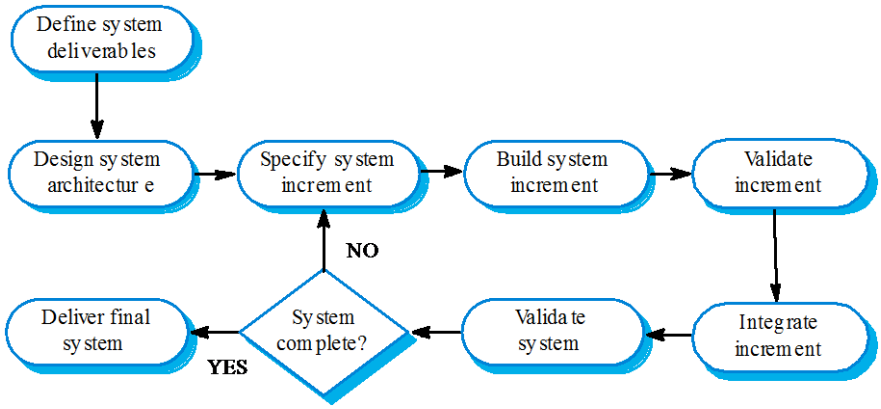
Ubrzani Razvoj Softvera *(Rapid software development)*

Procesi ubrzanog razvoja softvera su smišljeni u svrhu brzog razvoja smislenog (korisnog) softvera. Općenito su to iterativni procesi kod kojeg se faze specifikacije, dizajna, razvoja i testiranja isprepliču. Iako postoji niz različitih pristupa ubrzanom razvoju softvera, svi dijele iste osnovne karakteristike:

1. Proces specifikacije, dizajna i implementacije se odvijaju istovremeno. Ne radi se detaljna specifikacija sustava a dokumentacija dizajna je minimizirana. Korisnički zahtjevi uključuju samo najvažnije karakteristike sustava.
2. Sustav se razvija kao serija inkremenata. Krajnji korisnici i poslodavci su uključeni u specificiranje i razvijanje svakog inkrementa kako bi mogli odmah zatraziti izmjene u softveru koje bi se trebale u sljedećem inkrementu nadodati.
3. Korisnička sučelja se često razvijaju kroz interaktivan razvoj koji omogućava da se sučelje razvije ubrzano crtajući i razmještajući razna dugmad.

Glavne prednosti inkrementalnog pristupa razvoju softvera su:

1. ***Ubrzana isporuka korisničkih usluga*** Brza isporuka inkremenata može pružiti najosnovnije funkcionalnosti sustava korisniku, pa na taj način korisnici brzo dobiju koristi od razvoja softvera.
2. ***Anganžman korisnika u razvoju sustava*** Korisnici sustava se trebaju uvest u razvoj inkremenata jer mogu pružiti odaziv na sustav razvojnom timu.



Proces iterativnog razvoja

Glavne poteškoće iterativnog razvoja i inkrementalne isporuke su:

1. ***Problemi menadžmenta*** Kod inkrementalnog razvoja se izmjene dešavaju često te nije isplativo raditi puno dokumentacije. Daljnje, inkrementalni razvoj može zahtjevati korištenje nepoznatih tehnologija kako bi se postigao ubrzani razvoj.
2. ***Problemi ugovora*** Kad ne postoji jasna specifikacija može biti teško sastaviti ugovor za razvoj softvera.

3. ***Problemi validacije*** Radi manjka dokumentacije i stalnog mjenjanja zahtjeva, tim za validaciju i verifikaciju ne može unaprijed planirati testiranje sustava.
4. ***Problemi održavanja*** Stalna izmjena narušava strukturu sustava što znači da ljudi koji nisu bili članovi razvojnog tima teško razumiju način na koji softver fuknkcionira.

Ciljevi inkrementalnog i prototipnog razvoja:

1. Cilj inkrementalnog razvoja je isporuka funkcionirajućeg sustava krajnjem korisniku. To znači da se počinje od korisničkih zahtjeva koji su najjasniji i koji imaju najveći prioritet. Zahtjevi sa najmanjim prioritetom se implementiraju kada i ako ih korisnik zatraži.
2. Cilj „throw-away" prototipnog razvoj je potvrđivanje i izvođenje korisničkih zahtjeva (bolje shvačanje. Treba početi sa zahtjevima koji nisu jasni jer je potrebno da se razjasne. Zahtjevi koji su sasvim jasni nemoraju se nikada prototipirat.

Agilne metode *(Agile methods)*

Agilne metode se općenito oslanjaju na iterativni pristup specifikacije, razvoja i isporuke softvera a prvenstveno su razvijene za razvoj poslovnih aplikacija kod kojih se zahtjevi često mijenjaju tijekom samog procesa. Cilj im je klijentu isporučiti softver što brže koji onda mogu zatražiti modifikacije na zahtjevima koje će se integrirati u sljedećim iteracijama sustava. Najpoznatija agilna metoda je ekstremno programiranje (extreme programming) ali postoje i druge: Scrum, Crystal, Adaptive Software Development, DSDM i Feature Driven Development. Glavni i svima zajednički principi ovih metoda su:

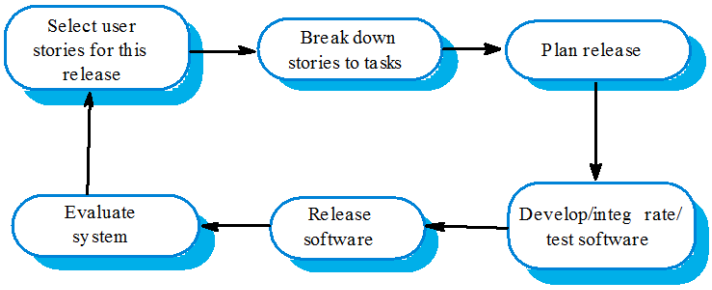
- ***Uključivanje klijenata*** Klijenti se trebaju uključiti u sam proces razvoja.
- ***Inkrementalna isporuka*** Softver se razvija u inkrementima kod kojih klijent specificira zahtjeve svakog inkrementa.
- ***Ljudi, ne proces*** Trebaju se prepoznati vještine članova tima te prepustit njima da razviju svoj način rada bez da ih se ograničava da rade po pravilima procesa.
- ***Prihvati promjene*** Za očekivat je da će se zahtjevi mijenjat, zato treba sustav dizajnirati tako da se promjene mogu implementirati.
- ***Jednostavnos održavanja*** Treba se fokusirati na jednostavnost kako kog razvoja softvera tako i kod samog procesa razvoja.

Agilne metode su najprikladnije za razvoj malih i srednje velikih poslovnih sustava i aplikacija. Nisu prikladne za razvoj velikih ili kritičnih sustava gdje se traže detaljne analize svih zahtjeva kako bi se shvatio njihov aspekt sigurnosti i zaštite.

Ekstremno programiranje *(Extreme programming)*

Kod ekstremnog programiranja, svi zahtjevi se izražavaju pomoću scenarija koji implementiraju serije zadataka. Programeri rade u parovima i razvijaju testove za svaki zadatak prije nego počnu pisati kod. Svi testovi moraju biti uspješno izvršeni kada se novi dio koda integrira u sustav. Načelno sadrži sve principe po kojima se ravnaju agilne metode. Kod ovakvog pristupa,

klijent je dio tima koji zajedno sa ostalim članovima razvija scenarije čime se na kraju dobije „story card“ koji obuhvaća sve potrebe klijenta.



Ciklus ekstremnog programiranja

Testiranje kod XP-a *(Testing in extreme programming)*

Da bi se izbjegli neki problemi testiranja i validacije sustava, XP stavlja najveći naglasak na proces testiranja za razliku od drugih agilnih metoda. Ključni elementi testiranja kod XP-a su:

1. Razvoj „prvo pisanje testa“ (Test-first development)
2. Razvoj inkrementalnog testiranja na osnovu scenarija.
3. Uključivanje korisnika u razvoj i validaciju testova.
4. Korištenje automatiziranih testiranja.

Prvo pisanje testa znači da se testovi napišu prije samog kod programa i to je jedna od najjačih karakteristika XP-a. Nakon što se dio koda integrira u sustav, testiranje je odmah moguće a i lako izvedivo korištenjem sustava automatiziranog testiranja koji te testove stavlja na testiranje. Problem može biti to što testovi ne predvide sve kritične odsječke sustava pa tako oni ostanu netestirani. Nekada je teško napisati dobar test radi kompleksnosti takvog problema.

Programiranje u parovima *(Pair programming)*

Inovativni postupak u razvoju softvera je da programeri rade u parovima. Oni stvarno sjede za istim računalom i razvijaju softver. Ovakav pristup ima niz prednosti:

1. Podupire ideju javnog učestvovanja i odgovornosti za sustav. Tim ima kolektivnu odgovornost te za greške ne ogovaraju individualci.
2. Djeluje kao vrsta neformalnog procesa provjere jer kod gledaju barem dvije osobe. Time se uspješno otkrivaju greške u kodu a ujedno je mnog „jeftiniji“ nego fromalni proces provjere.
3. Pomaže podržavanju refaktorizacije, koja je proces usavršavanja softvera. Softver bi trebao biti konstantno refaktoriziran, tj. djelovi koda bi se trebali stalno nanovo pisati kako bi poboljšali čitljivost cijele strukture.

Ubrzani razvoj aplikacija *(Rapid application development)*

Ubrzani razvoj aplikacija je tehnika koja se korisit za razvoj aplikacija koje rade sa puno podataka (data-intensive). Organizirane su kao set alata koji omogućavaju da se podaci kreiraju, pretražuju, prikazuju i prezentiraju u obliku formi.
Alati koji su uključeni u radno okružje URA-a su:

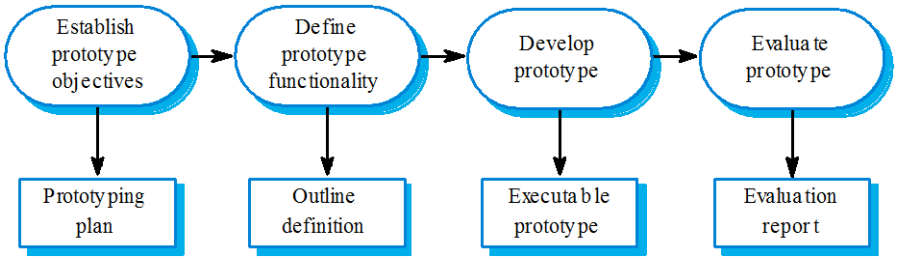
1. *Jezik za programiranje baze podataka* koji ugrađuje znanje o strukturi baze podatak i uključuje osnovne operacije za manipulaciju baze podataka.
2. *Generator sučelja* koji se koristi za kreiranje formi za unos i prikaz podataka.
3. *Poveznik na office aplikacije* kao što us proračunske tablice za analizu i manipulaciju numeričkih informacija ili leksički procesor za kreiranje predložaka izvještaja.
4. *Generator izvještaja* koji se koristi za kreiranje izvještaja podataka iz baze.

Vizualni razvoj je pristup URA-a koji se oslanja na integriranje sitnih ponovno iskoristivih (reusable) komponenti. Alternativa pristupu baziranom na ponovno iskoristivosti je ponovna iskoristivost komponenti koje su cjelovite sistemske aplikacije. To se naziva COTS (Comercial off-the-shelf) što znači da su te aplikacije već dostupne.
Glavna prednost ovakvog pristupa je da se mnoge funkcionalnosti aplikacija mogu implementirati brzo uz malu cijenu. No mana mogu biti performanse jer je potrebno da se sustav prebacuje sa jedne aplikacije na drugu što dodatno može otežati korištenje ako su korisniku nepoznata sučelja različitih aplikacija.

Prototipiranje softvera *(Software prototyping)*

Prototip je prvobitna verzija softverskog sustava koja se koristi za demonstraciju koncepta, isprobavanja raznih dizajnova, u biti za otkrivanje mogućih problema i njihovog rješenja. Brz razvoj prototipa je bitan kako bi se izdaci držali pod kontrolom i kako bi poslodavci mogli eksperementirati sa sustavom u ranoj fazi razvoja. Prototipiranje softvera može biti korisno u razvoju softvera na više načina:

1. U procesu projektiranja zahtjeva, prototip može pomoći u otkrivanju i validiranju sistemskih zahtjeva.
2. U procesu dizajniranja sustava, prototip se može koristiti za istraživanje mogućih rješenja softvera i za podršku dizajniranja korisničkog sučelja.
3. U procesu testiranja, prototip se može koristiti za usporedno testiranje sa sustavom koji će se isporučiti klijentu.



Proces prototipnog razvoja

Koristi koje se dobiju prototipom sustava su:

1. Poboljšana lakoća korištenja sustava.
2. Veća sličnost sustava onom sustavu kojeg korisnici žele.
3. Poboljšane kvalitete dizajna
4. Poboljšana održivost
5. Razvoj s manje truda

Glavni problem sa razvojem izvršnog „throw-away" prototipa je da način korištenja prototipa možda neće odgovarat načinu na koji će se koristiti finalna verzija sustava. Testeri prototipa možda nisu tipični korisnici.

Verifikacija i Validacija

Verifikacija i validacija je proces provjere i analize koji počinje sa preispitivanjem zahtjeva sve do preispitivanja dizajna, inspekcije koda i testiranja produkta. V & V se razlikuju po sljedećim pitanjima:

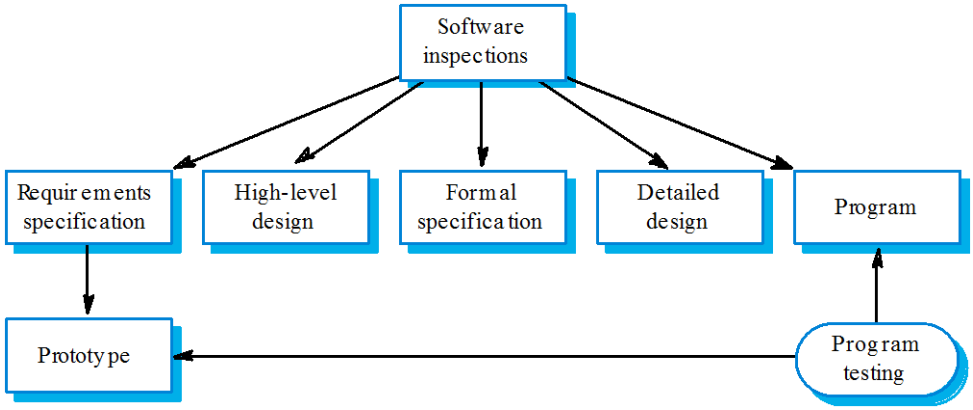
- **Validacija:** Da li radimo ispravan produkt?
- **Verifikacija:** Da li produkt radi ispravno?

Znači verifikacija kontrolira da li sustav ispunjava funkcionalne i nefunkcionalne zahtjeve a validacija se više bavi pitanjem „da li softver ispunjava očekivanja klijenta?". Glavni cilj ovog procesa je da se uspostavi pouzdanost da je softver prikladan za svoju namjenu, a mjera pouzdanosti ovisi o sljedećim kriterijama:

1. ***Funkcije softvera*** Koliko je softver kritičan za neku organizaciju.
2. ***Očekivanja korisnika*** Koji je odnos očekivanja korisnika u odnosu na ono što su dobili sa softverom.
3. ***Markentiški okoliš*** Kakva je konkurencija, koliko su klijenti spremni platiti i do kojeg termina sustav mora izaći na tržište?

Unutar V & V procesa, postoje dva komplementarna (nadopunjavajuća) pristupa provjere i analize sustava.

1. ***Inspekcija softvera*** analizira i provjerava ono što reprezentira sustav kao što je dokumentacija zahtjeva, dijagrami dizajna i „source code" programa.
2. ***Testiranje softvera*** uključuje testiranje neke implementacije softvera sa testnim podacima.



3.
4. *Statička i dinamička verifikacija i validacija*

Statičke tehnike inspekcije softvera mogu samo provjeriti odnos izmedju programa i njegove specifikacije. One ne mogu pokazat da je softver funkcionalan te se tim tehnikama ne mogu provjeriti izranjajuća svojstva kao što su performanse ili pouzdanost.
Testiranje uključuje pokretanje programa koristeći podatke kao što bi bili u stvarnosti, te promatranjem izlaznih podataka se otkrivaju anomalije i defekti. Postoje dva različita tipa testiranja:

1. ***Validacijsko testiranje*** je namjenjeno da pokaže da je softver ono što korisnik želi, tj. da ispunjava svoje zahtjeve.
2. ***Testiranje na defekte*** je prije namjenjeno da otkrije defekte u sustavu nego da simulira operacijsku uporabu. Cilj je pronaći proturječnosti između programa i njegove specifikacije.

Testiranje (općenito V & V) i **debagiranje** imaju različite ciljeve:

1. Proces V & V je namjenjen da pokaže postojanje defekata u sustavu softvera.
2. Debagiranje je proces pronalaska i uklanjanja tih defekata.

Debaging alati su ustvari set alata za jezičnu podršku koji su integrirani u kompilacijski sustav. Oni pružaju „run-time“ okružje za program.

Planiranje verifikacije i validacije (Planing verification and validation)

Verifikacija i validacija je skup proces te je nužno pomno planirati kako bi se dobila što veća korist inspekcija i testiranja, i kako bi se izdaci držali pod kontrolom.
Dio V & V procesa planiranja je da se donesu odluke o omjeru statičkog i dinamičkog pristupa verifikacije i validacije,da se odrede standardi i procedure za inspekciju i testiranje softvera, da se uspostave checklist-e koje će vodit inspekciju programa, te da se definira plan testiranja softvera.
Planiranje testova se bavi uspostavom standarda za proces testiranja. Menadžerima pomažu da rezerviraju potrebna sredstva i proračunaju trajanje testiranja, a inženjerima da dizajniraju i izvrše testove na sustavu.
Plan testa nije statički dokument već se razvija i mijenja tijekom razvojnog procesa zbog kašnjenja drugih faza. Trebao bi sadržati opis svih predmeta koji će se testirati, raspored

testiranja, procedure za upravljanje testnih procesa, hardverske i softverske zahtjeve, i sve probleme koji će se vjerojatno pojaviti tjekom testiranja.

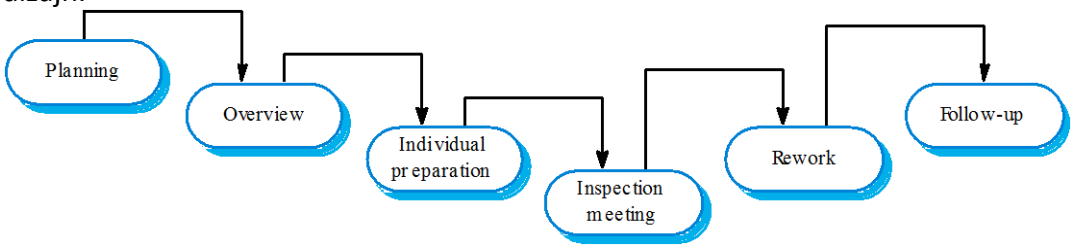
Inspekcija softvera (Software inspections)

Softverska inspekcija je statički proces verifikacije i validacije u kojem se sustav prekontrolirava kako bi se pronašle pogreške, propusti i anomalije. Općenito se fokusira na inspekciju source code-a ali se i svaka druga čitljiva reprezentacija softvera može provjeriti.
Tri glavne prednosti inspekcije spram testiranja su:

1. Tjekom testiranja. Jedna greška može prikriti druge greške te nikada nije sigurno da li postoji više grešaka ili su one nuspojava samo jedne greške. Pošto je inspekcija statički proces, ne postoje interakcije grešaka.
2. Nepotpune verzije sustava ne iziskivaju dodatne troškove pri inspekciji dok se za testiranje takvih sustava moraju razviti posebni testovi koji bi testirali samo djelove sustava što u konačnici povećava troškove.
3. Pored otkrivanja grešaka, inspekcija može procjeniti atribute kvalitete kao što su suglasnost sa standardima, prenosivost i održivost. Mogu se vidjeti neefikasnosti, neprikladni algoritmi i loši stilovi programiranja koji mogu biti teški za održavanje i aktualiziranje.

Proces inspekcije programa (The program inspection process)

Programske inspekcije su provjere čiji je cilj otkrivanja defekata programa. Glavna razlika ove od ostalih inspekcija je što se ona isključivo bavi pronalaskom defekata a ne pitanjima vezanim za dizajn.



Proces inspekcije softvera

Automatizirana statička analiza (Automated static analysis)

Namjena automatizirane statičke analize je da privuče pozornost inspektora na neke anomalije u programu kao što su varijable koje se koriste bez inijalizacije, varijable koje se ne koriste ili čije vrijednosti premašuju opseg dopuštenih vrijednosti.
Faze vezane za statičku analizu uključuju:

1. ***Analiza kontrole toka*** Ova faza isteče petlje koje imaju višestruke izlaze ili ulaze te nedostižne djelove koda.
2. ***Analiza korištenja podataka*** Ova faza ističe kako se varijable u programu koriste.
3. ***Analiza sučelja*** Ova analiza provjerava dosljednost rutine i proceduralne dekleracije te njihovo korištenje.

4. ***Analiza toka informacija*** Ova faza identificira ovisnost između ulaznig i izlaznih podataka.
5. ***Analiza puta*** Ova faza semantičke analize identificira sve moguće puteve kroz koje program može proći i izlaže sve naredbe izvršene na tim putevima.

Verfikacija i formalne metode (Verification and formal methods)

Formalne metode softverskog razvoja su bazirane na matematičkim reprezentacijama softvera. Te formalne metode se uglavnom bave matematičkom analizom specifikacije, tj. sa transformiranjem specifikacije u detaljniju, semantički ekvivalentnu, reprezentaciju ili sa formalnim verificiranjem da je neka reprezentacija sustava semantički ekvivalentna nekom drugom obliku reprezentacije.**(OMG!!!)**

Formalne metode se mogu koristiti u razne svrhe V & V procesa:

1. Formalna specifikacija se može razviti i matematički analizirati na proturječnosti.
2. Korištenjem matematičkih argumenata, formalno se može potvrditi da sustav odgovara svojoj specifikaciji.

Iako je nedvojbeno da će formalna specifikacija sustava sadržavati manje grešaka koje treba ispravljati, ona ne garantira da će takav sustav biti pouzdan u praksi. Razlozi za to su:

1. ***Specifikacija možda ne pretstavlja stvarne korisničke zahtjeve na sustav.***
2. ***Matematički dokaz može sadržati greške.***
3. ***Matematički dokaz možda pretpostavlja način korišteja sustava koji nije točan.***

„Cleanroom“ razvoj softver (Cleanroom software development)

Cilj ovakvog pristupa razvoju softvera je softver sa nula defekata (zero-defect). „Cleanroom“ pristup razvoju softvera je baziran na sljdećih pet strategija:

1. ***Formalna specifikacija*** Softver koji se razvija se formalno specificira.
2. ***Inkrementalni razvoj*** Softver se razlaže na inkremente koji se zasebno razvijaju i validiraju.
3. ***Strukturizirano programiranje*** Koristi se samo ograničen broj kontrolnih i podatkovnih apstrakcijskih konstrukcija. (whatever)
4. ***Statička verifikacija*** Razvijeni softver se statički verificira koristeći snažne metode statičke inspekcije.
5. ***Statično testiranje sustava*** Integrirani inkrementi softvera se statički testiraju.

U ovaj proces su uključena tri tima:

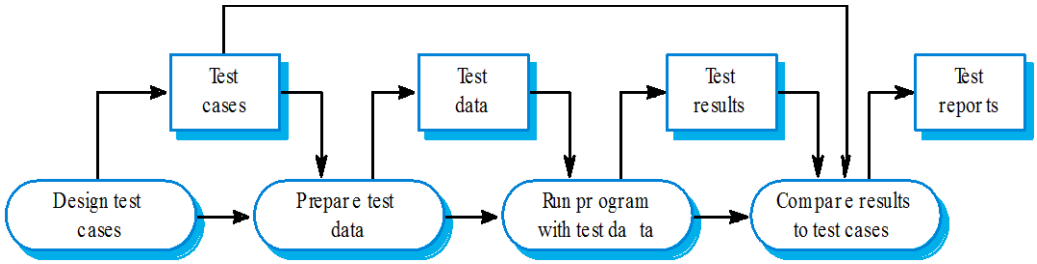
1. ***Specifikacijski tim*** Ovaj tim je odgovoran za razvoj i održavanje specifikacije sustava.
2. ***Razvojni tim*** Ovaj tim ima odgovornost za razvoj i verifikaciju softvera.
3. ***Certifikacijski tim*** Ovaj tim je odgovoran za razvoj statičkih testova koji će testirati softver nakon što se on razvije.

Testiranje softvera (Software testing)

Proces testiranja softvera ima dva cilja:

1. Da pokaže developeru i korisniku da softver zadovoljava svoje zahtjeve. Za korisnički softver, ovo znači da bi trebao postojati barem jedan test za svaki zahtjev iz dokumenta korisničke i sistemske specifikacije zahtjeva. Za generički softver, ovo znači da bi trebali postojati testovi za sve funkcije sustava.
2. Da bi se otkrile greške ili mane kod softvera čije je ponašanje netočno, neželjeno ili ne podliježe svojoj specifikaciji.

Prvi cilj vodi k validacijskom testiranju, a drugi k testiranju na greške (defect testing).
Testiranjem se ne može pokazati da sustav nema grešaka ili da će se ponašati po specifikaciji u svim okolnostima. Testiranjem se može pokazati samo prisutnost grešaka, ne njihova odsutnost.



Slika 3.1 – model procesa testiranja softvera.

Test case-ovi su specifikacije ulaza za test i očekivanih izlaza iz sustava, uključujući opis onoga što se testira. Test data su ulazi koji se upotrebljavaju za testiranje sustava.
Kod većine sustava, programeri testiraju komponente sustava koje su sami razvili, i onda ih predaju integracijskom timu koji integrira podsustave u kompletan sustav i zatim ga testira.

Testiranje sustava

Testiranje sustava uključuje integraciju dviju ili više komponenti koje implementiraju funkcije sustava i testiranje tako integriranog sustava. Kod iterativnog razvoja, sustav se testira na način da se testira inkrement koji se daje korisniku na korištenje, a kod waterfall procesa, sustav se testira kao cjelina.

Za najkompleksnije sustave, razlikujemo dvije faze kod testiranja sustava:

1. ***Integracijsko testiranje*** – gdje testni tim ima pristup source kodu sustava, te pokušava pronaći problem i indetificirati komponente koje se trebaju ispraviti.
2. ***Izdavačko testiranje (Release testing)*** – gdje se testira verzija sustava koja bi se mogla predati korisnicima. Testni tim pokušava potvrditi da sustav zadovoljava svoje zahtjeve i da je pouzdan.

Integracijsko testiranje

Proces integracije sustava uključuje izgradnju sustava od njegovih komponenti i testiranje rezultirajućeg sustava zbog problema koji bi mogli nastati zbog interakcije između tih komponenti.
***Top-down** integracija* označava proces kod kojeg se prvo razvije kostur čitavog sustava kojemu se kasnije dodaju komponente.
***Bottom-up** integracija* označava proces kod kojeg se prvo integriraju infrastrukturne komponente koje pružaju neke osnovne usluge (network, pristup bazi podataka), i onda se dodaju funkcijske komponente.

Veliki problem kod integracijskog testiranja je lociranje grešaka, te potreba za sastavljanjem minimalnog sustava (kojeg isto trebe testirati) kako bi se uopće mogla testirati interakcija dodavanjem novih komponenti. Nadalje, neke komponente iziskivaju niz drugih komponenti koje moraju biti integrirane kako bi se one mogle testirati.

Dakle, nakon integracije svakog novog inkrementa, važno je ponoviti testove za sve prethodne inkremente, kao i obaviti nove testove koji bi pokazali da li novi sustav radi kako treba. Ponovo obavljanje niza testova naziva se ***regresijsko testiranje***.

Izdavačko testiranje (Release testing)

Izdavačko testiranje je testiranje verzije sustava koja će biti distribuirana korisnicima. Primarni cilj ovog procesa je da pokaže da sustav ispunjava svoje zahtjeve. Mora se pokazati da sustav ima specificirane funkcionalnosti, performanse i pouzdanost, i da ne dolazi do grešaka tijekom normalnog rada. Ako ovo testiranje uspješno prođe, sustav se može dostaviti korisniku.
Izdavačko testiranje je obično proces black-box testiranja gdje su testovi rade na temelju sistemske specifikacije. Sustav se tretira kao crna kutija čije se ponašanje može odrediti jedino proučavanjem ulaza i izlaza iz sustava. Drugo ime za ovaj proces je testiranje funkcionalnosti, jer se tester brine samo o funkcionalnosti, a ne o implementaciji sustava.
Kod ovakvog testiranja treba se pokušat srušit sustav biranjem kritičnih djelova sustava kod kojih je vjerojatnost dolaska do pogreške najveća.

Testiranje performansi

Testovi performansi moraju biti dizajnirani na način da provjere da li sustav ispravno radi do nekog maksimalnog opterećenja. Kod ovakvih testova opterećenje se polako povećava sve dok performanse sustava ne postanu neprihvatljive. Kod testiranja performansi pokušava se pokazati da sustav zadovoljava svoje zahtjeve, ali se istodobno pokušavaju pronaći greške i mane u sustavu.
Iskustvo je pokazalo da je najbolji način za otkrivanje mana u sustavu dizajniranje testova oko samih granica sustava. Kod testiranja performansi, ovo znači da bi trebali odabirati zahtjeve koji su izvan granica softvera, odnosno trebali bi 'sustav staviti pod stres' (odatle ime *stress testing*).

Ovakav tip testiranja ima dvije funkcije:

1. Testira način na koje se sustav ponaša tijekom rušenja. Važno je da pad sustava ne izazove kvar podataka ili neželjeni gubitak korisničkih usluga. Stress testing provjerava da preopterećenjem sustava sustav 'meko pada' (fail-soft), a ne da se samo sruši i izazive gubitak ili kvar informacija.
2. Sustav se stavlja 'pod stres', te se na ovaj način mogu otkriti greške koje se inače ne bi mogle otkriti.

Stress testing je posebno važan kod distribuiranih sustava. Ovi sustavi pokazuju jako veliku degradaciju kada su preopterećeni.

Testiranje komponenti

Ovo je proces testiranja na greške, tako da mu je cilj pronaći greške u komponentama. Za testiranje komponenti zaduženi su programeri koji su ih razvili.
Postoji više tipova komponenti koje se mogu testirati u ovoj fazi:

1. Individualne funkcije ili metode unutar objekta
2. Objektne klase koje imaju više atributa i metoda
3. Složene komponente (composite components) koje su sastavljene od više različitih objekata ili funkcija i koje imaju definirano sučelje preko kojeg pristupamo njihovim funkcionalnostima.

Testiranje sučelja

Funkcionalnostima složenih komponenti pristupamo putem njihovog sučelja. Kod testiranja sučelja testira se da li se sučelja tih složenih komponenti ponašaju u skladu sa specifikacijom. Testiranje sučelja je posebno važno za objektno-orijentirani component-based razvoj.
Greške u sučelju se kod složenih komponenti ne mogu otkriti testiranjem individualnih objekata ili komponenti, jer takve greške mogu nastati zbog interakcije različitih dijelova od kojih su te komponente sastavljene.

Postoje različiti tipovi sučelja između programskih komponenti:

1. ***Parametarska sučelja*** – ovo su sučelja gdje se podaci ili funkcijske reference prenose od jedne do druge komponente.
2. ***Sučelje dijeljene memorije*** – ovo su sučelja kod kojih komponente dijele blok memorije. Jedan podsustav stavlja podatke u memoriju, a drugi ih koristi.
3. ***Proceduralna sučelja*** – ovo su sučelja gdje jedna komponenta obuhvaća skup procedura koje mogu pozvati druge komponente.
4. ***Sučelja izmjene poruka*** – ovo su sučelja gdje jedna komponenta zahtijeva uslugu od druge komponente tako da joj pošalje poruku.

Tipovi grešaka kod sučelja se mogu podijeliti na sljedeće tri klase:

1. ***Pogrešna upotreba sučelja*** – Pozivajuća komponenta zove neku drugu komponentu i napravi grešku u njezinom sučelju.
2. ***Pogrešno razumijvanje sučelja*** – Pozivajuća komponenta pogrešno shvati specifikaciju pozvane komponente i pretpostavi kako će se ona ponašati. Pozvana komponenta se ne ponaša na taj način, što dovodi do neočekivanog ponašanja pozivajuće komponente.
3. ***Vremenske greške*** – Ove se greške događaju kod sustava koji rade u stvarnom vremenu i koji koriste sučelje dijeljene memorije ili sučelje izmjene poruka. Proizvođač podataka i potrošač podataka mogu raditi različitim brzinama, što može dovesti do problema.

Testiranje sučelja na greške može biti teško jer se određene greške manifestiraju samo pod određenim uvjetima.

Test case dizajn

Test case dizajn je dio testiranja sustava i komponenti u kojem se dizajniraju test case-ovi (ulazi i predviđeni izlazi) da bi se testirao sustav. Cilj ovog procesa je kreirati skup test case-ova koji su efektivni u pronalaženju greški u programu i u pokazivanju toga da sustav zadovoljava svoje zahtjeve.
Pristupi test case dizajnu:

1. ***Testiranje bazirano na zahtjevima*** – gdje su test case-ovi dizajnirani na način da provjere sistemske zahtjeve.
2. ***Testiranje particija*** – gdje se identificiraju ulazne i izlazne particije i dizajniraju testovi tako da sustav izvrši ulaze sa svih particija i generira izlaze u sve particije. Particije su skupovi podataka koji imaju neke zajedničke karakteristike.
3. ***Strukturalno testiranje*** – gdje pri dizajniranju testova upotrebljavamo znanje o strukturi programa.

Testiranje bazirano na zahtjevima

Testiranje bazirano na zahtjevima je sistematski pristup test case dizajnu gdje se proučava svaki zahtjev i za njega se kreira skup testova. Ovo testiranje je više orijentirano na validaciju sustava, a ne na traženje greški – pokušava se demonstrirati da su zahtjevi ispravno implementirani unutar sustava.

Testiranje particija

Ulazni i izlazni podaci programa obično spadaju u više različitih klasa koje imaju neke zajedničke karakteristike. Programi se obično ponašaju na sličan način kao svi članovi neke klase. Na primjer, ako testiramo program koji obavlja neko računanje koje uključuje dva pozitivna broja, očekivati ćemo da se program ponaša na isti način za sve pozitivne brojeve.

Zbog ovog ekvivalentnog ponašanja, ovakve klase se ponekad nazivaju *ekvivalentne particije ili domene*. Da bi dizajnirali test case-ove, moramo identificirati sve particije sustava ili komponenti. Particije se odabiru korištenjem programske specifikacije ili korisničke dokumentacije, na temelju iskustva, gdje se predvide klase ulaznih vrijednosti za koje bi se mogla generirati greška. Nakon tog koraka, odabiru se test case-ovi iz svake particije. Kod test case-ova bi trebali upotrebljavati neke atipične vrijednosti (npr. sa granice particije), zato što se na greške češće nailazi korištenjem takvih vrijednosti (jer programeri i dizajneri uglavnom pri testiranju koriste samo tipične vrijednosti).

Strukturalno testiranje

Strukturalno testiranje je pristup test case dizajnu kod kojeg se pri izradi testova upotrebljava znanje o softverskoj strukturi i implementaciji. Ovakav se pristup ponekad naziva white-box, glass-box testiranje ili clear-box testiranje.
Ako se razumije algoritam koji se je upotrijebio kod komponenti, to nam može olakšati identificiranje particija i test case-ova.

Testiranje puta (Path testing)

Glavni cilj testiranja puta je da osigura da je skup test case-ova dizajniran na takav način da se svaki nezavisni put unutar programa izvrši barem jednom.
Početna točka testiranja puta je graf toka programa. Ovo je pojednostavljeni prikaz svih puteva unutar programa. Čvorovi unutar grafa predstavljaju odluke, a strelice tok kontrole.

Automatizacija testiranja

Testiranje je skupa i naporna faza softverskog procesa, i zato su se razvili alati za testiranje pomoću kojih se cijena testiranje znatno smanjuje. Neki od takvih alata mogu uključivati: test menadžer (upravlja radom testnih programa), test data generator, oracle (generira predviđanja i očekivane rezultate testa), uspoređivač fajlova, dinamički analizator, simulator (npr. korisničkog sučelja).

Evolucija softvera (Software evolution)

Nakon što se sustav distribuira korisnicima i počne koristiti, on se uvijek mijenja da bi mogao ostati koristan. Otkriju se novi zahtjevi, a stari se zahtjevi promijene. Dijelovi softvera se moraju mijenjati da bi se popravile greške u radu sustava, sustav se mora prilagoditi novim platformama, moraju se poboljšati njegove ne-funkcionalne karakteristike i performanse. Razvoj softvera ne prestaje u trenutku kada se sustav dostavi korisnicima, već se nastavlja sve dok se sustav koristi.

Evolucija softvera je važna zato što razne organizacije danas potpuno ovise o svojim softverskim sustavima i u njih ulažu milijune dolara. Otprilike 90% cijene softvera otpada na njegovu evoluciju.
Softversko inženjerstvo je spiralni proces kod kojeg se tijekom čitavog 'života' softvera stalno radi na zahtjevima, dizajnu, implementaciji i testiranju.
Nakon što se korisnicima distribuira prva verija sustava (nekada čak i prije ove faze), odmah se počinje raditi na drugoj verziji. Ovakav pristup koristi se kada je ista organizacija zadužena i za razvoj i za evoluciju softvera. Većina generičkog softvera je razvijena na ovaj način.
Kod korisničkog softvera to ne mora biti tako. Jedna organizacija može biti zadužena za razvoj, a druga za evoluciju sustava. Nailazi se na diskontinuitete u procesu softverskog inženjerstva. Kada tranzicija između razvoja i evolucija nije jako usko povezana (seamless), proces mijenjanja softvera nakon dostave naziva se *održavanje softvera*.

Dinamika evolucije programa

Dinamika evolucije programa je proučavanje mijenjanja sustava.
Dinamiku evolucije softvera su najviše istraživali Lehman i Belady. Iz tih istraživanja proizašao je skup zakona (Lehmanovi zakoni) koji su vezani za mijenjanje sustava.
Lehmanovi zakoni:

1. Održavanje sustava je neizbježan proces, jer se zahtjevi i okolina sustava stalno mijenjaju.
2. Nakon što se sustav promijeni, njegova struktura je degradirana. Ovo se može izbjeći jedino ulaganjem u preventivno održavanje, gdje se radi na poboljšanju strukture sustava, dok se samom sustavu ne dodaju nove funkcionalnosti.
3. Veliki sustavi imaju vlastitu dinamiku koja se formira u ranoj fazi razvojnog procesa. Ta dinamika ograničava broj mogućih promjena u sustavu. Nakon što sustav preraste neku minimalnu veličinu, postane ga sve teže mijenjati. Budući da je velik i kompliciran, teže ga je razumijeti, veća je vjerojatnost da će programeri napraviti više grešaka. Dakle, trebale bi se uvoditi male promjene kako bi se izbjeglo smanjenje pouzdanosti sustava. Velika promjena će sigurno uzrokovati puno novih mana u sustavu.
4. Većina velikih programskih projekata radi u tzv. zasićenom stanju. To znači da promjena u resursima ima jako malen efekt na evoluciju sustava.
5. Dodavanje nove funkcionalnosti u sustav neizbježno dovodi do novih mana u sustavu.

Ovih je prvih pet inicijalnih Lehmanovih zakona. Šesti i sedmi zakon su slični i kažu da će se nezadovoljstvo korisnika sa sustavom povećavati ako se on ne održava i ako mu se ne dodaju nove funkcionalnosti. Osmi zakon govori o radu kod feedback procesa.

Održavanje softvera

Održavanje softvera je generalni proces mijenjanja sustava nakon što se on dostavi korisnicima. Termin 'održavanje softvera' se obično veže za korisnički softver, gdje je jedan tim zadužen za razvoj, a drugi za evoluciju sustava.

Postoje tri tipa održavanja softvera:

1. Održavanje koje je usredotočeno na ispravljanje mana u sustavu – Relativno je jeftino ispraviti greške u kodiranju, dok je ispravljanje grešaka u dizajnu nešto skuplje. Popravljanje grešaka u zahtjevima je najskuplje jer se mora obaviti opsežan redizajn čitavog sustava. Otprilike 17% ukupnog održavanja otpada na ovo održavanje.
2. Održavanje koje je usredotočeno na to da softver prilagodi različitim radnim okolnostima – Ovakvo je održavanje potrebno kada se promijeni sistemski hardver, operacijski sustav na kojem sustav radi ili neki pomoćni softver. Otprilike 18% ukupnog održavanja otpada na ovo održavanje.
3. Održavanje koje je usredotočeno na dodavanje ili mijenjanje sistemskih funkcionalnosti – Ovo je održavanje potrebno kada se promijene sistemski zahtjevi. Otprilike 65% ukupnog održavanja otpada na ovo održavanje.

Održavanje poslovnih aplikacija ima otprilike istu cijenu kao i njihov razvoj, dok je održavanje real-time embedded sustava 4 puta skuplje od njihovog razvoja.

Faktori cijene održavanja softvera:

1. Stabilnost tima – Održavanje je jeftinije ako je za njega zadužen razvojni tim, jer bi novi tim prvo trebao razumijeti softver.
2. Odgovornost koja proizlazi iz ugovora – Razvojni tim ne mora biti ugovorom obvezan da održava softver.
3. Sposobnosti tima – Tim koji održava softver je često neiskustan i neupoznat sa aplikacijskom domenom.
4. Struktura i starost programa – Kako program stari, njegova struktura se degradira, te on postaje teži za razumijeti i mijenjati.

Predviđanje održavanja (Maintenance prediction)

Menadžeri mrze iznenađenja, pogotovo ako ona rezultiraju neočekivano velikim cijenama. Zato bi se trebale predvidjeti moguće promjene u sustavu i dijelovi sustava koje će biti najteže održavati te procijeniti ukupan trošak za održavanje sustava u nekom određenom vremenskom periodu. Ova predviđanja su usko povezana: da li će se neka promjena prihvatiti ovisi o održavanju komponenti na koje će ta promjena utjecati; implementiranjem promjena degradira se struktura sustava; cijena održavanja ovisi o broju promjena a cijena implementacije promjena ovisi o održavanju komponenti sustava.
Da bi se moglo procijeniti koliko će biti zahtjeva za promjenama u sustavu, potrebno je razumijeti odnos između sustava i njegove okoline. Mora se u obzir uzeti:

1. Broj i kompleksnost sistemskih sučelja – što su sučelja kompliciranija i što ih ima više, veća je vjerojatnost da će se morati mijenjati.

2. Broj nestabilnih sistemskih zahtjeva – zahtjevi koji su vezani uz organizacijsku politiku i procedure su nestabilni.
3. Poslovne procese u kojima se sustav upotrebljava – kako se poslovni procesi razvijaju, generiraju nove sistemske zahtjeve.

Da bi se procijenilo održavanje sustava, potrebno je shvatiti broj i tipove veza između sistemskih komponenti kao i njihovu kompleksnost. Što su sustav ili komponenta kompleksniji, skuplje ih je održavati. Kod procjenjivanja održavanja sustava, u obzir možemo uzeti: broj zahtjeva za ispravljanjem grešaka u sustavu, prosječno vrijeme potrebno za određivanje broja komponenti koje su zahvaćene promjenom u sustavu, prosječno vrijeme potrebno da bi se implementirao zahtjev za promjenom, broj zahtjeva za promjenama.
Da bi se odredila cijena održavanja, koriste se informacije o broju zahtjeva za promjenama i o održavanju sustava.

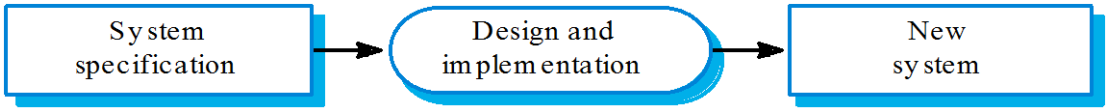
Proces evolucije

Evolucija sustava započinje zahtjevom za promjenama u sustavu, a traje sve dok traje životni vijek samog sustava. Ovisi o tipu softvera koji se održava, o razvojnim procesima koji se upotrebljavaju i o sposobnostima i iskustvu ljudi koji su zaduženi za evoluciju. Proces evolucije uključuje analizu promjena, planiranje nove verzije sustava, implementaciju sustava i distribuciju sustava korisnicima.
Prvo se razmatra cijena implementacija predloženih promjena i njihovo djelovanje na ostatak sustava. Ako se zahtjevi za promjenama prihvate, planira se nova verzija sustava koja se distribuira korisnicima nakon implementacije promjena. Proces se zatim ponavlja, tako da u obzir uzme neke nove zahtjeve za promjenama.
Proces implementacije promjena je iteracija razvojnog procesa. Međutim, kritična razlika između procesa implementacije i razvojnog procesa je da proces implementacije počinje sa razumijevanjem programa.
Kritični zahtjevi za promjenama moraju se implementirati jako brzo. Mogu nastati zbog tri razloga: ako se pronađe mana u sustavu koja onemogućuje normalan rad sustava, ako promjene u radnoj okolini sustava onemogućavaju normalan rad sustava te ako dođe do neočekivanih promjena unutar organizacije koja upotrebljava sustav.
Problem kod implementacije kritičnih zahtjeva je da se te promjene često zaboravljaju dokumentirati, te dokumentacija i kod nisu usklađeni. Budući da se ovakve promjene moraju obaviti jako brzo, ne traži se najbolje rješenje nego najbrže rješenje, čime se ubrzava proces starenja sustava te se povećava cijena daljnjeg održavanja.

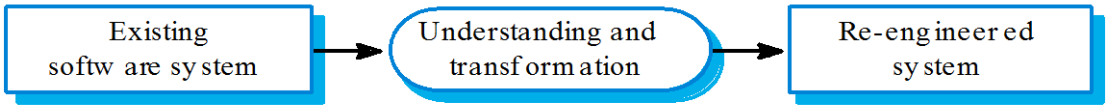
System re-engineering

Proces evolucije sustava uključuje razumijevanje programa koji se mora promijeniti te implementacije potrebnih promjena. Međutim, mnoge je sustave teško razumijeti i promijeniti, posebno starije legacy sustave.

Da se pojednostave problemi mijenjanja legacy sustava, mogu ponovo proći kroz proces re-engineeringa da bi se poboljšala njihova struktura i razumljivost. System re-engineering se brine o ponovnoj implementaciji legacy sustava tako da bi ih se lakše održavalo. Može uključivati ponovno pisanje dokumentacije sustava, restrukturiranje sustava, prevođenje koda u neki noviji programski jezik, mijenjanje i obnovu strukture sustava. Funkcionalnost sustava se ne mijenja te arhitektura sustava ostaje ista.
System re-engineering ima neke prednosti nad običnom evolucijom sustava: manji rizik i manja cijena.
Razlika između system re-engineering-a i novog razvoja softvera (*forward engineering*) je u polaznoj točki za razvoj. Novi razvoj softvera polazi od dokumenta sistemske specifikacije, dok se kod re-inženjerstva stari sustav ponaša kao specifikacija za novi sustav.
Aktivnosti kod system re-engineeringa: prevođenje koda u novi programski jezik, analiziranje programa da bismo ga mogli razumijeti, poboljšanje strukture programa, reorganizacija strukture programa, očistiti i restrukturirati sistemske podatke.
Faktori koji utječu na cijenu re-engineering procesa: kvaliteta softvera koji prolazi kroz taj proces, dostupnost pomoćnih alata (CASE alati), količina podataka koja se treba konvertirati, dostupnost iskusnog osoblja.



Forward engineering



Softw are re-eng ineering

Slika 4.1 – Forward engineering and System re-engineering

Glavna mana softverskog re-engineeringa je što postoje neke granice u sustavu nakon kojih proces softverskog re-engineeringa više nije moguć.

Evolucija legacy sustava

Organizacije koje imaju ograničen budžet za održavanje legacy sustava, moraju odlučiti koja je najbolja strategija za evoluciju takvih sustava. Odabir strategije ovisi o kvaliteti sustava i o njegovoj značajnosti za posao.
Postoje 4 takve strategije:

1. U potpunosti odbaciti sustav.
2. Ostaviti sustav nepromijenjen i nastaviti s uobičajenim održavanjem.
3. Obaviti proces re-engineeringa da bi se poboljšalo održavane.
4. Zamijeniti čitav sustav ili jedan njegov dio s novim sustavom.

Ako su i kvaliteta sustava i njegova značajnost za posao mali, održavanje takvih sustava bi bilo skupo i nepraktično, te bi se ovakvi sustavi trebali u potpunosti odbaciti.
Ako je kvaliteta sustava mala, ali je njegova značajnost za posao velika, znači da je sustav skupo održavati. Trebao bi se obaviti proces re-engineeringa.
Ako je kvaliteta sustava velika, ali je njegova značajnost za posao mala, sustav je jeftino održavati. Ne treba ih odbacivati sve dok ne počnu zahtijevati skupe promjene.
Ako su i kvaliteta sustava i njegova značajnost za posao veliki, trebalo bi se nastaviti s normalnim održavanjem.
Da bi se odredila značajnost nekog sustava za posao (poslovni pogled na legacy sustave), moraju se identificirati osobe zainteresirane za taj sustav, i pitati ih određena pitanja o sustavu. Pitanja koja se tim osobama postavljaju su sljedeća: postoji li definirani procesni model i da li se on slijedi, da li različiti dijelovi organizacije upotrebljavaju različite dijelove sustava za iste funkcije, koji su odnosi sa drugim poslovnim procesima i da li su oni potrebni, kako je proces prilagođen sustavu, da li je proces efektivno podržan od strane aplikacijskog legacy softvera.
Da bi se odredila kvaliteta sustava (tehnički pogled na legacy sustave), moraju se u obzir uzeti i sustav i okolina unutar koje on radi. Okolina uključuje hardver i asocirani pomoćni softver (kompajleri, linkeri) koji su potrebni za održavanje sustava. Okolina je važna jer jako puno promjena u sustavu je rezultat promjena u okolini sustava.
Da bi se procijenila okolinska kvaliteta sustava, važni su sljedeći faktori: stabilnost dobavljača (ako ta tvrtka više ne radi, postoji li neka druga koja će održavati sustav), učestalost pada sustava, starost, performanse, zahtjevi za podrškom (podrška za hardver i softver), cijena održavanja, komunikacija sa drugim sustavima (odvija li se bez problema).
Da bi se procijenila tehnička kvaliteta sustava, važni su sljedeći faktori: razumljivost sustava (koda, strukture, varijabli), dokumentacija (je li konzistentna i potpuna), podaci (jesu li up-to-date i konzistentni), performanse, programski jezik (podržavaju li ga moderni kompajleri), konfiguracijski menadžment, testiranje podataka, iskustvo osoblja koje održava aplikaciju.
Da bi se procijenila kvaliteta aplikacije, važni su sljedeći faktori: broj zahtjeva za promjenama, broj različitih korisničkih sučelja koje sustav koristi, volumen podataka koje sustav koristi (više podataka, kompleksniji sustav).