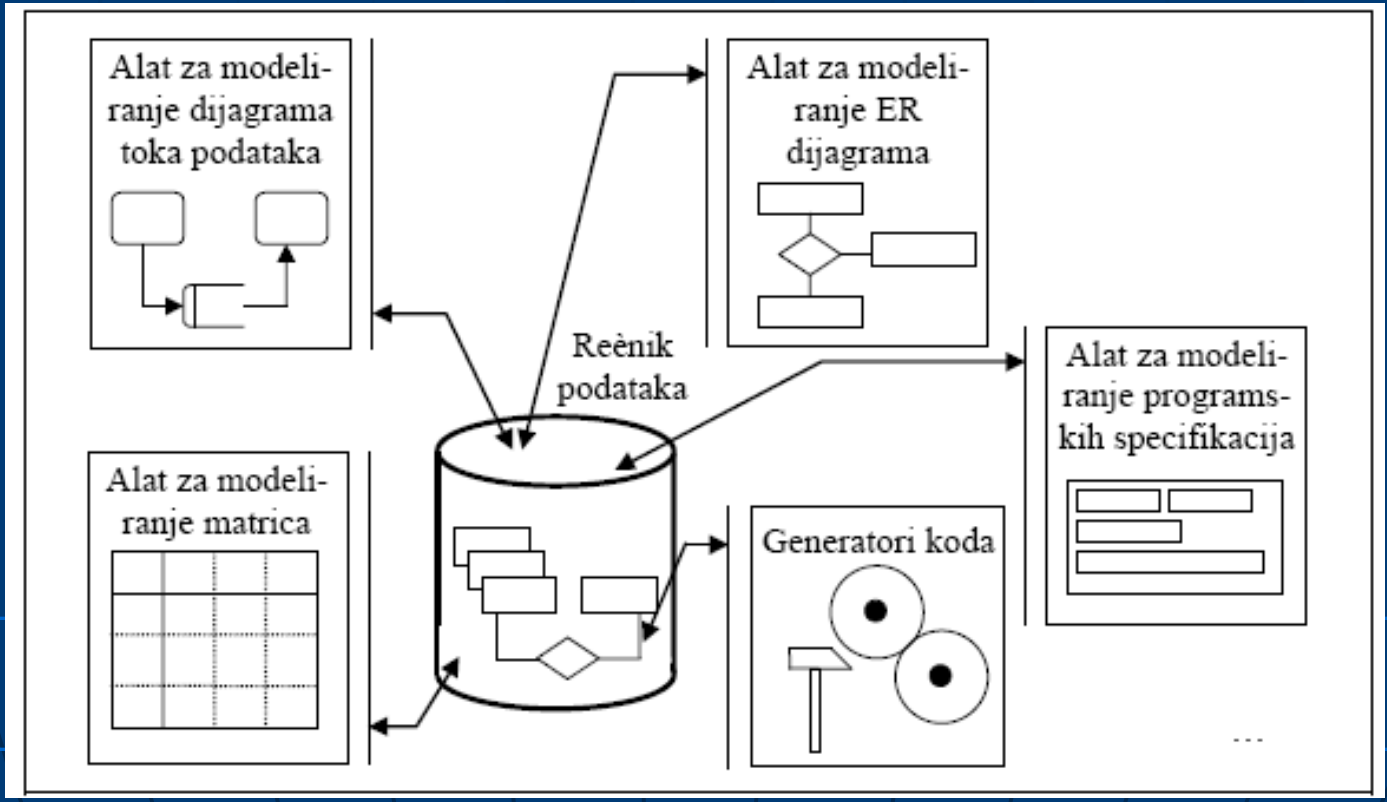
↗ **Alati za upravljanje kvalitetom (Quality management tools)** analiziraju i utvrđuju konzistentnost i kompletnost modela, opisa i dizajna. Ukoliko dođe do pojave greške, CASE alati ih identifi kuju i obaveštavaju korisnike.

↗ **Dokumentacioni alati (Documentation tools)** se koriste za sakupljanje, organizaciju i izveštavanje neophodne dokumentacije iz repozitorijuma.

↗ **Alati generatora koda i dizajna (Design and code generator tools)**

↗automatski generišu dizajn baze podataka iz modela podataka, aplikacione programe ili značajne delove ovih programa.

CASE ALATI


Alat za modeli-
ranje dijagrama
toka podataka
Alat za modeli-
ranje ER
dijagrama
Reènik
podataka
Alat za modeli-
ranje programs-
kih specifikacija
Alat za modeli-
ranje matrica
Generatori koda
...

CASE STUDY

Podela CASE alata

horizontalna

- za više faze životnog ciklusa (analiza, dizajn)
- za srednje faze životnog ciklusa (izrada aplikacija,
- Implementacija)
- za niže faze (podrška eksploataciji)

vertikalna

- upravljanje, planiranje, praćenje
- tehnički alati
- podrška projektu (rečnici, skladišta)

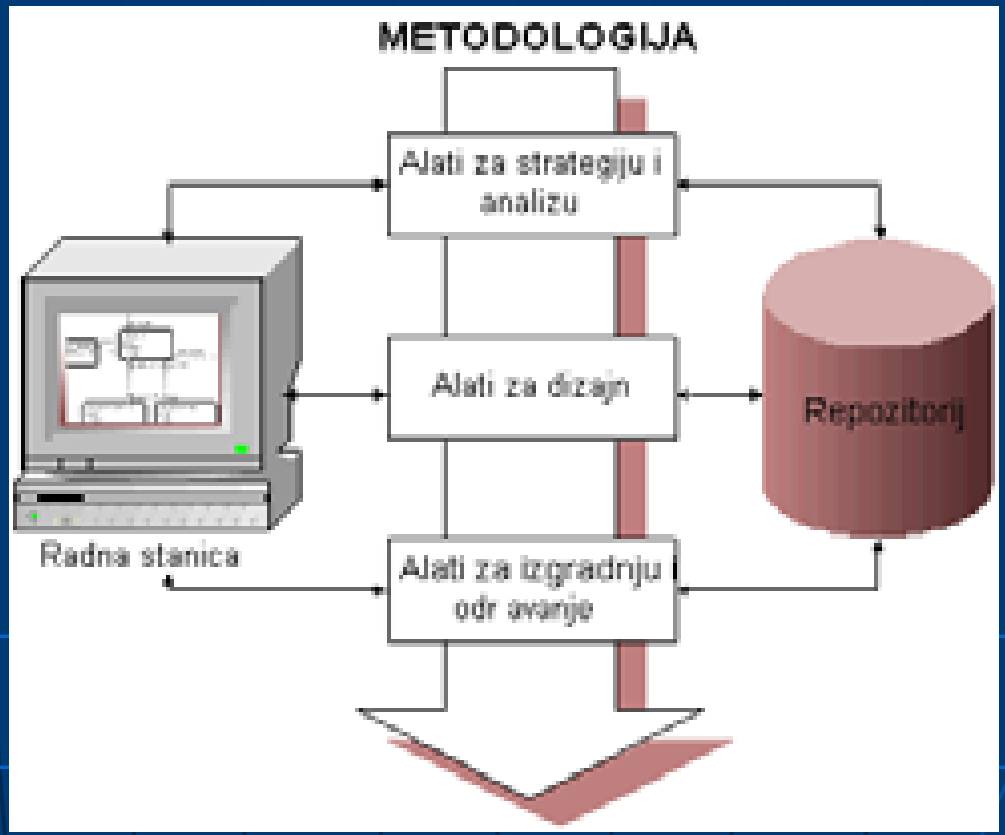
prema broju korisnika

- jednokorisnički
- višekorisnički (mrežni)

CASE STUDY

Arhitektura CASE alata

- Karakteristike:
- • Grafički sistem
- • Obezbeđuje minimalnu redundanciju
- • Ima mogućnost provere grešaka
- • Radi na generisanju koda



Projektantski CASE proizvodi

- ***Projektantski CASE*** proizvodi treba da podrže prve tri faze životnog ciklusa. U domenu projektovanja informacionih sistema, CASE proizvod koji podržava fazu strategije, moze da sadrzi alate za podrsku:

- planiranja projekta (izbora metodologije i tehnika razvoja informacionog sistema, načina i standarda za primenu izabrane metodologije i tehnika),
- upravljanja projektom (detaljnog planiranja i izdavanja zadataka i vremenskog terminiranja projekta),
- planiranja i upravljanja resursima (materijalnim, kadrovskim i finansijskim),
- praćenja realizacije projekta i
- sprovodjenja postupaka kontrole kvaliteta.

CASE faza snimanja i analize

- Kada je u pitanju podrška u fazi snimanja i analize, CASE proizvod može da sadrži alate za izradu:
- strukturnih modela sistema (model funkcionalne, organizacione i prostorne strukture),
- modela procesa koji se odvijaju u realnom sistemu,
- dijagrama toka podataka,
- konceptualne šeme baze podataka i matrica, kojima se iskazuju medjuzavisnosti izmedju elemenata konceptualne šeme baze podataka, kao i funkcionalne, organizacione, ili prostorne strukture sistema.

CASE faza projektovanja

- Za fazu projektovanja, CASE prozivod može sadržati alate za:
- prevodjenje konceptualne šeme baze podataka u implementacionu šemu,
- implementaciono projektovanje šeme baze podataka, koje se može sprovoditi direktno, bez prethodne izrade i prevodjenja konceptualne šeme, ili putem modifikacija prevedene konceptualne šeme,
- generisanje programskih specifikacija aplikacija (struktura menija, opisa ekranskih ili štampanih formi, podšema i standardnih procedura za upite i ažuriranje baze podataka) i
- implementaciono projektovanje programskih specifikacija aplikacija, koje se može sprovoditi direktno, bez prethodnog generisanja programskih specifikacija, ili putem modifikacija prethodno izgenerisanih programskih specifikacija.

CASE STUDIJA

CASE alati na tržištu:

- PVCS
- *Maestro II*
- **Oracle Designer**
- VisualAge Pacbase
- Paradigm Plus
- *ERvin*
- *BPwin*

Stp/UML
System Architect
DOORS
MetaEdit+ Rational Rose
WithClass 2000
Data Integrator Designer
MS Visio

CASE STUDIJA

CASE obuhvata:

- – Generator koda (Code genaration tools)
- – Alatke za modeliranje podataka (Data modeling tools)
- – UML (Unified Modeling Language)
- – Refactoring tools
- – QVT (Queries/Views/Transformations)
- – Alatke za upravljanje konfiguracijom, sa kontrolom revizije

CASE STUDIJA

Generator koda (Code generator)

- • Programski kod se može unositi u neki od programskih editora, a može se i automatski generisati

- • Kreiranje štampe izveštaja iz baze podataka može se pisati "peške", a može se koristiti i npr. program Access, sa mnogo komotnijem (user frendly) okruženju i tada Access sam kreira kod

- • Kada se kod automatski generiše izbegavaju se greške (ljudski faktor)

CASE STUDIJA

Osnovni zadaci generatora koda su:

- – Selekcija naredbi (Instruction selection) – koje

naredbe treba koristiti

- – Redosled naredbi (Instruction scheduling) – kojim

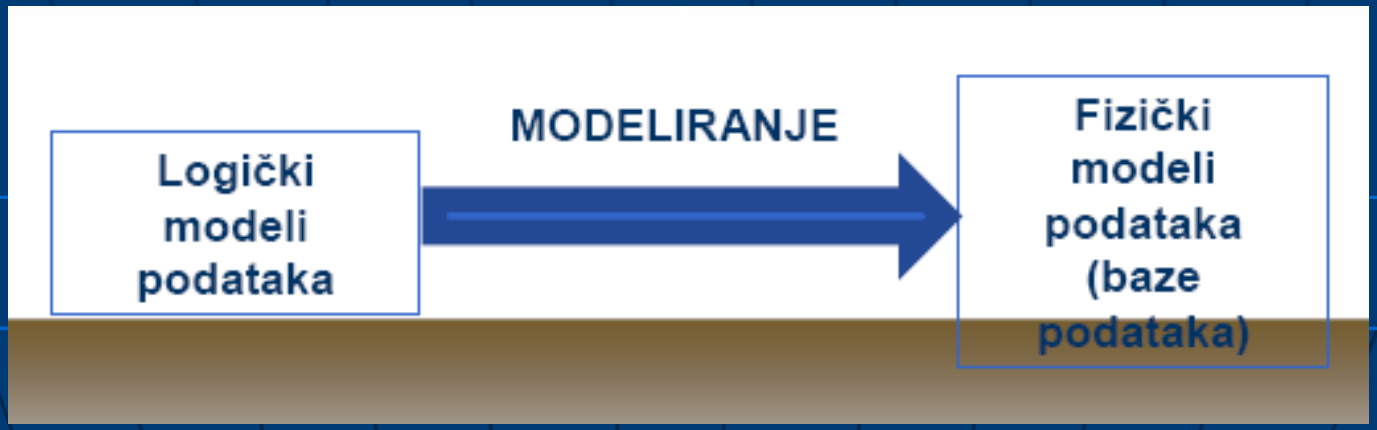
redosledom poređati naredbe, a optimalan redosled ubrzava izvršavanje programa

- – Alokacija registra (Register allocation) – alokacija varijabli u odnosu na procesorske registre

CASE STUDIJA

Alatke za modeliranje podataka

- • Modeliranje podataka je proces kreiranja modela podataka, upotrebom teorije modeliranja



CASE STUDIJA

Kada se vrši modeliranje, praktično se strukturiraju i organizuju podaci

- • Strukturirani podaci se zatim, implementiraju u sistem upravljanja bazama podataka (Database Management System)

- • Upravljanje velikom količinom strukturiranih i nestrukturiranih podataka osnovna je funkcija

informacionih sistema

CASE STUDIJA

UML (Unified Modeling Language)

- • U oblasti softverskog inženjeringa UML je standardni jezik za objektno modeliranje
- • UML je jezik opšte namene koji obuhvata grafičke simbole koji se koriste za kreiranje apstraktnih modela sistema (tzv. UML modela)
- • Naziva se još i meta model

CASE STUDIJA

Refactoring tools (prečišćavanje koda)

- • Namenjeni su usavršavanju koda sa ciljem povećanja unutrašnje konzistentnosti (doslednosti) i jasnoće

- • Ove alatke ne pronalaze greške, niti dodaju novu funkcionalnost programima

- • Trivijalan primer: ako je u programu varijabla definisana kao ks, samo programer zna šta ona zaista predstavlja, ali ako je preimenujemo u kamatna_stopa, biće i ostalima jasno

CASE STUDY

QVT (Queries/Views/Transformations) je standard za transformaciju modela definisan od strane Object Management Group-e

- Alatke za upravljanje konfiguracijom, sa kontrolom revizije (Configuration management) pre svega kontrolišu promene koje nastaju na hardveru, softveru, dokumentaciji i sl. tokom životnog veka sistema

- http://www.unl.csi.cuny.edu/faqs/softwareengi nering/

tools.html – dopunske informacije

CASE za Web servise

SOFTVERSKI ALATI za Web Servise i WSDL	Vrlo nizak	Nizak	Normalan	Visko	Vrlo visok
	Kodianje, editovanje, ispravljanje greški	Jednostavan *front end*, *back end*, malo integracije	Osnove životnog cilkusa, srednji nivo integracije	Jako podržan životni ciklus i srednji nivo integracije	Jako podržan, proaktivni životni ciklus i visoka integracija
Q-WS	X	X-			
MapPoint Web Service SDK,	X	X	X	X	
IBM Web Services ToolKit	X	X	X	X	
XMLSpy® 2005	X	X	X-		
Stylus Studio 6	X	X			
Panacea Software	X	X-			
	Gde oznaka X ukazuje na postojanje a X- na delimičnо postojanje				

CASE alat za modeliranje podataka – ERWin data modelar

- Izgrađen na konceptima koji su postavljeni IDEF1X (**I**ntegration **Def**inition for information modeling) standardom
- Namenjen za modeliranje podatka **ER** (entity Relationship) metodom
- Mogu se ravnopravno koristiti **IDEF1X** i **IE** (**I**nformation **E**ngineering) notacije

CASE alat za modeliranje podataka – ERWin data modelar

Modeliranje podataka obuhvata dva aspekta

- Logičko definisanje modela
- Fizički dizajn baze

- Korišćenje ERwin-a podrazumeva istovremeni razvoj logičkog modela i fizičko definisanje baze

CASE alat za modeliranje podataka – ERWin data modelar

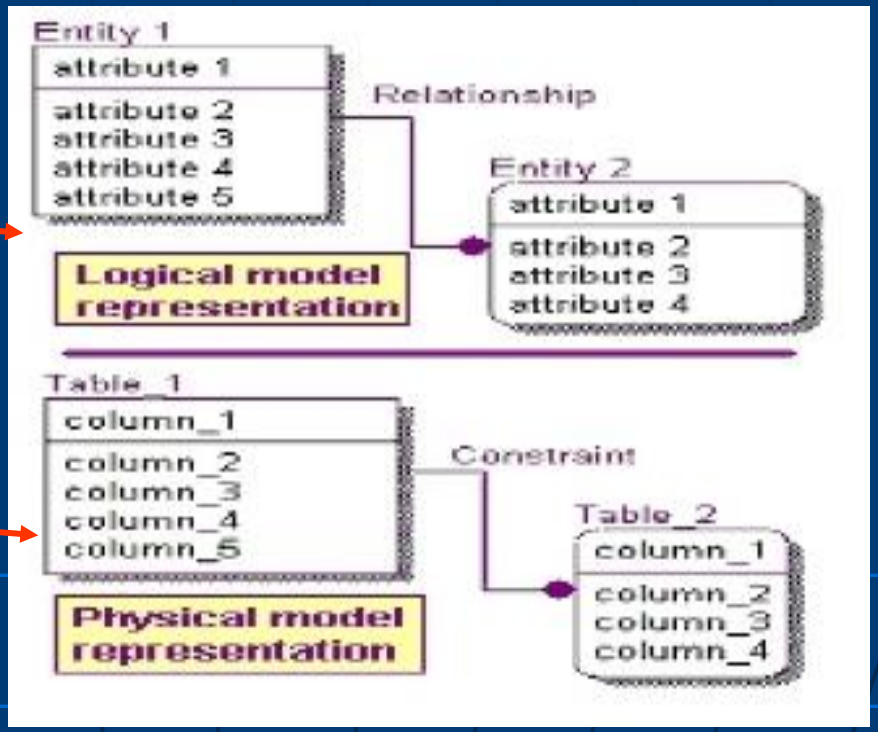
Osnovi objekti u modelima

Logički model

- Objekti
- Atributi
- Veze (relacije)

Fizički model

- Tabele
- Polja
- Ograničenja (constraints)
- Pogledi (views)



CASE alat za modeliranje podataka – ERWin data modelar

U bilo kom trenutku definisanja logičkog nivoa baze, potrebno je odabrati **ciljni server** (DBMS na kome će baza biti realizovana)

- Izbor ciljnog servera **nije definitivan**, što znači da se on može menjati
- ERwin insistira na **nezavisnosti modela od DBMS platforme**

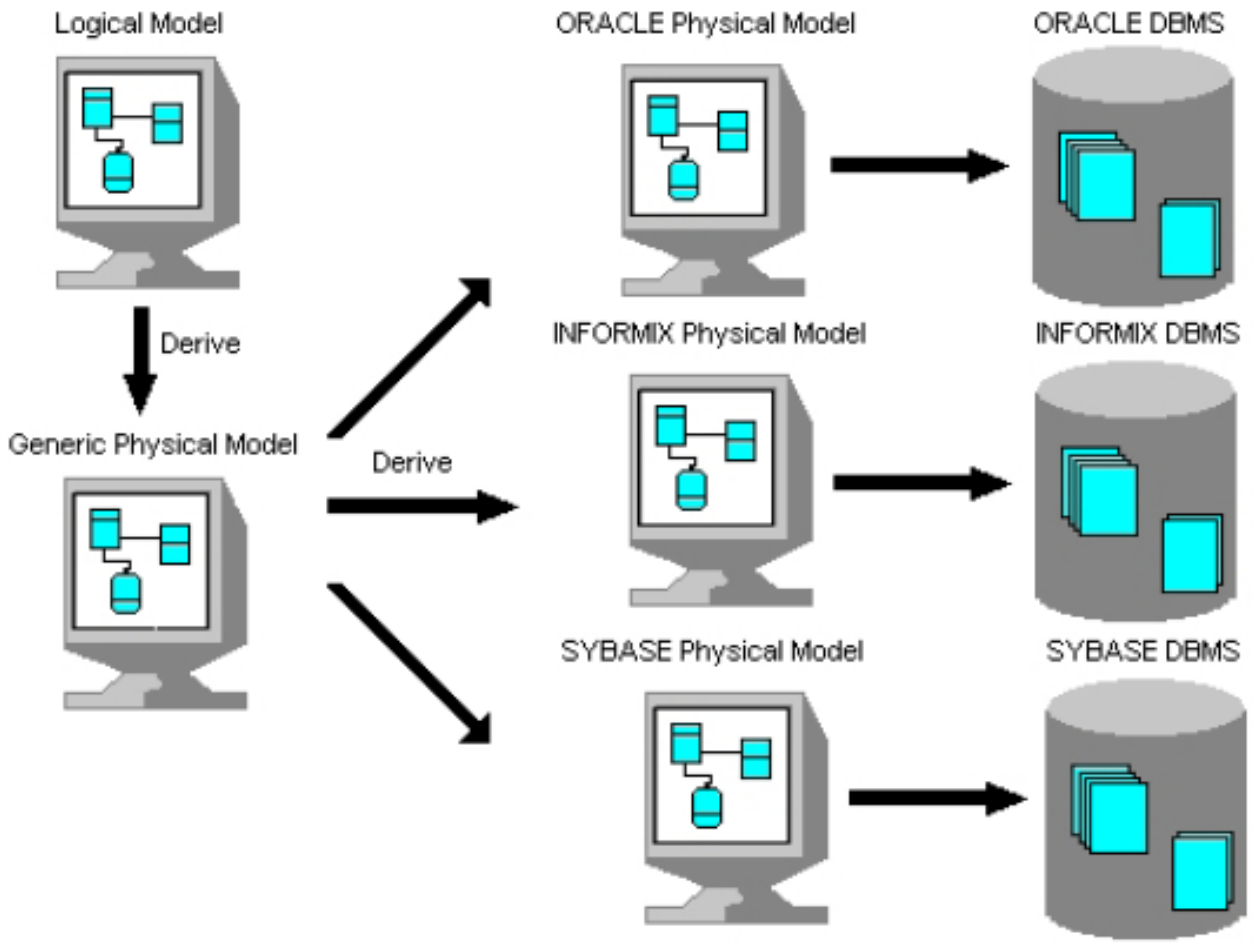
CASE alat za modeliranje podataka – ERWin data modelar

U Erwin-u postoje **logički** i **fizički tipovi podataka**

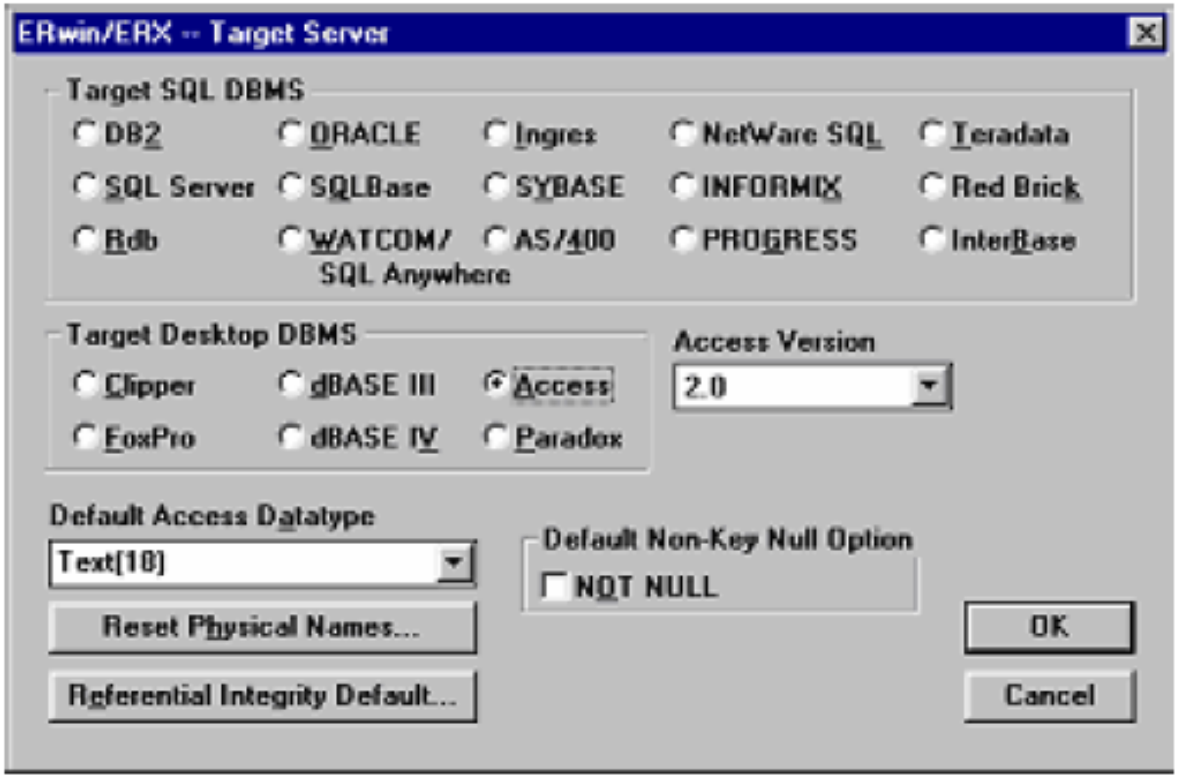
- **Logički tip podataka** (logical datatype) je predefinisani skup karakteristika atributa, koji određuje dužinu polja, pripadnost ERwin domenu i naziv.

- ERwin poseduje listu predefinisanih logičkih tipova koje možemo dodeljivati atributima

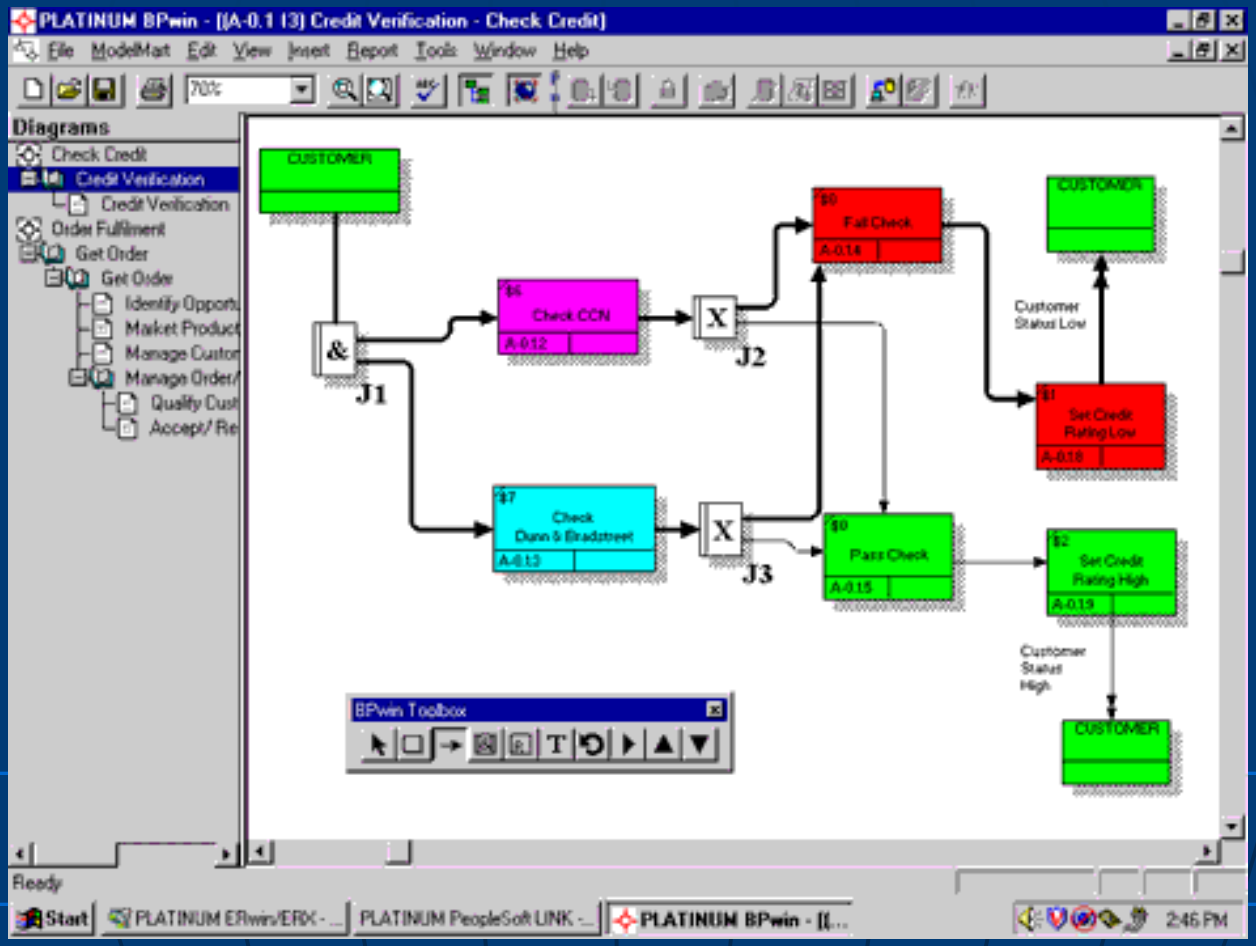
AllFusion™ ERwin® Data Modeler



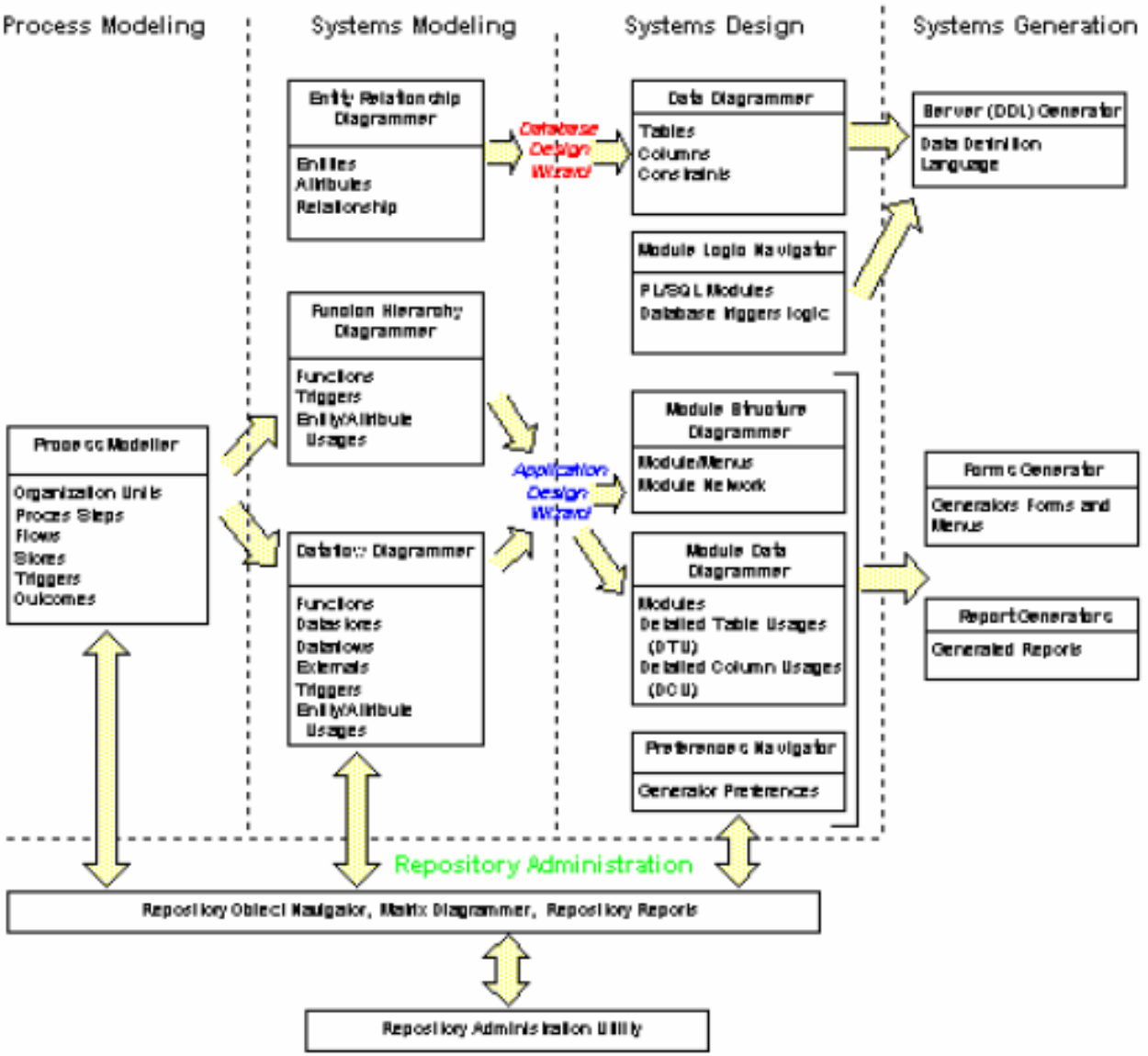
ERwin – primer izbora servera DBMS



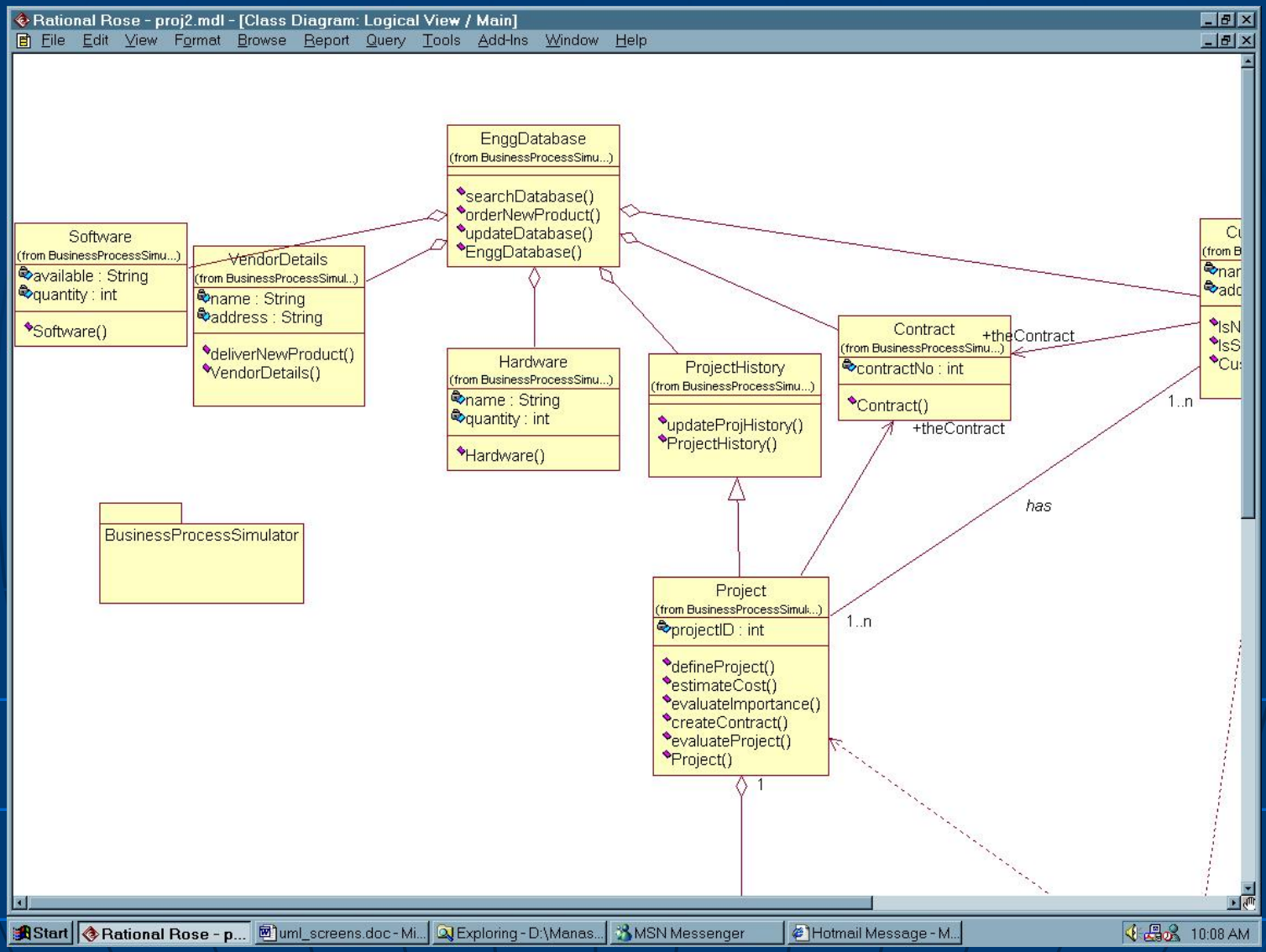
BPwin


PLATINUM BPwin - [(A-0.1 13) Credit Verification - Check Credit)
File ModelMart Edit View Insert Report Tools Window Help
Diagrams
Check Credit
Credit Verification
Credit Verification
Order Fulfillment
Get Order
Get Order
CUSTOMER
&
J1
Check CCN
A-0.12
X
J2
Fail Check
A-0.14
CUSTOMER
Customer Status Low
Set Credit Rating Low
A-0.18
Check Dunn & Bradstreet
A-0.13
X
J3
Pass Check
A-0.15
Set Credit Rating High
A-0.19
Customer Status High
CUSTOMER
BPwin Toolbox
Ready
Start
PLATINUM ERwin/ERX - ...
PLATINUM PeopleSoft LINK -...
PLATINUM BPwin - [(...
2:46 PM

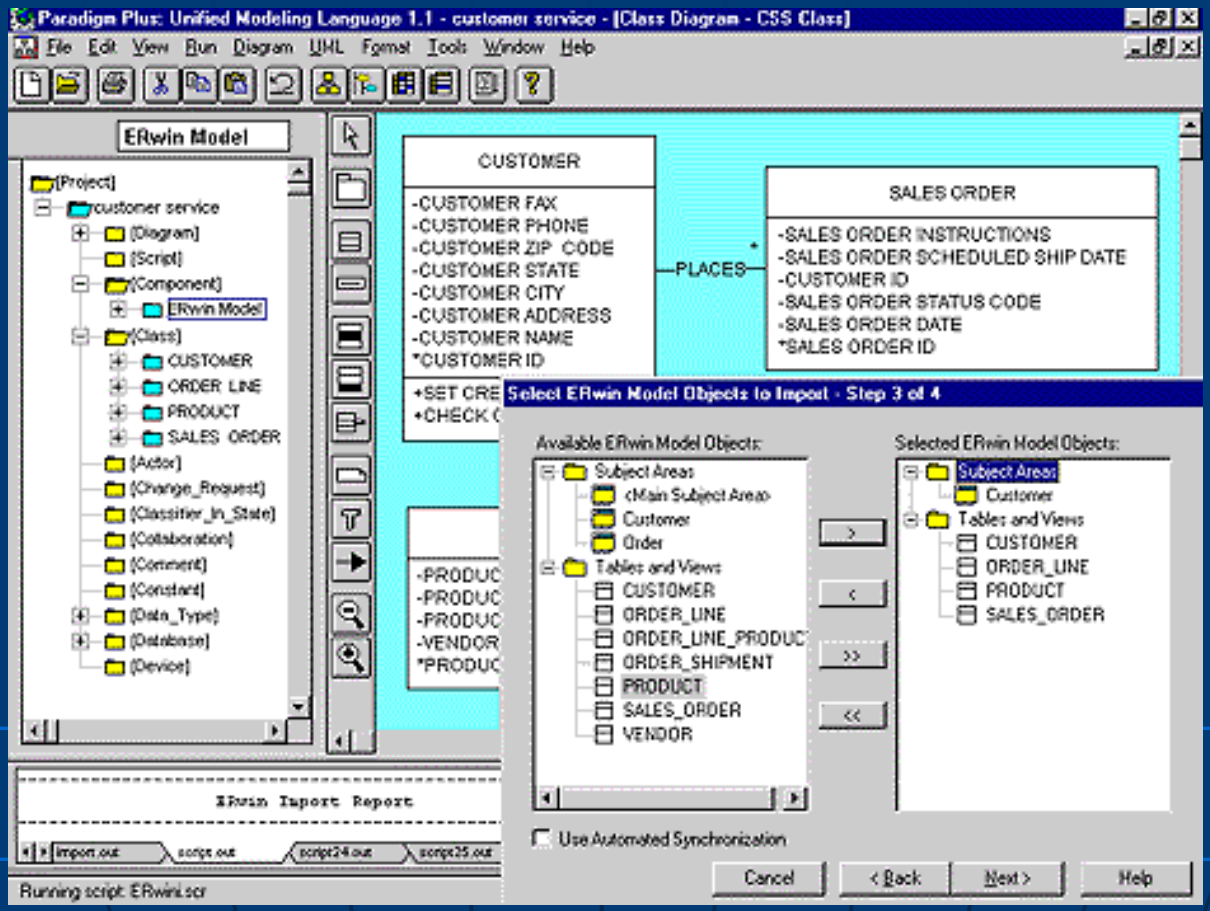
ORACLE Designer 2000



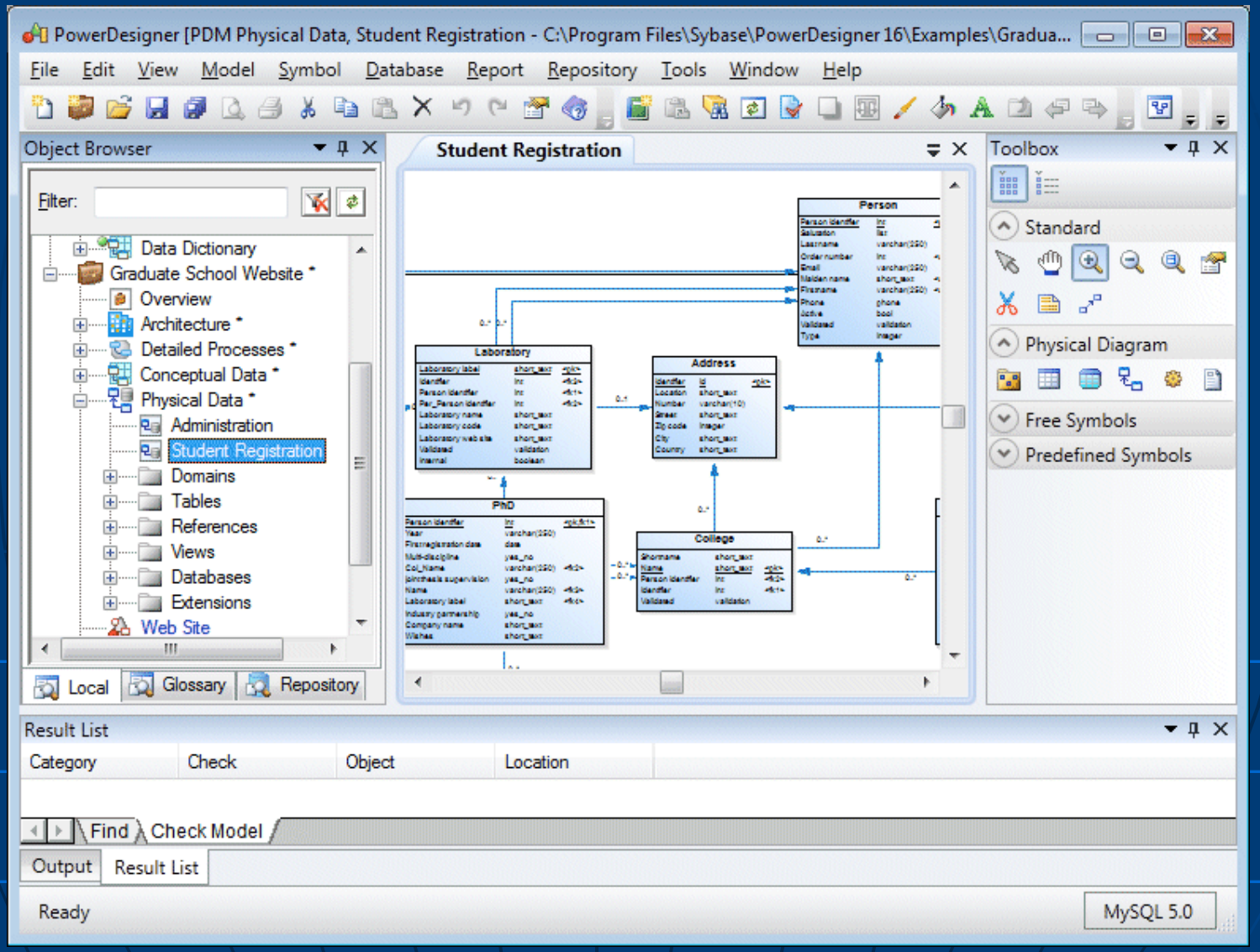
Rational Rose



Paradigm Plus



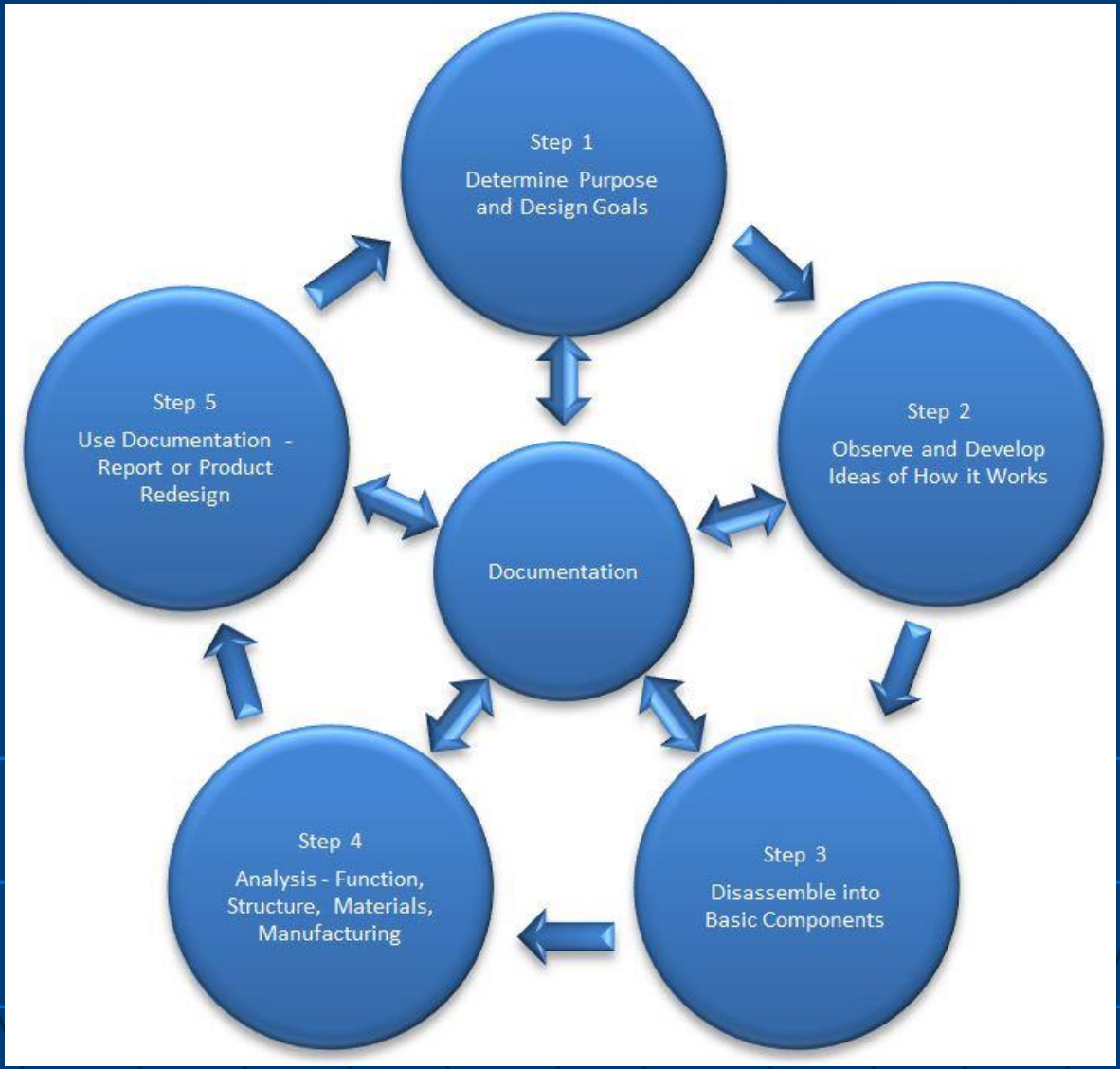
PowerDesigner



Reverzno inzenjerstvo

- Pojam ***reverznog inzenjerstva*** u razvoju programskih proizvoda se vezuje za postupak ručnog, ili automatizovanog generisanja projektnih i programskih specifikacija, na osnovu prethodno realizovanog programskog proizvoda.
- Nastanak pojma i tehnika reverznog inzenjerstva je motivisan sledećom situacijom. U mnogim organizacijama uložena je velika količina materijalnih, finansijskih i ljudskih resursa u razvoj i eksploataciju informacionih sistema.
- Jedan od zahteva, koji se postavlja prilikom prelaska na nove informacione tehnologije, jeste da se u razvoj inovirane verzije informacionog sistema ne kreće ispočetka, već da se sav ulozeni napor, iskustvo, postojeća rešenja i resursi što bolje iskoriste, jer je to daleko ekonomičnije od ponovnog projektovanja informacionog sistema.
- Tehnike reverznog inzenjerstva se koriste za ostvarivanje sledećih ciljeva:
  - ✓ · generisanje projektne i programerske dokumentacije za aktuelnu verziju programskog proizvoda, za koju prethodno takva dokumentacija nije uradjena, u cilju stvaranja osnova za održavanje tekuće verzije tog programskog proizvoda ili
  - ✓ · generisanje projektnih i programskih specifikacija programskog proizvoda u formatu "razumljivom" CASE proizvodu, pomoću kojeg se želi razviti nova verzija tog programskog proizvoda.

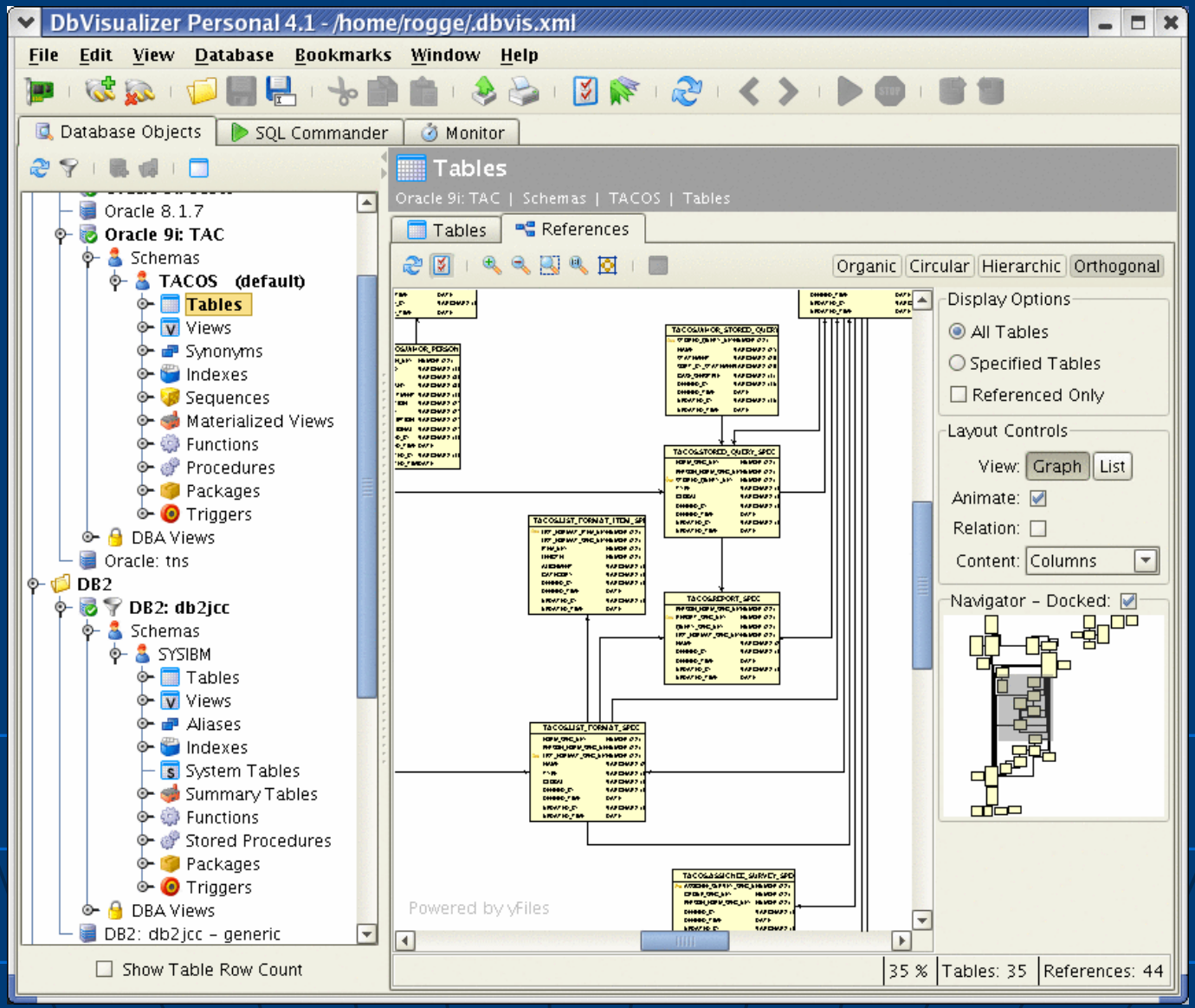
Reverzno inzenjerstvo



Reverzno inzenjerstvo

- U domenu informacionih sistema, tehnike reverznog inzenjerstva se primenjuju za ostvarenje sledećih zadataka:
- Generisanje implementacionog opisa šeme baze podataka, na osnovu nekih od sledećih parametara:
  - · realnih podataka koji postoje u bazi,
  - · stanja recnika podataka konkretnog sistema za upravljanje bazama podataka, pod kojim je posmatrana baza podataka realizovana, ili
  - · opisa datoteka i formata slogova u programskom kodu aplikacija tekuće verzije informacionog sistema,
  - · generisanje konceptualne šeme baze podataka, na osnovu implementacione šeme baze podataka i
  - · generisanje programskih specifikacija (struktura menija, ekranskih formi ili formi za izveštaje, podšema i procedurne logike) na osnovu programskih kodova aplikacija tekuće verzije informacionog sistema.

Reverzno inzenjerstvo



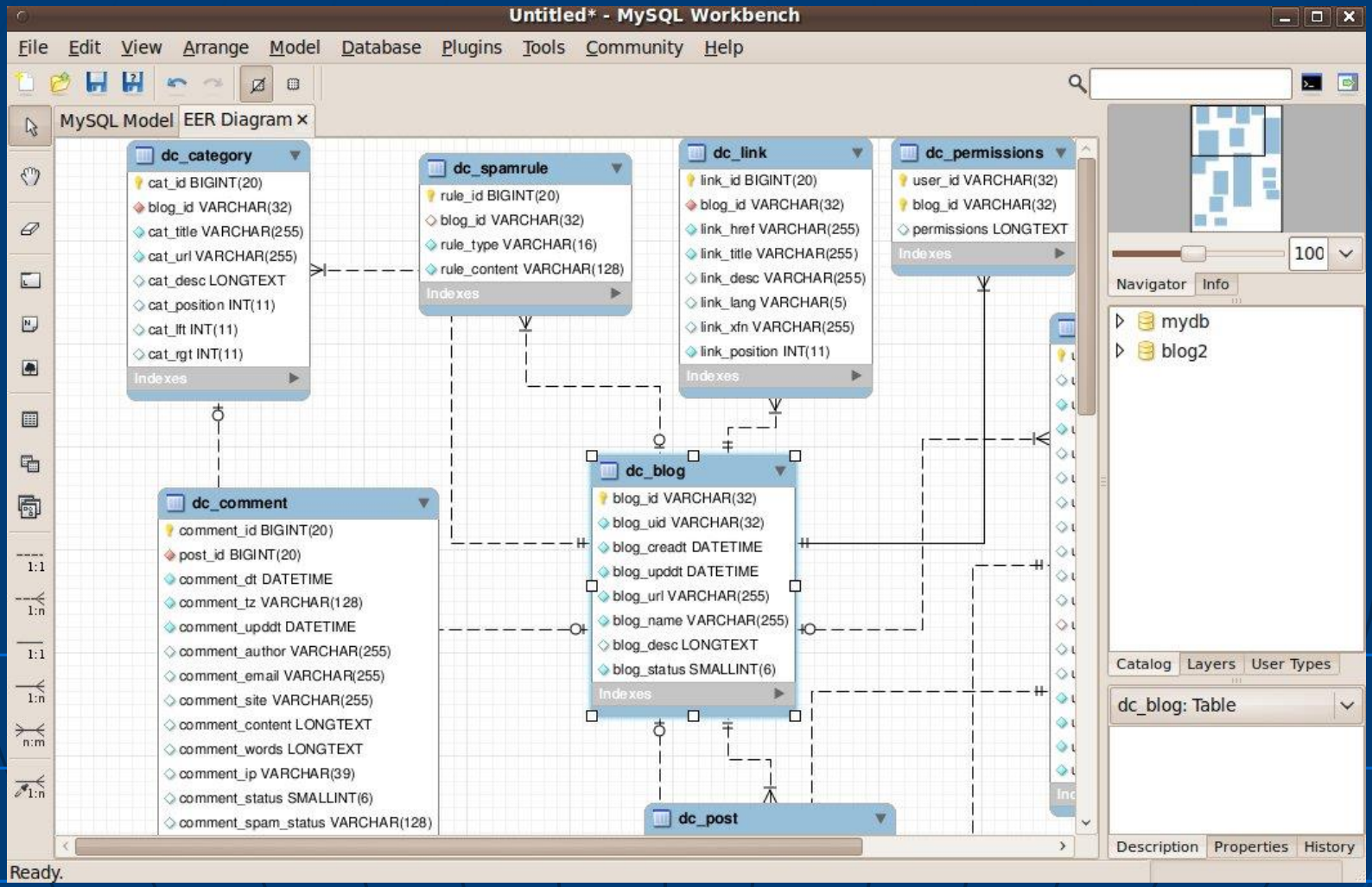
Reverzno inzenjerstvo

- Izbor tehnike reverznog inženjerstva i njena automatizovana primena u velikoj meri zavisi, kako od prirode konkretnog zadataka koji se rešava, tako i od kvaliteta, tj. "informativnosti" ulazne specifikacije, na koju se tehnika reverznog inženjerstva primjenjuje.

- Samim tim je i kvalitet generisanog rezultata u reverznom inženjerstvu bitno određen kvalitetom ulazne specifikacije. Ukoliko se, na primer, tehnika reverznog inženjerstva primenjuje za generisanje implementacione šeme BP, najbolji rezultat se, u opštem slučaju, može očekivati ukoliko se kao ulazna specifikacija koriste podaci iz rečnika podataka sistema za upravljanje bazama podataka, a najlošiji ako se kao ulazna specifikacija koriste samo realni podaci iz BP.

- Ova konstatacija, medjutim, ne mora biti uvek tačna. Ukoliko rečnik podataka konkretnog sistema za upravljanje bazama podataka sam po sebi nije dovoljno informativan, ili se u tom rečniku, iz nekog razloga, ne nalaze svi potrebni podaci, tada ni izlazni rezultat reverznog inženjerstva ne može biti dobar.

Reverzno inzenjerstvo



Testiranje softvera

Postoji li nacin da razvijamo softver koji ce biti kvalitetan, lak za održavanje i zadovoljiti zahteve tržišta?! Postoji li nacin da taj „nacin“ bude jedinstven, ponovljiv i dokumentovan i da nam umanji troškove prekasno spoznatih nedostataka proizvoda?!

Otuda primena standarda u informacionim tehnologijama, a posebno onih koji se odnose na kvalitet softverskih proizvoda.

Negde izmedu elemenata uspeha i svuda u pozadini procesa razvija se nova i sve popularnija disciplina. Ono što se proizvodi za druge, mora se probati i okusiti na sopstvenom primeru, otuda testiranje softvera.

Danas, kada je tržište nemilosrdno, ostaje samo jedan siguran nacin da se postignu željeni ciljevi i da se obezbedi bar približno kontinuitet uspeha u softverskoj industriji.

Softverski proizvod se, kao najnežnija biljka, planira, razvija, prati i kontroliše, a na kraju posebno oblikuje, kako bi se isporucio kupcu i zadovoljio zahteve. Kao i obicno, sve se koncentriše oko poslednje karike u lancu - kupca. Cini se da je pored svih napora, dokumentacije, pravila, procesa, pored sve organizacije i planova, najvažnija rec onih koji proizvod zaista koriste. Recept za uspeh je jako važan, a sadržaj recepta cine: standard, kvalitet, testiranje i razvoj softvera i kupac. Nekad i izmešanim redosledom.

Testiranje softvera

Primena standarda u informacionim tehnologijama:

Standardizacija u informacionim tehnologijama doprinosi efikasnijem uspostavljanju informacionih funkcija, njihovoj vecoj stabinosti i lakšoj tranziciji. Primena medunarodnih, nacionalnih i internih standarda u procesu razvoja softverskih proizvoda stvara uslove za razvoj efikasnog, ekonomicnog, pouzdanog i sigurnog softverskog proizvoda.

Standardizacijom procesa razvoja softvera, njegovim planiranjem, kvantifikovanjem i pracenjem, dokumentovanjem i neprekidnim poboljšanjem i unapredenjem stvaraju se preduslovi za realizaciju softverskih proizvoda definisanog kvaliteta. Dobro dokumentovani sistem, u skladu sa standardima, je lako zamenjiv, prenosiv sa jedne softverske i hardverske platforme na drugu i štiti investiciju.

Da bi se razvio kvalitetan informacioni sistem neophodno je da se njegov razvoj zasniva na usvojenim standardima (medunarodnim, nacionalnim, internim) i da se vrše mnogobrojna vrednovanja tokom njegovog životnog ciklusa. Vrednovanja ukljucuju:

- ✓ vrednovanje korišcenih softverskih proizvoda, meduproizvoda u vakoj fazi životnog
- ✓ ciklusa i, krajnjih proizvoda tj.
- ✓ instaliranog softvera i dokumentacije.

Testiranje softvera

Vrednovanje i standardizacija alata koji se koriste u procesu razvoja informacionih sistema stvaraju mogucnost da kvalitet samog procesa, kao i krajnjih proizvoda, bude na željenom i ocekivanom nivou. Posmatrano sa stanovišta složenih informacionih sistema, u cijem razvoju i implementaciji ucestvuje više organizacija, primena standarda, ne samo da obezbeduje odgovarajuci kvalitet krajnjeg proizvoda i procesa razvoja, nego stvara mogucnosti za razmenu projekata izmedu pojedinih organizacija, olakšava obuku korisnika i stvara uslove za zajednicki rad na projektima predstavnika razlicitih organizacija.

Ekspanziju razvoja softvera, u razlicitim oblastima, pratio je i razvoj standarda, procedura, metoda i alata za razvoj i upravljanje softverom. Raznovrsnost je stvorila moguce poteškoce u upravljanju softverom, posebno kada se radi o softveru koji je vezan za proizvode ili usluge. Ova pojava uslovila je potrebu da se za softversku disciplinu definiše zajednicki okvir koji bi poslužio svima koji se bave softverom da „govore istim jezikom“ u razvoju, projektovanju i upravljanju softverom u njihovim okruženjima.

Standard je tako projektovan da se može prilagoditi potrebi organizacije, projekta ili specificnoj primeni. Može se primeniti u slucajevima kada je softver samostalan entitet ili sastavni deo složenog sistema.

Testiranje softvera

Primena standarda na kvalitet softvera

Stanje u softverskoj tehnologiji još ne obezbeduje dovoljno dobru i široko prihvaćenu šemu za ocenjivanje kvaliteta softverskih proizvoda. Od 1976.godine uradjeno je mnogo od strane pojedinaca na definisanju osnove za kvalitet softvera. Usvojeni su i prošireni mnogi modeli (McCall-a, Boehm-a, US Air Force i drugi).

Duže vreme pouzdanost je bila jedini način za merenje kvaliteta. Vremenom su kroz razne studije predloženi i drugi modeli. Iako su studije bile korisne, stvorile su zabunu, zbog toga što su ponudjeni mnogi aspekti kvaliteta. Dakle, postojala je potreba za jednim modelom standarda.

Iz tog razloga je ISO/IEC JTC1 počeo da radi na usaglašavanju i ohrabruje opšte prihvacenu standardizaciju. Prva razmatranja potiču iz 1978. godine, a 1985.godine počeo je razvoj standarda ISO/IEC 9126. Predloženi modeli prvenstveno uvode svojstva softvera koja zavise od aspekata primene ili implementacije, radi opisivanja kvaliteta softvera.

Prvi korak ISO tehničkog komiteta u sistematskom uredjivanju svojstava je propao zbog nedostatka definicija. Eksperti su razlicito interpretirali termine. Sve pominjane strukture nisu imale zajedničku osnovu. Na kraju je odlučeno da je najbolji način za postavljanje medunarodnog standarda postavljanje skupa karakteristika koje se zasnivaju na definiciji kvaliteta. Konacno, nastali su i prvi standardi koji se odnose na kvalitet proizvoda i kvalitet softvera: ISO 9000, ISO/IEC 9126 i mnogi drugi prateći standardi.

Testiranje softvera

Pojam kvaliteta softvera

Kvalitet predstavlja sposobnost da se proizvede softver koji zadovoljava ili nadmašuje postavljene zahteve (prema definisanim merljivim kriterijumima) i koji je proizveden definisanim procesom.Ovaj proces ne svodi se samo na zadovoljenje definisanih zahteva, već se u okviru njega moraju definisati mere i kriterijumi koji usmeravaju proces postizanja kvaliteta. Potrebno je usvojiti pravila za jedan ponovljiv i upravljiv proces čiji proizvodi će dostizati odredjeni nivo kaliteta.

Jedna od definicija koja se pominje u softverskoj literaturi je i definicija sa stanovišta potrošača. Potrošač definiše kvalitet kao meru zadovoljavanja potreba kupca od strane proizvoda ili usluge. Drugom rečju, upotrebnim kvalitetom.

Najčešce zablude o kvalitetu:

1. Kvalitet može biti naknadno dodat i“ utestiran“ – kvalitet mora biti opisan i ugradjen u proces stvaranja proizvoda.

2. Kvalitet dolazi sam od sebe – kalitet ne nastaje tek tako. Proces razvoja mora se definisati, sprovoditi i kontrolisati kako bi se dostigao odredjeni nivo kvaliteta.

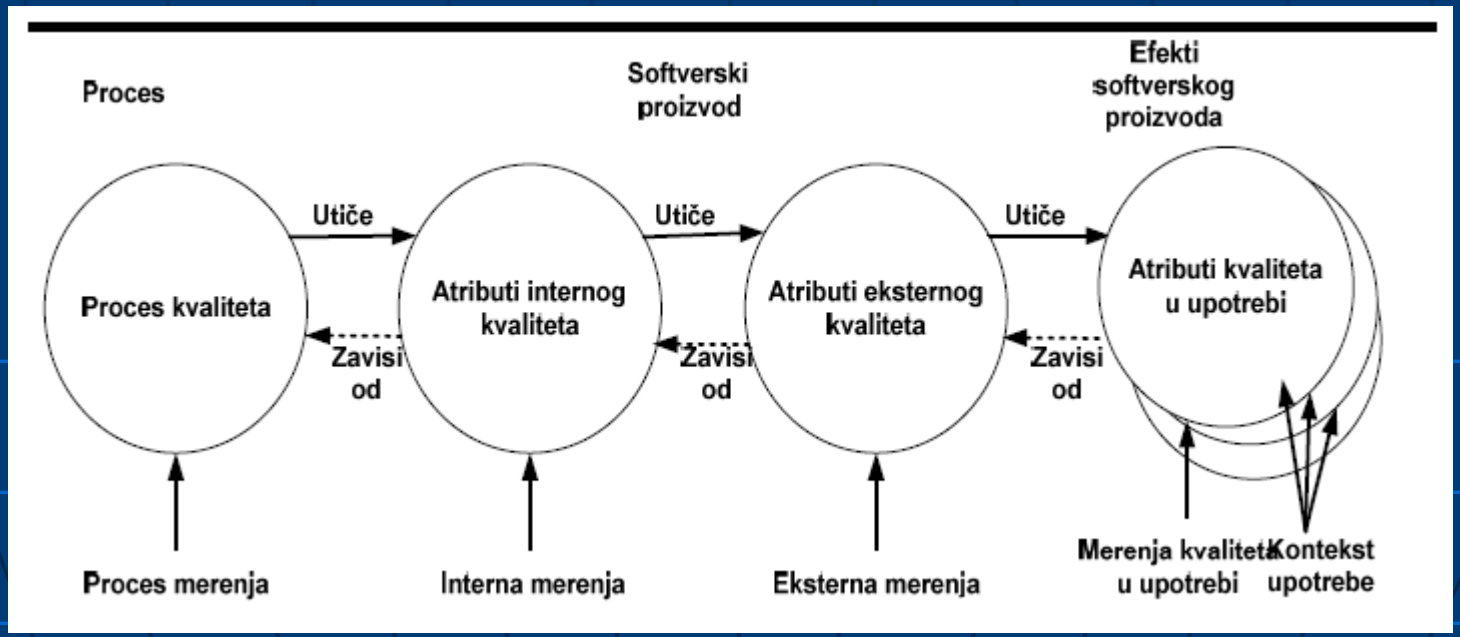
3. Kvalitet je jednodimenzionalna karakteristika i svakom znači isto- kvalitet ima više dimenzija od kojih su najvažnije: funkcionalnost, pouzdanost, upotrebljivost, efikasnost, stepen podrške.

Svaku od dimenzija prati odgovarajući tip testiranja softvera.

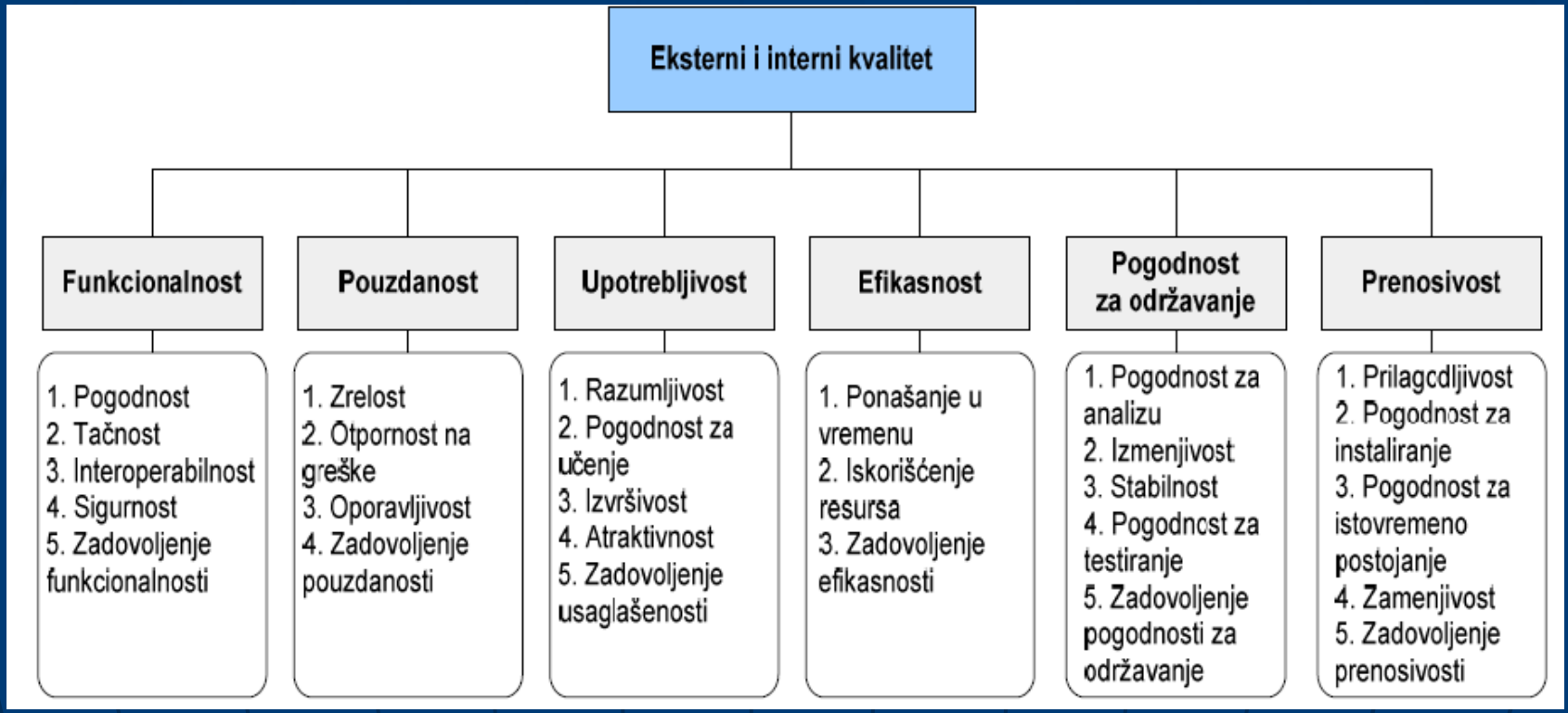
Testiranje softvera

Kako se meri kvalitet?

Kada se govori o kvalitetu i zadovoljenju zahteva za kvalitetom, tada se ističe pozicija kupca ili naručioca tj. korisnika kome se proizvod distribuira. Korisnički zahtevi za kvalitetom odnose se na kvalitet proizvoda u upotrebi. U toku životnog ciklusa proizvoda vrši se i vrednovanje softverskog proizvoda, kako bi se zadovoljenje zahteva za kvalitetom ugradilo u proces razvoja softvera. Kvalitet se procenjuje merenjem intenih (obično statičko merenje medjuproizvoda) i eksternih (merenje ponašanja koda kada se izvršava) atributa ili merenjem atributa kvaliteta u upotrebi. Osnovu za merenje kvaliteta pruža ovaj model kvaliteta.



Testiranje softvera



Model kvaliteta za eksterni i interni kvalitet

Za svaku karakteristiku i podkarakteristiku, sposobnost softvera je odredjena skupom internih atributa koji mogu da se izmere. Karakteristike i podkarakteristike mogu da se mere eksterno stepenom sposobnosti koji je obezbeden sistemom koji sadrži softver.

Testiranje softvera

Model kvaliteta u upotrebi

Svojstva kvaliteta u upotrebi su podeljena na četiri podkarakteristike:

- ***efektivnost*** (sposobnost softverskog proizvoda da omogući korisnicima postizanje ciljeva sa tacnošću i kompletnošcu u kontekstu spečificne upotrebe)

- ***produktivnost*** (sposobnost softverskog proizvoda da omogući korisnicima upotrebu odgovarajuće količine resursa u vezi sa efektivnošću postignutom u specificiranom kontekstu upotrebe)

- ***bezbednost*** (sposobnost softverskog proizvoda za postizanje prihvatljivih nivoa rizika štete ljudima, poslu, softveru, imovini ili okruženju u kontekstu specifične upotrebe)

- ***zadovoljstvo*** (sposobnost softverskog proizvoda da zadovolji korisnike u kontekstu specifične upotrebe)

Testiranje softvera

Pojam testiranja softvera

Testiranje softvera je proces za merenje kvaliteta softvera. Kvalitet se obično vezuje za pojmove tačnosti, potpunosti, sigurnosti, ali takodje uključuje i neke zahteve propisane ISO standardom, kao i tehničke zahteve ISO 9126: karakteristike, pouzdanost, efikasnost, portabilnost, istrajnost, kompatibilnost i korisnost.

Testiranje je proces tehničke istrage, čiji je cilj da otkrije informacije vezane za kvalitet proizvoda, naravno, sa poštovanjem konteksta funkcionisanja proizvoda. Ovaj proces uključuje izvršavanje programa ili aplikacije sa namerom pronalaženja grešaka. Kvalitet nije apsolutna, već vrednost za neku osobu. Kada se ovo uzme u obzir testiranje se ne može uzeti kao parametar za uspostavljanje kontrole nad softverom; testiranje obezbeduje kritiku i komparaciju za stanje i ponašanje proizvoda u odnosu na specifikaciju.

Postoje mnogi pristupi testiranju softvera, ali je efektivno testiranje kompleksnih proizvoda u osnovi proces istrage, a ne samo pitanje kreiranja i praćenja rutinske procedure.

Jedna od definicija testiranja je: *„Proces ispitivanja softvera u cilju procene istog“,* gde se ispitivanje odnosi na operacije koje tester pokušava da izvrši nad proizvodom, a proizvod odgovara na to ponašanjem izazvanim probama testera. Testiranje se vezuje i za dinamičko analiziranje proizvoda kroz faze razvoja. Za statičko testiranje smatra se istraživanje, a za dinamičko pokretanje programa uz predefinisani skup test scenarija na odredenom nivou razvoja.

Testiranje softvera

Istorija testiranja softvera

U dokumentu Gelperina i Hetzela iz 1988. godine pominje se podela istorije testiranja na sledeće faze:

Perod od	Period do	Faza	Opis
-	1956	***Debagovanje***	Programe je pisao, a zatim i testirao, programer, do trenutka uklanjanja svih bug-ova. Testovi su kreirani ad hoc, na osnovu iskustva programera i fokusirani na operativnost sistema. Ne postoji jasna granica između testiranja i razvoja.
1957	1978	***Demonstracija***	Debagovanje se i dalje izvodilo kako bi sistem proradio, ali je dodat još jedan nivo testiranja - da se demonstrira da softver ne samo radi, već da radi ono što treba.
1979	1983	***Destrukcija***	Fokus testiranja se pomera sa obezbeđenja da softver radi ono što treba, na otkrivanje grešaka u sistemu.
1983	1987	***Evaluacija***	Testiranje postaje aktivnost na kraju svakog nivoa u razvoju životnog ciklusa, jer se prihvata stanovište da što ranije otkrivanje grešaka u sistemu izaziva manje troškove.
1988	-	***Prevencija***	Ideja je da se spreče greške u bilo kom stadijumu životnog ciklusa proizvoda testiranjem svakog nivoa. Testiranje se fokusira na ispravljanje grešaka.

Testiranje softvera

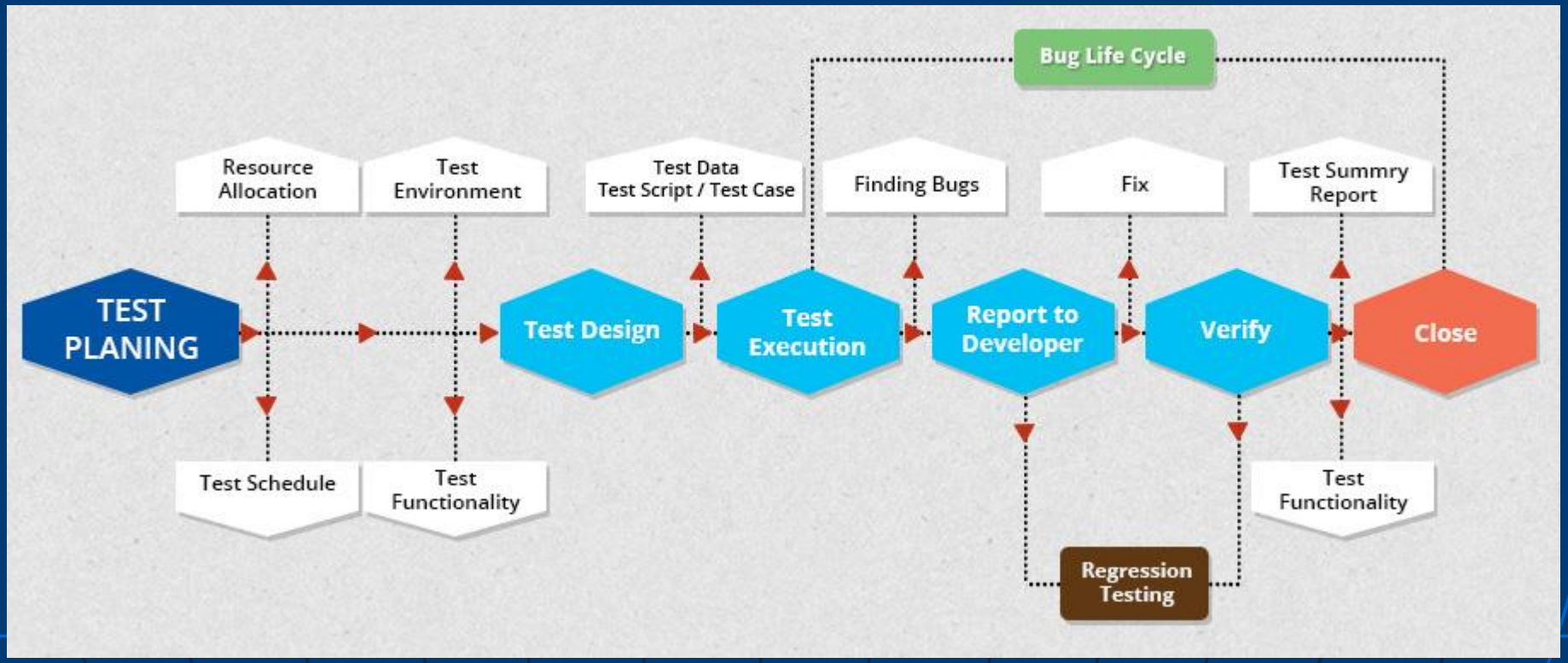
Proces testiranja

Testiranje se sastoji od sledećih aktivnosti:

1. ***Planiranje testiranja*** (odredjuje se predmet, cilj i razlog testiranja (na osnovu zahteva, modela i drugih ulaza testiranja), gde se testiranje vrši, kada se vrši i ko izvršava testove)
2. ***Dizajn testova*** (odreduje kako se sprovodi testiranje na osnovu artefakta tj. Test primera)
3. ***Implementacija testova*** (pravljenje višestruko upotrebljivih test scriptova koji realizuju test primere)
4. ***Izvršavanje testova*** (izvršavanje implementacije testa radi provere funkcionalnosti sistema)
5. ***Evaluacija testova*** (procena testova tj. validnosti izvršavanja testa, analiza izlaza, pregled zbirnih rezultata, uticaj promene zahteva i ulaza na plan testiranja)

Svaka od ovih aktivnosti ima ulaze i izlaze. U procesu testiranja koriste se i različiti alati koj pospešuju proces i daju bolji uvid u rezultate (npr.Rational, Test Complete i sl.).

Testiranje softvera



Testiranje softvera

Tipovi testiranja

Inspekcije, recenzije i pregledi su specificne tehnike fokusirane na ocenjivanje artefakata (pogodno za dokumentaciju, programski kod) i efikasne su metode za poboljšanje kvaliteta i razvojnog procesa proizvoda. Sprovode se na sastancima, gde jedan od učesnika ima ulogu vodećeg, a ostali uloge zapisičar a(beleži pitanja, predloge, probleme i sl.). Ove tehnike opisuju se u okviru IEEE standarda na sledeći nacin:

Recenzija (review) – formalni stastanak na kome se artefakt ili skup artefakata predstavlja korisniku ili drugim stranama koje učestvuju u procesu odobravanja ili komentarisanja.

Inspekcija (inspection) – fomalna tehnika procene u kojoj se artefakti detaljno pregledaju. Pregled vrše osobe koje nisu autori, da bi se lakše uočile greške i drugi problemi.

Pregledi (walkthrough) – Tehnika pregleda u kojoj autor upoznaje ostale članove tima sa elementima artefakta koji je izgradio. Ostali članovi učestvuju aktivno u raspravi.

Testiranje softvera

Tehnike testiranja

1. ***Jedinično testiranje (unit testing)*** – primenjuje se na pojedine klase, module ili komponente programskog koda. Ova tehnika deli se na tehnike bele i crne kutije.

2. ***Integraciono testiranje*** – primenjuje se na softverski sitem kao celinu.

3. ***U testiranja višeg reda spadaju:***

testiranje sigurnosti (security testing), testiranje kolicine podataka, testiranje upotrebljivosti, testiranje integriteta (integrity testing), test u stresnim uslovima(stress testing), etalonski test, test zagušenja, test opterecenja, test konfiguracije (configuration testing), test instalacije (installation testing)

4. *Regresiono testiranje* – na osnovu jednom razvijenog testa više puta se sprovodi testiranje softvera (tipično nakon neke izmene u softveru da bi se utvrdilo da nisu pokvarene funkcionalnosti softvera).

- Česta je i podela testiranja na:
  1. black-box, white-box i gray-box testiranje,
  2. manuelno i automatsko testiranje,
  3. staticko i dinamicko testiranje.

Testiranje softvera

1. Black-box ili funkcionalno testiranje počiva na test uslovima koji se razvijaju na nivou programa ili funkcionalnosti sistema. Tester zahteva informacije o ulaznim podacima i izlazima koje prati, ali zapravo ne zna kako sistem radi. Tester se fokusira na testiranje funkcionalnosti programa koje nisu u skladu sa specifikacijom. Na ovaj način, tester posmatra program kao crnu kutiju i ne zanima ga unutrašnja struktura programa ili sistema. Neki od primera ove vrste testiranja su: tabele odluka, testiranje opsega, testiranje graničnih vrednosti, testiranje integriteta baze, testiranje izuzetaka, nasumicno testiranje i sl. Najveća prednost ovog testiranja je da testovi obuhvataju ono što sistem ili program treba da radi i potpuno su prirodni i razumljivi za bilo koga.

White-box ili strukturno testiranje obuhvata ispitivanje interne strukture programa ili sistema. Testni podaci se dobijaju ispitivanjem logike programa ili sistema, bez brige o zahtevima koje treba da zadovolji. Tester poznaje internu strukutru i logiku programa. Specifični primeri koji spadaju u ovu kategoriju uključuju testiranje izraza, gransko testiranje, testiranje uslova i sl. Prednost ovog pristupa je da je potpun i da se fokusira na proizvedeni kod. Greške i nenamerni propusti, zbog poznavanja unutrašnje strukture ili logike, imaju više šanse da budu otkriveni. Loša karakteristika je da ne postoji način da se otkriju elementi koji nedostaju i greške vezane za osetljive podatke. Poslednji nedostatak je da ovaj tip testiranja ne može da obuhvati kompletnu logiku programa, jer bi to podrazumevalo kreiranje jako velikog broja testova.

Gray-box ili funkcionalno i srukturno testiranje predstavlja kombinaciju black i whitebox testiranja. Tester proučava zahteve i komunicira sa developerom kako bi razumeo internu strukturu sistema.

Testiranje softvera

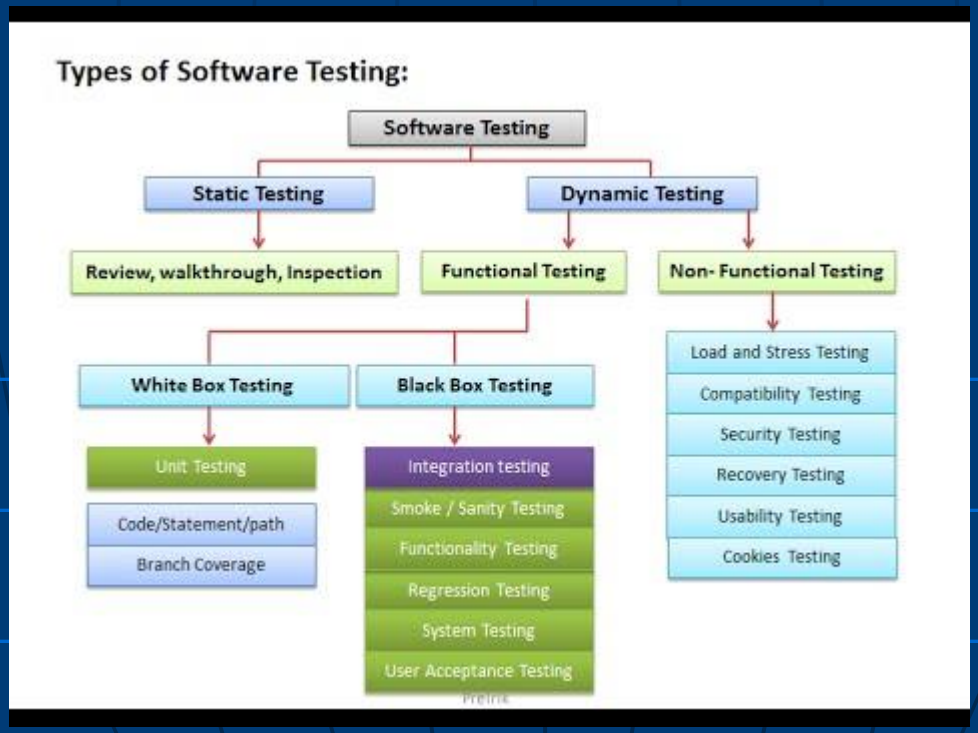
2. Manuelno i automatsko testiranje

Osnove manuelnog testiranja počivaju na tome da su njegovi nosioci ljudi i da nije implementirano na računaru; osnove automatskog, upravo na tome da je implementirano na računaru.

3. Statičko i dinamičko testiranje

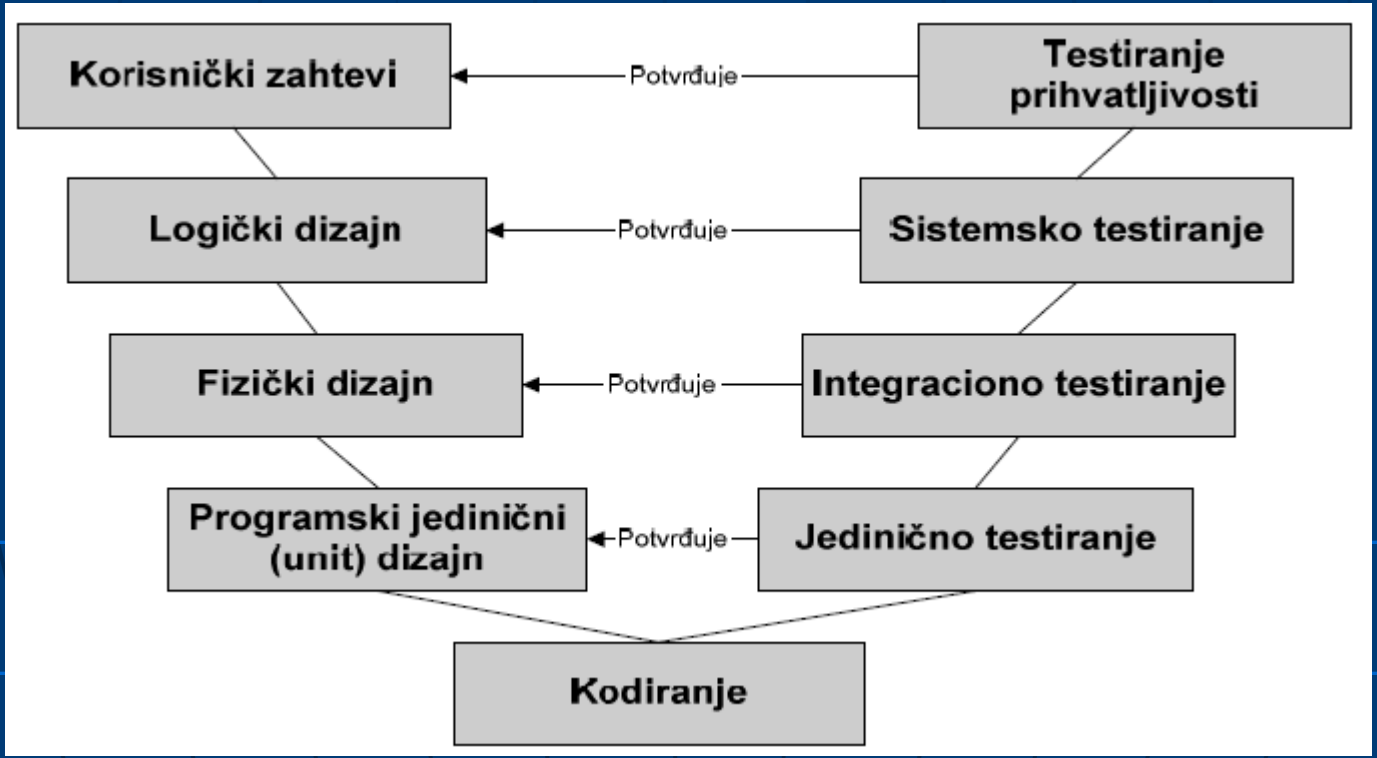
Statičko testiranje ne zavisi od vremena npr. provera sintakse, strukture i sl.

Dinamičke tehnike zavise od vremena i podrazumevaju izvršavanje specifičnog niza instrukcija na papiru ili racunaru.



Testiranje softvera

Testiranje životnog ciklusa softvera znači da se testiranje obavlja paralelno sa razvojnim ciklusom i da je kontinualan proces. Testiranje softvera treba da započne u ranoj fazi životnog ciklusa aplikacije, ne samo u tradicionalnoj validacionoj fazi testiranja, nakon što je kodiranje završeno. Testiranje mora biti integrisano u razvoj aplikacije. Kako bi se to ostvarilo, potrebno je da deo organizacije koji se bavi razvojem blisko komunicira sa funkcijom za obezbedjenje kvaliteta.

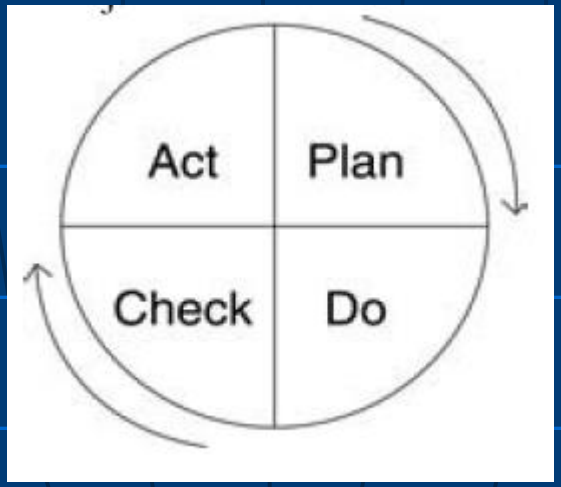


Testiranje softvera

Za vreme faze zahteva razvija se test plan. Plan obuhvata organizaciju test aktivnosti. Za vreme logičog, fizičkog i jediničnog programskog dizajna, razvija se test plan sa više detalja. U ovom periodu kreiraju se i test case-ovi.

Test case se definiše kao skup test podataka i test skriptova. ***Test skriptovi*** vode testera kroz test i obezbedjuju odvojena izvršavanja testa. Test uključuje očekivane rezultate koji potvrdjuju da je test ispunio cilj. Test skriptovi se izvršavaju i rezultati se analiziraju za vreme aplikacionog testiranja. Opšta forma test case-a tj. slučaja predstavlja dokument koji definiše ulaze, izlaze i scenario jednog prolaska kroz niz akcija u programu koje vode ka ostvarenju rezultata tj. specificnog cilja.

U ***spiralnom pristupu***, sekvencijalni koraci mogu, sa odredjenim nivoom varijabiliteta, nedostajati ili biti različiti. Spiralni pristup podrazumeva krug: planiraj, radi, proveri i deluj.



Testiranje softvera

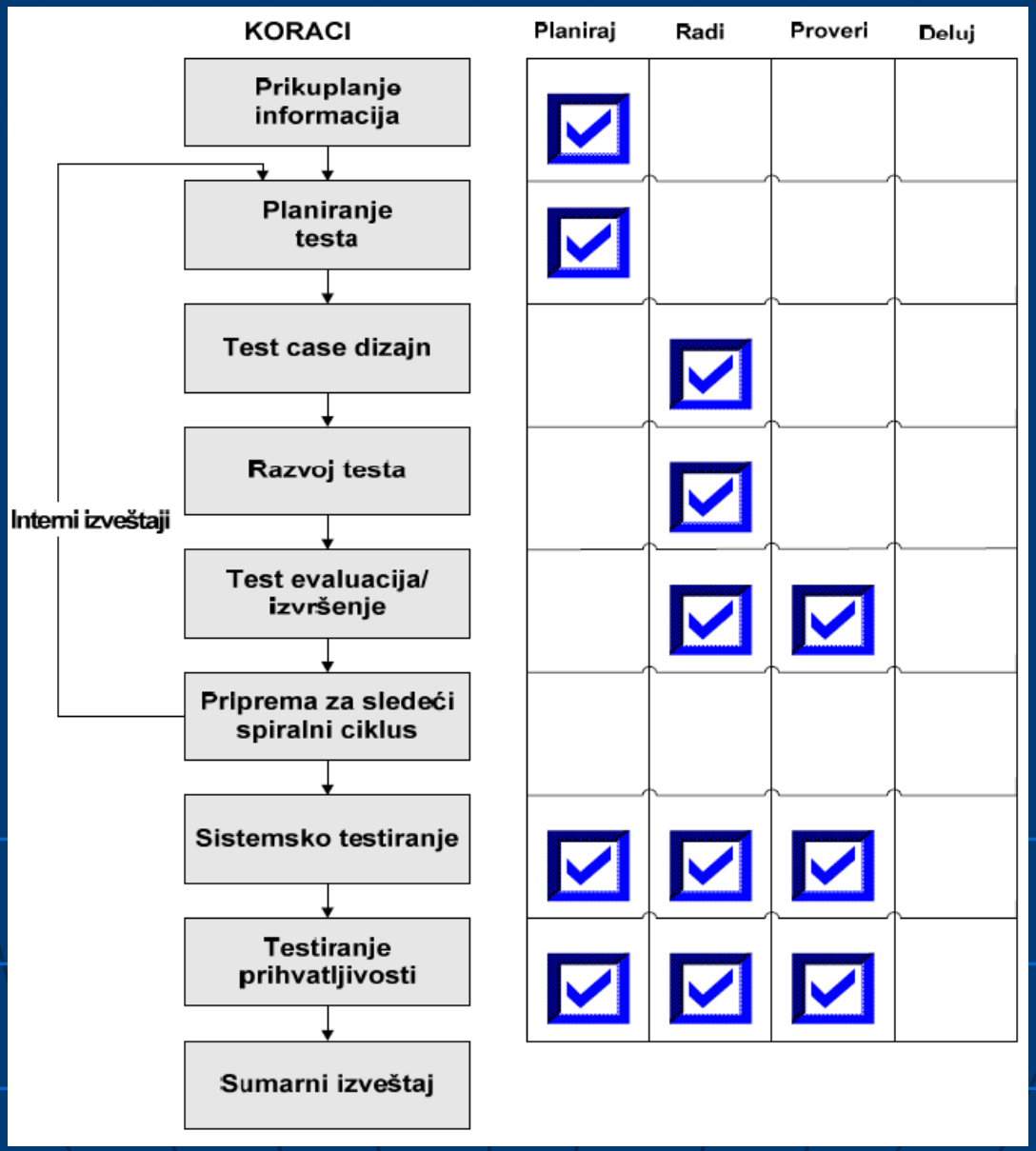
1. *Planiraj*: Korak razvoja test plana kao osnove za uspešno testiranje. Dobar test plan sadrži: uvod, celokupni prikaz plana, zahteve za testiranje, test procedure i detalje test plana.

2. ***Radi***: Ovaj korak sastoji se iz dizajna test case-ova, razvoja testova i izvršavanje testova. Ovaj korak opisuje kako da se dizajniraju i izvrše testovi koji se nalaze u test planu. Dizajn uključuje funkcionalne testove, GUI testove, podelu sitema i testove prihvatljivosti. Kada se završi faza dizajna, počinje razvoj testova koji ukljucuje izradu test skriptova i procedura koje obezbedjuju detalje za svaki test case. Ovaj korak uključuje i postavku testa, regresiono testiranje novih i starih testova i beleženje otkrivenih defekata.

3. Proveri: Korak uključuje metričke mere i analize. Važno je objaviti meduizveštaje testiranja. Uključeno je i beleženje rezultata i povezivanje sa test planom i ciljevima testa.

4. ***Deluj***: Ovaj korak uključuje pripremu za sledeću spiralnu iteraciju. Zahteva obradu funkcionalnih/GUI testova, test paketa, test case-ova, test skriptova, testova delova sistema i testova prihvatljivosti i izmene sistema za praćenje defekata i sistema verzije i kontrole, ako je to potrebno. Korak uključuje i odredjivanje mera za odgovarajuće radnje koje su povezane sa poslom koji nije izvršen po planu ili rezultate koji nisu očekivani. Primeri uključuju reevaluaciju test tima, test procedura i tehnoloških dimenzija testiranja.

Testiranje softvera



Testiranje softvera

Osiguranje kvaliteta softvera (Software Quality Assurance)

Pojam SQA odnosi se na sistematizovane aktivnosti koje obezbjeduju dokaz podobnosti krajnjeg softverskog proizvoda za upotrebu. Postiže se korišćenjem prihvaćenih pravila za kontrolu kvaliteta za proveru integriteta i produžavanje života softvera. Veza izmedu osiguranja kvaliteta, kontrole kvaliteta i funkcije provere softvera, kao i softverskog testiranja se često mešaju.

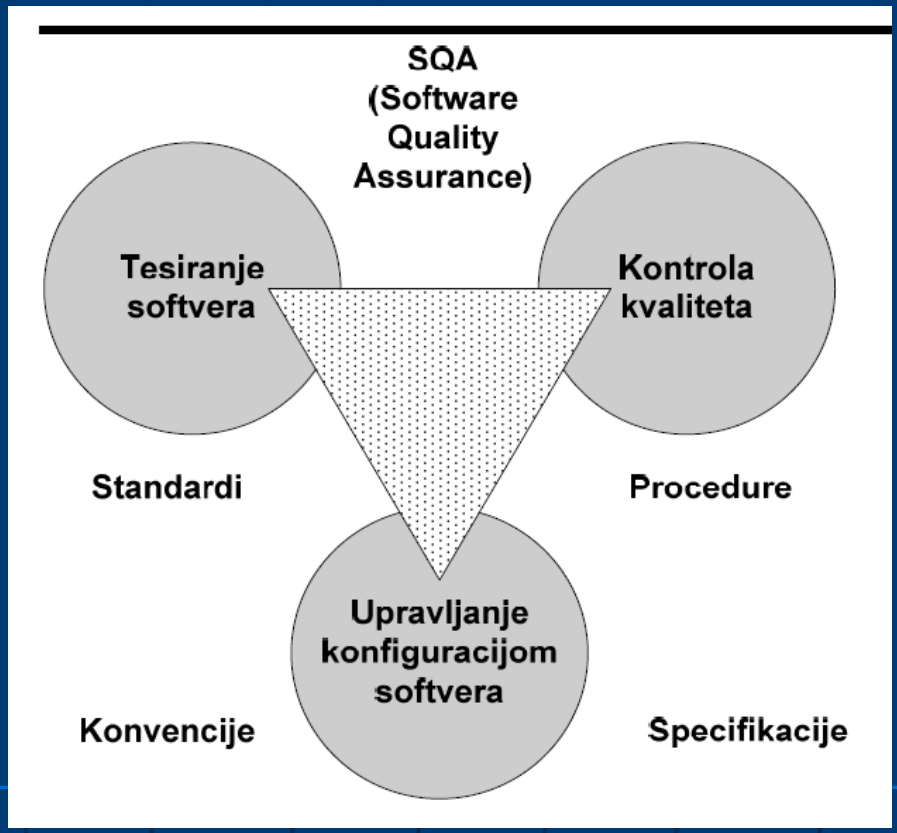
Obezbedenje kvaliteta obuhvata skup podržanih aktivnosti potrebnih da bi se obezbedilo adekvatno poverenje u sprovedeni proces i stalno unapredenje u cilju stvaranja proizvoda koji zadovoljavaju specifikacije i odgovaraju svrsi. Kontrola kvaliteta je proces u kom se kvalitet proizvoda poredi sa primenjenim standardima i akcijom koja se preduzima kada se otkrije neodgovarajuća karakteristika proizvoda. Provera je ispitivanje koje verifikuje slaganje sa planovima, politikom i procedurama.

Obezbedenje kvaliteta softvera je planiran napor da softver ispuni odredjene kriterijume i da ima dodatne specificne atribute npr. portabilnost, efikasnost, fleksibilnost. Kolekcija je aktivnosti i funkcija koje se koriste za praćenje i kontrolu projekta tako da specificni ciljevi budu postignuti sa željenim stepenom sigurnosti. Ovo nije odgovornost samo SQA funkcije, vec i menadžera projekta, lidera projekta, osoblja i korisnika.

Testiranje softvera

Komponente funkcije obezbedenja kvaliteta:

1. ***Testiranje softvera*** – odredjuje da li su zadovoljeni funkcionalni zahtevi. Testovi su jednako dobri kao i test case-ovi, ali se mogu preispitivati radi provere da li su svi testovi izvršeni sa svim mogućim ulazima i u svim mogućim stanjima sistema.
2. ***Kontrola kvaliteta*** – procesi i metode za nadgledanje posla i zadovoljenja zahteva.
3. ***Upravljanje konfiguracijom softvera*** – pronalaženje, praćenje i kontrolisanje promena softverskih elemenata sistema. Obuhvata identifikaciju komponenti sistema, kontrolu verzije, razvoj konfiguracije i kontrolu izmena.



Testiranje softvera

Najbolje prakse u testiranju softvera

Testiranje softvera je sve zastupljenije u savremenim organizacijama koje se bave razvojem softvera. U IBM istraživanju Centra za softverski inženjering opisuju se najbolje prakse u softverskom testiranju. Neke od njih podrazumevaju alate za testiranje, ali su uglavnom prakse u pravom smislu te reči. Istraživanje je obuhvatilo više softverskih organizacija koje su primenjivale ove prakse i prihvatile iste kao ključ uspeha.

Kolekcija praksi ima mnoge izvore. Neke poticu iz literature, neke iz prakticne primene i onog što se u praksi ceni. Prakse su podeljene na osnovne, utemeljene i inkrementalne.

Osnovne prakse predstavljaju korake na kojima se sve zasniva i koji kada se jednom savladaju postaju dobra osnova za dalji razvoj. U osnovne prakse se ubrajaju:

Testiranje softvera

1. Funkcionalne specifikacije – predstavljaju ključni deo mnogih procesa razvoja koji se pojavio sa Modelom vodopada. Često opisuju potrebe korisnika sitema, koje funkcije su neophodne, kao i zahtevane karakteristike ulaza i izlaza.

Funkcionalne specifikacije predstavljaju dokumentaciju koja opisuje zahtevano ponašanje sistema. Testeri ih koriste za zapisivanje test case-ova na nivou blackbox testiranja. Prednost posedovanja funkcionalne specifikacije je da se generisanje testova može odvijati paralelno sa razvojem koda.

Na ovaj nacin se ostvaruje paralelizam izvršenja i uklanjaju uskih grla u razvojnom procesu, jer su testovi gotovi kada se završi pisanje koda. Takode, na ovaj način se forsira odredeni nivo jasnosti iz perspektive dizajnera i arhitekte koji je neophodan za efikasan razvoj. Konačno, funkcionalne specifikacije postaju dokumentacija koju je moguće deliti sa korisnicima, kako bi uvideli još jednu dimenziju onog što se razvija.

2. Pregledi i inspekcije – Softverske inspekcije je izmislio Mike Fagan sredinom sedamdesetih godina u IBM-u i smatra se za jednu od najefikasnijih metoda debagovanja koda. Danas, 20 godina kasnije, postoji literatura, alati i konsalting organizacije koje se bave praksom softverskih inspekcija.

Testiranje softvera

3. Formalni ulaz i izlazni kriterijum- Ovaj pojam potiče iz evolucije Modela vodopada i modela zvanog ETVX (Entry criteria, Tasks, Verifications and Validations, Exit criteria). Ideja je da svaki korak procesa ima precizan ulaz i precizan izlazni kriterijum. Kriterijumi se definišu u toku procesa razvoja i nadgledaju od strane menadžmenta.

4. Funkcionalni test – varijacije – Većina funkcionalnih testova napisana je kao black- box test na osnovu funkcionalne specifikacije. Broj generisanih testova obično je varijacija ulaznih parova i izlaznih uslova. Varijacija se odnosi na odredjenu kombinaciju ulaznih, za dostizanje odredenih izlaznih uslova. Pisanje funkcionalnih testova uključuje i pisanje različitih varijacija, kako bi se pokrilo što više stanja programa. Najbolja praksa uključuje razumevanje načina pisanja varijacija i kreiranja adekvatnog pokrića za test funkcije.

5. Višeplatformsko testiranje – Mnogi proizvodi dizajnirani su da funkcionišu na različitim platformama koje predstavljaju dodatni teret pri lansiranju i testiranju proizvoda. Kada se kod prenosi sa jedne platforme na drugu neophodne su modifikacije kako bi se unapredile ili održale performanse.

Testiranje softvera

6. Interne bete – Beta verzija je prva verzija programa koja je lansirana van organizacije ili zajednice koja razvija softver u cilju procene ili black/gray-box testiranja. Ideja ***bete*** je da se proizvod isporuči ograničenom broju kupaca i da se dobije povratna veza, kako bi se ispravili problemi širih razmera. Za veće kompanije, mnogi proizvodi se koriste interno, na ovaj način formirajuci tzv. Beta publiku. Tehnike koje podržavaju interne beta testove povećavaju efikasnost korišćenja unutrašnjih resursa. Reduciraju se i troškovi i izdaci u odnosu na eksternu betu.

7. Automatsko izvršavanje testa - Cilj automatskog testa je da minimizira količinu manuelnog rada koji se utroši pri izvršavanju testa i rezultira većom pokrivenošću tj. većim brojem test case-ova.

8. Beta programi – odnose se na interne beta verzije.

9. Nocni bildovi – obično predstavljaju verzije koje se automatski kreiraju od strane kontrolnog i sistema za buildove, obično preko noćći, omogućavajuci testerima da odmah testiraju izmene. Koncept noćnih bildova je u modi već duže vreme. Podrazumeva učestale buildove, za razliku od onih koji se rade na dnevnom nivou. Prednost je da ako se dese veće regresije zbog napravljenih grešaka, one se brzo otkrivaju, a noviji release-ovi softvera dostupni su ranije i testerima i developerima.

Testiranje softvera

Inkrementalne prakse obuhvataju:

1. ***Povezivanje testera i developera –*** Poznato je da povezivanje testera i developera unapreduje i test case-ove i kod koji se razvija. Ekstremni slučaj ove prakse je Microsoft, gde svaki developer ima svog testera. Ova praksa obuhvata razgraničavanje koje vrste povezivanja imaju efekta, i u kojim se okruženjima mogu upotrebiti.

2. Pokrice koda – Koncept se zasniva na strukturnoj dimenziji koda. Pokriće koda odnosi se na numeričko merenje koje odreduje elemente koda koji su testirani. Postoje mnoge metrike: izrazi, grane i podaci na koje se odnosi ovaj pojam. Postoji i nekoliko alata koji pomažu merenju i obezbedjuju dodatnu pomoć da se odrede elementi koji nisu obuhvaćeni. Ova oblast je takodje predmet debate već nekoliko decenija. Ova praksa, dakle, treba da sadrži informacije o alatima i metodama koje se sprovode pri testiranju koda i prate rezultate pozitivnih iskustava.

3. Automatski generator okruženja(Drake) – Jako zahtevan deo u smislu vremenskih resursa je postavljanje testnih okruženja kako bi se sprovodilo testiranje. Ovo vreme može biti još duže ako postoji više operativnih sistema, verzija i koda koji se izvršava na više platformi. Ovo igra veliku ulogu u sistemskom testiranju. Alati koji mogu automatski postaviti okruženja, izvršavati test case-ove, snimati rezultate i potom automatski preći na novo okruženje, imaju jako veliku vrednost. IBM je razvio alat pod nazivom DRAKE koji precizno radi. Ova praksa obuhvata probleme, alate i tehnike koji su povezani sa postavkom okruženja, padovima okruženja i automatskim pokretanjem test case-ova.

Testiranje softvera

Inkrementalne prakse obuhvataju:

4. *Testiranje u cilju pomaganja isporuci na zahtev* – Ova praksa je ideja Microsofta u kojoj se posmatra test proces kao nešto što omogućava kasne promene i prilagodjavanje zahtevima tržišta. Ona na neki način menja ulogu testiranja u onu koja obezbedjuje dobru regresivnu sposobnost i rad sa kasnim izmenama koje još uvek ne narušavaju proizvod i planirani izlazak na tržište.

Ubraja se u filozofska gledišta testiranja, koji postavlja testiranje u novu dimenziju. Praksa mora identifikovati kako da se razradi ovaj koncept u organizacijama i sa proizvodima na specificnim tržištima. Može se primeniti u elektronskoj trgovini, gde je veća i interakcija kupaca i veći kompetitivni pritisak.

5. ***Dijagram stanja zadataka(Tucson)*** – Obuhvata funkcionalne operacije aplikacije ili modula u obliku dijagrama prelaza stanja. Prednosti ovog oblika su da omogućava automatsko kreiranje test case-ova ili kreiranje metrika koje su bliže funkcionalnoj dekopoziciji aplikacije. Postoje i alati koji podržavaju ovu praksu. Problemi su najcešće dobijanje funkcionalnog izgleda proizvoda koji ne postoji u kompjuterizovanoj ili dokumentovanoj formi i kreiranje dijagrama prelaza stanja.

Jedan od alata Test Master zaista koristi ove dijagrame za generisanje funkcionalnih testova.

6. ***Simulacija otkaza memorijskih resursa*** – Praksa se odnosi na odredeni bug, gubitak memorije zbog lošeg upravljanja heap-om ili nedostatka garbage collection-a. Ovo je veliki problem za mnoge C programe i Unix aplikacije. Postoji i na drugim platformama i jezicima. Postoje komercijalni alati koji simuliraju pad memorije i proveravaju memorijske nedostatke. Praksa obuhvata generisanje i razvoj metoda i tehnika za upotrebu različitih platformi i jezičkih okruženja.

Testiranje softvera

Inkrementalne prakse obuhvataju:

***7. Statistic**ko testiranje(Tucson)* –** Koncept statističkog testiranja obuhvata ideju da se, umesto metoda za debagovanje, koristi testiranje kako bi se postigla pouzdanost softvera. Postoje mnogi argumenti koji govore u prilog ovoj metodi. Statisticko testiranje odnosi se na razradivanje softvera u operativnom smislu i potom merenje meduotkaza koji se koriste da procene pouzdanost. Dobar proces razvoja treba da doprinosi povećanju srednjeg vremena medu otkazima svaki put kada se ispravi bag. Nakon toga ovaj proces postaje release kriterijum i izvor uslova za zaustavljanje testiranja.

***8. Poluformalne metode* –** Formalne metode u softverskom inženjeringu traju nekoliko decenija. Ključni koncept formalne metode je posezanje za verifikacijom programa umesto testiranja i debagovanja. Vizija formalnih metoda je oduvek bila takva da, ako je specifikacija softvera dobra, ona može dovesti do automatskog generisanja koda, koji zahteva minimum testiranja. U praksi se raspravljalo o opsegu semiformalnih metoda koji industrija ignoriše. Semiformalna metoda je takva metoda u kojoj specifikacija može biti u obliku dijagrama prelaza stanja ili tabela koji se mogu koristiti i za generisanje testova.

***9. Testovi provere koda* –** Ova ideja odnosi se na spregu automatskog test programa (obično regresionog testa) sa kontrolnim sistemom promena. Microsoft je koristio ovakve sisteme. Na ovaj način se omogućava pokretanje automatskog testa na tek izmenjenom kodu, tako da su šanse za neuspelost builda zbog koda svedene na minimum. Kod Microsofta se neguje praksa da dok god kod ne prode test, ne prolazi ni sledeći build.

Testiranje softvera

Inkrementalne prakse obuhvataju:

***7. Statistic**ko testiranje(Tucson)* –** Koncept statističkog testiranja obuhvata ideju da se, umesto metoda za debagovanje, koristi testiranje kako bi se postigla pouzdanost softvera. Postoje mnogi argumenti koji govore u prilog ovoj metodi. Statisticko testiranje odnosi se na razradivanje softvera u operativnom smislu i potom merenje meduotkaza koji se koriste da procene pouzdanost. Dobar proces razvoja treba da doprinosi povećanju srednjeg vremena medu otkazima svaki put kada se ispravi bag. Nakon toga ovaj proces postaje release kriterijum i izvor uslova za zaustavljanje testiranja.

***8. Poluformalne metode* –** Formalne metode u softverskom inženjeringu traju nekoliko decenija. Ključni koncept formalne metode je posezanje za verifikacijom programa umesto testiranja i debagovanja. Vizija formalnih metoda je oduvek bila takva da, ako je specifikacija softvera dobra, ona može dovesti do automatskog generisanja koda, koji zahteva minimum testiranja. U praksi se raspravljalo o opsegu semiformalnih metoda koji industrija ignoriše. Semiformalna metoda je takva metoda u kojoj specifikacija može biti u obliku dijagrama prelaza stanja ili tabela koji se mogu koristiti i za generisanje testova.

***9. Testovi provere koda* –** Ova ideja odnosi se na spregu automatskog test programa (obično regresionog testa) sa kontrolnim sistemom promena. Microsoft je koristio ovakve sisteme. Na ovaj način se omogućava pokretanje automatskog testa na tek izmenjenom kodu, tako da su šanse za neuspelost builda zbog koda svedene na minimum. Kod Microsofta se neguje praksa da dok god kod ne prode test, ne prolazi ni sledeći build.

Testiranje softvera

Inkrementalne prakse obuhvataju:

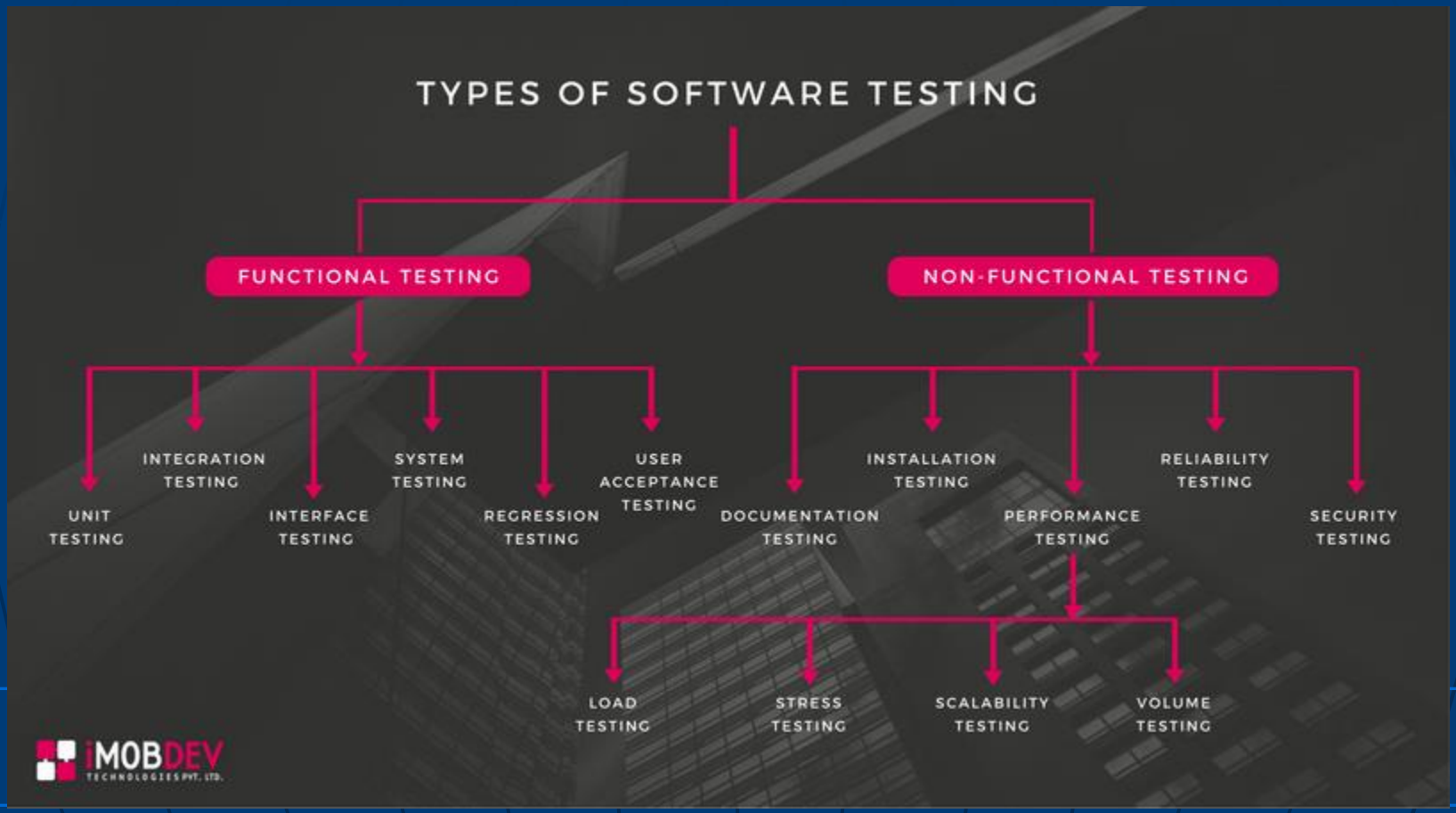
10. ***Minimiziranje regresije test case-ova –*** U organizacijama koje imaju istoriju razvoja i proizvode koji sazrevaju tokom mnogih release etapa, nije neobično uočiti skup regresionih testova koji je jako velik. Negativne posledice ovolikog skupa testova su predugo trajanje naspram male dodatne vrednosti. Postoji nekoliko metoda koje umanjuju regresione testove. Jedna od njih posmatra pokriće proizvedenog koda i svodi test caseo-ve na minimalan broj. Ovakva metoda, mada atraktivna, meša strukturnu metriku sa funkcionalnim testom.

11. ***Implementirane verzije za MTTF1-*** Prednosti koju beta programi omogućavaju su da pojedinac dobija veliki uzorak korisnika za test proizvoda. Ako je proizvod napravljen tako da se otkazi snimaju i vraćaju prodavcu, on će obezbediti odličan izvor merenja srednjeg vremena medju otkazima softvera. Postoji nekoliko upotreba ove metrike. Prvo, može se koristiti kao mera kvaliteta softvera u pogledu koji je kupcu značajan. Drugo, omogućava merenje srednjeg vremena medu otkazima istog proizvoda pod različitim profilima korisnika ili grupe korisnika. Treće, može biti podrška za uočavanje prvog otkaza podataka koji može doprineti dijagnozi i rešavanju problema. Microsoft je potvrdio da sprovodi najmanje prva dva principa u svojim beta verzijama.

12. Benchmark trendovi – Benchmarking je širok koncept koji se primenjuje na mnoge discipline u različitim oblastima. U svetu softverskog testiranja može se interpretirati kroz tehnike i performanse metoda testiranja koje su iskusili drugi developeri softvera.

13. Bug prednosti- Prednosti bagova se odnose na inicijative koje menjaju organizaciju u smislu fokusa na otkrivanje softverskih grešaka. Ponekad ukljucuju i nagrade. Iskustvo potvrduje da ovakav napor teži da identifikuje veci od uobicajenog broja bagova. Potrebni su i dodatni resursi za ispravku. Konacan rezultat je veci kvalitet proizvoda.

Testiranje softvera



Testiranje softvera

Selenium

Ranorex

testim

watir

QuickTest Professional

headspin

Parasoft Selenic

froglogic

HVALA NA PAŽNJI