Konstruisanje (tj. izgradnja tj. neposredna proizvodnja tj. izrada) softvera (eng. software construction) je deo ukupnog procesa razvoja softvera. Razvoj softvera (eng. software development) je vrlo komplikovan proces,posebno kad se radi o većim projektima tj. većem softveru, i on obuhvata čitav niz aktivnosti:

• početno definisanje problema prk

• razvoj zahteva

• konstrukciono planiranje tj. planiranje konstruisanja

• dizajn softverske arhitekture tj. arhitektonski dizajn ( eng. architectural design) tj. dizajn na visokom nivou abstrakcije

• detaljni dizajn tj. dizajn na niskom nivou abstrakcije (eng. low-level design), tj. dizajn pojedinih komponenti softvera

• kodiranje (eng. Coding),

• eliminacija greški tj. debugging tj. ispravke softvera,

• testiranje jedinica softvera (eng. unit testing)

• integracija tj. integracija softverskih jedinica (eng. Integration),

• integraciono testiranje (eng. integration testing),

• sistemsko testiranje (eng. SystemTtesting)

, • korektivno održavanje

**Konstruisanje softvera je važno iz sledećih razloga:**

* • konstruisanje je jedan veliki deo u procesu razvoja softvera, npr. od 30% do 80% u zavisnosti od tipa i veličine projekta
* • konstruisanje je centralna aktivnost razvoja softvera, tj. arhitektura softvera i zahtevi softvera se urade pre konstruisanja a sistemsko testiranje posle konstruisanja u cilju verifikacije korektnosti konstruisanja
* • produktivnost individualnog programera može da varira veoma puno (faktor 10) pa je potrebno fokusirati se na efikasno konstruisanja softvera
* • konstruisanje softvera je obavezni deo razvoja softvera, naime, pre konstruisanja treba pažljivo uraditi razvoj zahteva i arhitektonski dizajn a posle konstruisanja treba uraditi detaljno sistemsko testiranje, ali često u praksi tj. u realnom životu se odmah počne sa konstruisanjem bez pažljivog razvoja zahteva i arhitekture, a sistemsko testiranje se često uradi na brzinu jer je projekat premašio rok završetka

KONTEKST KONSTRUISANJA SOFTVERA

Konstruisanje softvera je deo životnog ciklusa softvera i treba znati kontekst konstruisanja softvera u okviru životnog ciklusa softvera

STANDARDI

Standardni relevantni za konstruisanje softvera predstavljaju standarde za format i sadržaj dokumenta, programske jezike, platforme, notacije modela softvera i IEEE standard

Sledeći standardi su relevantni za konstruisanje softvera:

• Standardi za format i sadržaj dokumenata,

• Standardi za programske jezike, npr. standardi za C++ I Javu

• Standardi za platforme

• Standardi za notacije modela softvera, npr. UML notacija

• IEEE Std 1016-1998, Recommended Practice for Software Design Descriptions, itd.

Ime nekog standarda sadrži redni broj nekog dokumenta standarda, godinu usvajanja standarda, i ime standarda

IEEE Standards-lista: Ime nekog standarda sadrži redni broj nekog dokumenta standarda, godinu usvajanja standarda, i ime standarda. Evo nekih relevantnih standarda, Software development standards: IEEE Std 830-1998, Recommended Practice for Software Requirement Specification,

IEEE Std 1233-1998, Guide for Developing System Requirements Specifications, 9

ITD

Evo liste standarda za upravljanje softverskim projektima:

• IEEE, Standard for Software Project Management Plans,

• IEEE, Standard for Developing Sofware Life Cxcle Processes

• IEEE, Standard for Software Productivity Metrics • itd.

Po pravilu, programeri vole da pišu programsku dokumentaciju. Dobra programska dokumentacija je ponos programera, jer je to finalni rezultat njegovog rada. Softverska dokumentacija može da ima puno formi, a jedan specifičan oblik su i komentari u programu.

**Dokumentacija softverskog projekta** obuhvata kako informacije u okviru listinga programskog koda, ali i izvan tih listinga, odvojeni dokumenti koji opsuju razvoj pojedinih edinica softvera. Kod velikih projekata, većina dokumntacije je izvan listinga, i ovo je tzv. eksterna dokumentacija, external documentation. Eksterna konstrukciona dokumentacija je na višem nivou softvera u poredjenju sa izvornim kodom, i na nižem nivou softvera u poredjenju sa zahtevima softvera i arhitekture softvera.

**Detaljni dizajnerski dokument,** detailed-design document,opisuje detaljni tj. low-level dizajn (dizajn na nižem nivou softvera). On opisuje dizaj**n na nivou klase (class-level)** i **na nivou podprograma(routine-level)**, odluke i alternative koje su azmotrene u toku procesa detaljnog dizajna.

Za razliku od eksterne dokumentacije, interna dokumentacija je dokumentacija u okviru listinga programskog koda. To je najdetaljnija dokumentacija, na nivou programskog koda. Ova unutrašnja dokumentacija obuhvata komentare u programu, izbor imena varijabli i konstanti je vrlo bitan, izbor imena podprograma, dobar raspored (layout) pojedinih delova programskog koda, itd.

## Pogledajmo listu tipičnih softverskih dokumentata.

1) **Statement of work** (Preliminarni kratki opis funkcionalnosti softvera-od strane korisnika);

Specifikacija softverskih zahteva (Software requirements specification) – Opisuje šta ce i kako raditi završeni softver.

2) **Model objekata/klasa** (Object/class model) – Opisuje glavne tzv. objekte/klase podataka softvera. I Korisnički scenariji (Use case scenarios) – Opis serije nizova mogućih ponašanja softvera sa korisničke tačke gledišta.

1. **Projektni vremenski plan** (Project schedule) – Opis redosleda zadataka i procena vremena i potrebnih napora.
2. **Plan testiranja softvera** (Software test plan) – Opis testiranja softvera. Testovi prihvatljivosti (Acceptance tests)– Testovi naznačeni od strane korisnika da odrede prihvatljivost softvera. Izveštaj o testiranju (Test report) - Specificira koji testovi su obavljeni i kako se softver ponašao. Izveštaj o defektima (Defect report) – Opisuje nezadovoljstvo korisnika sa konkretnim nezadovoljstvom korisnika zbog konkretne greške ili nedostataka softvera.
3. **Softverski dizajn** (Software design) – Opisuje strukturu softvera. Arhitektonski dizajn (Architectural design) – Opisuje strukturu na gornjim hijerarhijskim nivoima softvera, zajedno sa interkonekcijama (interkonekcijama izmedju pojedinih modula tj. blokova). Detaljni dizajn (Detailed design) – Opisuje dizajn modula ili dizajn objekata (klase) tj dizajn na najnižem hijerarhijskom nivou softvera.
4. **Plan obezbedjenja kvaliteta** (Quality assurance plan) – Preciziranje aktivnosti koje će biti izvršene u cilju obezbedjenja kvaliteta.
5. **Korisničko uputstvo** (User manuel)- Objašnjava kako da se koristi završen softver.
6. **Izvorni kod** (Source code) tj. programske instrukcije – Listing softverskih instrukcija (program u izvornom obliku/kodu).
7. **Test repor**t (Opisuje testove i kako se sistem ponaša),

10) **Defect report** (Opisuje nezadovoljstvo korisnika, tj. greške i nedostatke softvera)

**Data-dictionary** tj. rečnik podataka je baza podataka koja opisuje sve značajne podatke u projektu. Npr.ovaj alat sadrži

• u velikim projektima stotine ili hiljade definicije klasa

• na velikim projektima omogućuje da se ima pregled imenovanja klasa itd, i izbegavanja korišćenja istih imena ili nesaglasnih imena

• ovaj alat omogućuje optimizaciju izbora imena u projektu za pojedine varijable, klase, itd

**Teorema formatiranja: dobar vizuelni raspored pokazuje logičku strukturu programa**

**Mogu se navesti sledeći ciljevi dobrog struktuiranja:**

• tzv. "beli prostor" ("white space") , npr. "uvlačenje" instrukcija ("indentation"), se koristi da vizualizuje logičku strukturu programskog koda

• povećava se čitljivost tj. razumljivost programa

• omogućuje se lako modifikovanje programa

## L2

Evo tipičnih razloga za greške inicijalizacije varijabli:

• varijabli nije nikad zadata vrednost

• jedan deo objekta je inicijalizovan ali ne ceo (nekoliko članova u objektu nije inicijalizovano)

• varijabli nije alocirana memorija

• varijabla je inicijalizovana, ali ta vrednost više ne važi već treba zadati novu vrednost

**PRAVILA ZA IME**

• ime varijable treba da tačno i u potpunosti opisuje šta dotična varijabla reprezentuje

• ime varijable po pravilu treba da opisuje problem koji se rešava a ne način rešavanja pomoću konkretnog programskog jezika

• treba optimizirati dužinu imaena varijable da ne bude suviše dugačka ali ne i suviše kratka

• ako ime varijable treba da sadrži reči kao što su Max, Min, Total, Sum itd., onda tu reč (tzv. "modifikator varijable") treba staviti na kraj imena a ne na početak, jer prvo treba naznačiti smisao varijable a tek onda modifikator varijable

Evo konvencija za imenovanje varijabli:

• razlikovati imena varijabli i imena potprorama, npr. variableName i RoutineName(), dakle imena varijabli i objekata počinju malim slovima a imena potprograma velikim slovima

• razlikovati imena klasa i imena objekata

• identifikovati globalne varijable, npr. sa prefiksom "g", npr. varijabla g\_Temperature

• identifikovati klasne varijable ("member variables") sa "m", npr. m\_Ime

• identifikovati imenovane konstante, npr. u Javi i C++ se koriste velika slova npr. TEMPERATURE\_MAX

• identifikovati enumeracije (enumerated type) pomoću prefiksa "E" • ako imamo ime varijable sa više reči, onda svaka reč počinje velikim slovom ili praviti razmake sa crtom, npr. room\_temperature ili roomTempoerature

• pojedini jezici imaju svoje konvencije, i treba ih koristiti, npr. u C++ su imena varijabli i imena podprograma su kompletno u malim slovima, npr. room\_temperature, a u Javi imena varijabli i metoda koriste malo slovo za prvu reč a velika slova za ostale reči, npr.roomTemperature

**Magični broj, ili potpis** je celobrojna ili tekstualna konstanta koja se koristi za jedinstvenu identifikaciju resursa ili podataka.

Takav broj sam po sebi nema nikakvo značenje i može

izazvati zabunu ako se pojavi u programskom kodu bez odgovarajućeg konteksta ili komentara, dok pokušaj njegove promene u drugi, čak i po vrednosti, može dovesti do apsolutno nepredvidivih posledica. Iz tog razloga su se takvi brojevi ironično nazivali magijom.

Evo sugestija za upotrebu stringova (strings) i znakova (characters):

• izbegavati "magične" stringove, tj. stringove čije značenje nije jasno • izbegavati "magične" znakove

Konačno, evo preporuka za upotrebu nizova:

• proveriti da se indeks niza kreće unutar dimenzije niza, a ne da prevazilazi dimenziju niza, jer u nekim jezicima biće javljena greška, a u nekim jezicima neće biti javljena greška ali ćete dobiti pogrešan rezultat

G**lobalne varijable**

Ponekad je korisno primeniti globalne podatke, a povesti računa o povećanom riziku. Ipak, treba izbegavati primenu globalnih podataka.

Evo razloga za upotrebu globalnih podataka:

• neki podaci se koriste u celom programu, i onda je korisno takve podatke koristiti kao globalne podatke

• globalni podaci mogu da pojednostave program tamo gde se neki podaci koriste u većem delu programa ili se često pojavljuju listi parametara čitavog niza potprograma

**Cesto kod sekvencijalnih instrukcija je bitan redosled instrukcija, npr. instrukcija1 učitava podatke, instrukcija2 vrši proračun na učitanim podacima, a instrukcija3 štampa rezultate prora**

**Dubogo ugneždavanje tj. Deep Nesting,** dublje od 3 ili eventualno 4 nivoa, treba izbegavati, jer je teško za razumevanje. Ali, nije teško izbeći duboko ugneždavanje, i postoje tehnike za to. Naime, može se 1)ili redizajnirati if-else instrukcija, ili 2) se može refaktorisati programski kod u jednostavnije podprograme.

Control flow (kontrola protoka) u nekom program označava redosled ukom se individualne instrukcije ili funkcije izvršavaju/pozivaju, kod izvršavanja programa ili evaluacije programa.

Sledeće su vrste kontrolnih instrukcija:

• Skok tj. bezuslovno grananje

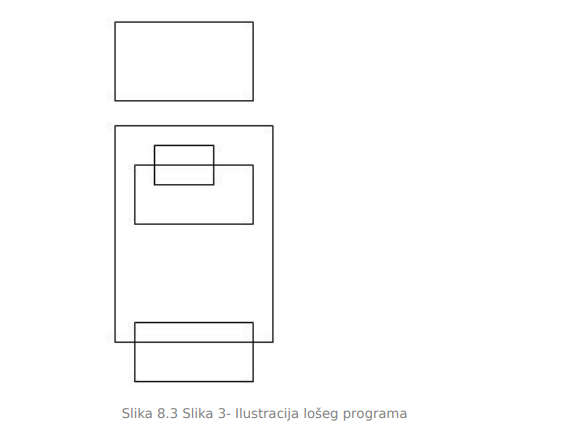
• Uslovno grananje tj. izbor

• Petlja

• Podprogrami (udaljeni skup instrukcija koji se poziva po potrebi) • Zaustavljanje programa

Instrukcija return je u jeziku C kontrolna konstrukcija koja omogućuje programu da izadje iz potprograma po potrebi.

Instrukcije koje su u vezi treba grupisati, u blokove. Ako ima preklapanja na pojedinim blokovima, konstrukcija kao na drugoj slici, to je znak da je izv. kod loše organizovan.



#### KONCEPT STRUKTURNOG PROGRAMIRANJA

Tehnika strukturnog programiranja daje programeru na upotrebu veoma korisne logićke uzorke u grafičkoj formi, koje je lakše analizirati i proveravati nego sam izvorni kod programa.

Strukturno programiranje je veliki korak napred u odnosu na nestrukturno programiranje. Kod nestruktuiranog programiranja, ceo program se sastoji od jednog bloka, ponavljanje operacija i grananje u programu se obavlja pomoću GOTO i JUMP naredbi. Takav program se teško razume, teško ispravlja, a lako se greši u kodu. Struktuirano programiranje deli veliki program u modularne procedure.

**Proceduralno programiranje** nudi standardna rešenja za računarske aplikacije, u blokovima izv. koda, koji se mogu pozvati pomoću imena potprograma, koji nude višestruku aplikaciju, ponovnu upotrebu, pomoću liste argumenata, tj. ulaznih podataka. Promenljive koje su deklarisane u okviru procedure tj. potprograma, kao npr. ’sum’, su vidljive jedino u okviru podprograma. Jednom kada se ima gornji potprogram, on se može koristiti ponovo, i opet ponovo. Takva ponovna upotreba može se ostvariti:

• uključivanjem tog potprograma u bilo koji program koji zahteva takav podprogram.

• Ili se može taj potprogram staviti u biblioteku potprograma, odakle se mogu pozivati pomoću imena potprograma.

#### SLOZENOST PODPROGRAMA

Složenost nekog programa uveliko zavisi od upotrebe kontrolnih struktura, tj. od upotrebe ifinstrukcija i petlji, i case-instrukcija. Evo tehnike za merenje kompleksnosti potprograma:

1. Započeti sa brojem 1 koji označava osnovnu putanju kroz potprogram
2. Dodaj 1 za svaku ključnu reč tipa if, while, for, and, or ,
3. I dodaj 1 i za svaki pojedinačni case-slučaj tj. za svaku pojedinačnu granu u okviru caseinstrukcije

**Ako je stepen kompleksnosti veći od 10, to je indikacija da podprogram treba razbiti na dva potprograma.**

Upotreba "goto" instrukcija (i njenih ekvivalenata) kvari koncept strukturnog programiranja, i treba ga izbegavati jer dobijamo program koji je teško razumeti, održavati, i teško kompajlirati.

#### L3

Detaljni dizajn to je dizajniranje algoritama za pojedine delove (tj. tzv. jedinice (units)) softvera, i dizajniranje pojedinih objekata.

**Životni ciklus softvera** definiše se kao serija različitih aktivnosti koje se dešavaju u toku razvoja softvera. Takodje, pojavljuju se različiti produkti (deliverables) proizvedeni tokom životnog ciklusa softvera, npr. softverski izvorni kod (source code) i korisničko uputstvo (user manuel). Sledeće su aktivnosti koje čine životni ciklus softvera:

1. **Analiza izvodljivosti (Feasibility study)** – Odredjivanje da li je predloženi razvoj svrsishodan(vredan da se izvodi).
2. **Analiza tržišta (Market analysis) –** Odredjivanje da li postoji potencijalno tržište za predloženi proizvod.
3. **Odredjivanje zahteva (Requirements determination) –** Odredjivanje (specificiranje) koje funkcije bi softver trebao da sadrži (odredjivanje funkcionalnosti). Prikupljanje zahteva (Requirement elicitation): obezbedjivanje zahteva od korisnika.
4. **Analiza domena (Domain analysis):** Odredjivanje koji ciljevi i strukture su zajedničke za postavljeni problem.
5. **Planiranje projekta (Project planning)** – Odredjivanje kako da se razvije softver. Analiza troškova (Cost analysis)– Procena troškova razvoja softvera. Vremensko planiranje (Scheduling) – Izrada vremenskog plana razvoja softvera. Obezbedjivanje kvaliteta (Quality assurance): definisanje aktivnosti koje će obezbediti kvalitet softvera. Pregled radova (Work-breakdown structure) – definisanje podciljeva potrebnih da se razvije softver.
6. **Softverski dizajn (Software design)** - Kreirati softver da omogući zeljenu funkcionalnost (da ispuni željene zahteve). Arhitektonski dizajn(Architectural design) – Dizajniranje strukture softvera (softverskog sistema). Detaljni dizajn (Detailed design) – Dizajniranje algoritama za pojedine delove (tj. tzv. jedinice (units)) softvera, i dizajniranje pojedinih objekata. Dizajniranje interfejsa (Interface design) – Dizajniranje interfejsa (interakcije) izmedju pojedinih delova softvera.

7) **Implementacija softvera (Software implementation)** – Izgradnja softvera, tj pisanje koda, kao i debugging

8) **Testiranje (Testing)** – Egzekucija (pogon) softvera sa podacima u cilju obezbedjivanja/ provere ispravnog rada softvera. Pojedinačno testiranje (Unit testing) - Testiranje pojedinih delova softvera od strane prvobitnog kreatora/proizvodjača softvera. Integraciono testiranje (Integration testing) – Testiranje tokom integracije pojedinih delova softvera. Sistemsko tesiranje (System testing) – Testiranje softvera u uslovima/okruženju koje odgovara operacionim-pogonskim uslovima. Alfa testiranje (Alpha testing) – Testiranje od strane korisnika softvera na lokaciji kod proizvodjača softvera. Beta testiranje (Beta testing) – Testiranje od strane korisnika na lokaciji kod korisnika softvera. Testiranje prihvatljivosti (Acceptane testing) – Testovi u cilju zadovoljenja korisnika/kupca.

**9) Isporuka (Delivery)** – Isporuka korisniku efektivnog softverskog rešenja. Instalacija (Installation) – Instalisanje softvera na loaciji kod korisnika. Treniranje (Training) – Treniranje korisnka da koristi softver. Tehnička podrška (Help desk) – Odgovaranje na pitanja korisnika.

**10) Održavanje (Maintenance**) – Ažuriranje i poboljšavanje/modifikovanje softvera u cilju obezbedjenja korisnosti u odredjenom vremenskom periodu.

### KONSTRUISANJE SOFTVERA - AKTIVNOSTI

##### Šta je to konstruisanje softvera?

Konstruisanje softvera je deo ukupnog procesa razvoja softvera i obuhvata detaljni razvoj softvera.

Konstruisanje softvera (software construction) je jedna velika oblast tj. podoblast softverskog inženjerstva, koje obuhvata:

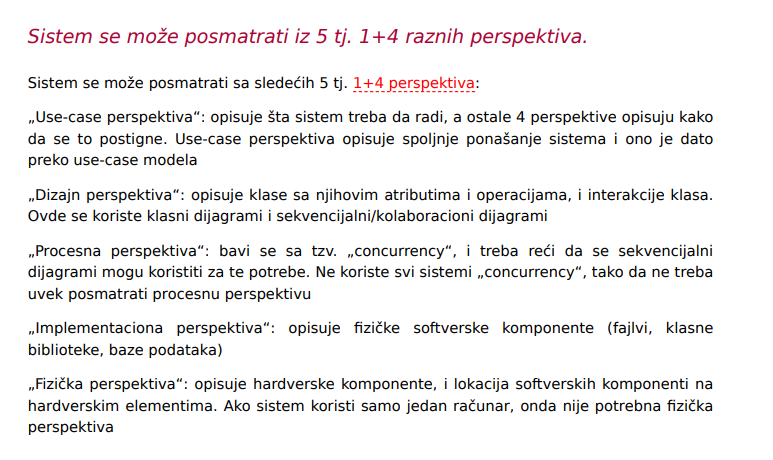
**detaljni dizajn softvera, kodiranje, testiranje jedinica softvera (unit testing), integraciono testiranje, eliminaciju defekata tj. debugging, i verifikaciju softvera.**

Dakle, konstruisanje softvera je jako povezano i značajno se preklapa se sa dizajnom softvera i testiranjem softvera (koje takodje predstavjaju discipline softverskog inženjerstva) i verifikacijom softvera.

**Verifikacija softvera** obuhvata testiranje softvera (unit testing tj. testiranje funkcija i klasa, zatim testiranje modula tj. grupe klasa, zatim integraciono testiranje gde se testira nekoliko modula zajedno I testiranje sistema gde se testira ceo system, kao i funkcionalno testiranje, i nefunkcionalno testiranje tj. performance test).

Verifikacija obuhvata i inspekciju programa pre njegovog izvršavanja tj. code reviews (gde ima nekoliko tehnika).

Konstruisanje softvera podrazumeva upotrebu standarda, npr. standardi za format dokumenata I sadržaj dokumenata. Takodje, kod konstruisanja softvera se koristi UML (Unified Modelling Language) kao standardni model (notaciju) softvera. Konstruisanje softvera je deo ukupnog procesa razvoja softvera i između ostalog obuhvata i detaljni razvoj softvera. Kod razvoja softvera koriste se mnogi modeli, a jedan broj ovih modela se koristi kod konstruisanja softvera. Npr. UML modeli softvera. Ali takodje i procesni modeli koji opisuju process razvoja softvera: “vodopadni model” (linearni model), ili iterativni modeli kao što su “evolutivni prototajping” (evolutionary prototyping), ekstremno programiranje (extreme programming), Scrum model.



Softverski razvojni proces koji koristi UML (dakle, za o.o. sisteme) je poznat kao Unified Software Development Process tj. skraćeno Unified Process- UP, i koji je predložen od strane autora UML-a (Jacobson, Booch, Rumbaugh, 1999).

UP proces je definisan kao niz aktivnosti koje je potrebno uraditi da bi se od korisničkih zahteva dobio funkcionalni sistem koji zadovoljava te zahteve. UP je arhitektonski-centriran , iterativan i inkrementalan. Pod arhitektonski-centriran misli se da je sistemska arhitektura razvijena da zadovolji ključnih use-case slučajeva u smislu platforme na kojoj će sistem da se koristi i u smislu strukture sistema i podsistema.

UP se sastoji od četri faze:

početak, elaboracija, konstrukcija, i tranzicija.

Svaka faza može da sadrži nekoliko iteracija. Svaka iteracija je u stvari jedan mini-projekt.

Rezultat primene UP je niz modela:

• Use-case model

• Model analize

• Model dizajna

• Fizički model tj. model „razmeštanja“ (deployment model)

• Implementacioni model

• Test model

MAD FIT

Medjutim, UP se takodje može podeliti na sledeće aktivnosti tj. modele tj. podprocese (workflows):

• Zahtevi

• Analiza (model analize)

• Dizajn (model dizajna)

• Implementacija (implementacioni model)

• Testiranje (test model)

Zahtevi se menjaju tokom izrade projekta, i zato treba koristiti requirement checklist tokom konstruisanja.

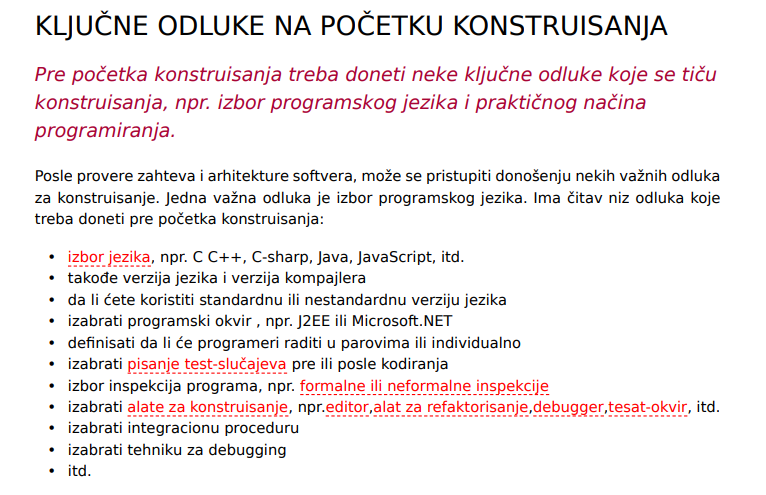
Bez dobre specifikacije arhitekture u startu, možete da imate pogrešno rešenje problema.

Sumiramo šta je i zašto važno proveriti pre početka konstruisanja. Pre početka konstruisanja treba proveriti:

• koja vrsta projekta i koji model životnog ciklusa izabrati?

• da li su zahtevi dovoljno dobro definisani da bi se počelo konstruisanje softvera?

• da li je arhitektura dovoljno dobro definisana da bi se počelo konstruisanje?



U praksi je vrlo važno pitanje: kako motivisati kvalitetno kodiranje? Programski kod je glavni rezultat kod konstruisanja softvera, pa se postavlja pitanje kako motivisati kvalitetno kodiranje, tj. kvalitetno pisanje programa? Postoje sledeće tehnike motivacije za to :

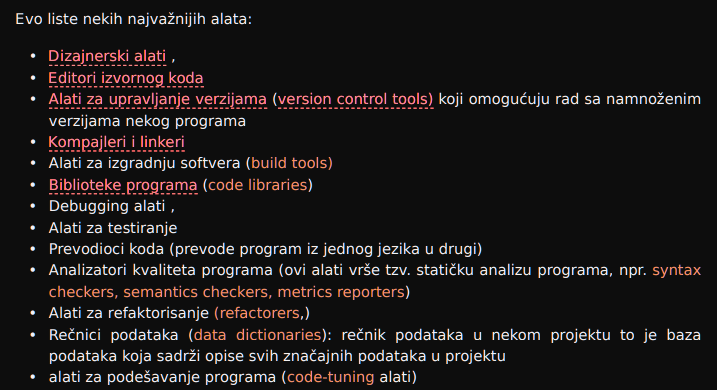
• Na svakom delu projekta angažovati 2 programera

• Pregledati svaku liniju programskog koda, gde dva ’recenzenta’ pregledaju programski kod

• Overiti programski kod od strane vodećeg inženjera

• Nagrade za dobar programski kod

## L4



**Programski alati** mogu da dramatično smanje vreme konstruisanja softvera. Naime , ako koristite najnovije alate, i ako ste familijarni sa tim alatima, produktivnost može da se poveća za npr. 50%.

Ovi alati takodje smanjuju onaj dosadni deo posla koji je prisutan kod programiranja, npr. neke stvari koje se ponavljaju u programima. Ovde posmatramo samo konstrukcione alate, dok se ovde ne bavimo alatima za specifikaciju zahteva i alatima za upravljanje projektima. Neki programeri rade godinama pre nego što otkriju neke vrlo korisne alate. Cilj ovog poglavlja je da se izvrši pregled raspoloživih alata, što pomaže programerima da otkriju da li su prevideli neki alat.

DIZAJNERSKI ALATI

S jedne strane, grafički dizajnerski alati su samo divni paketi za crtanje. Ali ovi alati nude više u odnosu na jednostavne tj. obične alate za crtanje.

Postojeći dizajnerski alati su uglavnom grafički alati za kreiranje dizajnerskih dijagrama. Dizajnerski alati su ponekad ugradjeni u CASE (computer-aided software engineering) alat koji ima šire funkcije od dizajnerskih funkcija, dok neki prodavci (vendors) prodaju samostalne (standalone) dizajnerskealate kao CASE alate. Grafički dizajnerski alati omogućuju da se dizajn predstavi u uobičajenim grafičkim notacijama dizajna:

• UML,

• Arhitektonski blok dijagrami ,

• Hijerarhijske karte ,

• Entitetski dijagrami ,

• Klasni dijagrami

Neki grafički dizajnerski alati podržavaju samo jednu od ovih grafičkih notacija, a neki podržavaju čitav niz ovih grafičkih notacija. S jedne strane, ovi grafički dizajnerski alati su samo divni paketi za crtanje. Ali ovi alati nude više u odnosu na jednostavne tj. obične alate za crtanje.

• Npr. ako nacrtate neki dizajnerski dijagram i sklonite ili dodate jedan element sa ovog dijagrama, onda će grafički dizajnerski alat da automatski rearanžira dijagram, uključujući strelice izmedju pojedinih elemenata na dijagramu, itd.

• Npr. neki dizajnerski alati omogućuju kretanje izmedju višeg i nižeg nivoa apstrakcije.

• Zatim, grafički dizajnerski alati omogućuju proveru konzistentnosti dizajna izmedju nekoliko različitih dijagrama, a neki alati o mogućuju kreiranje programa direktno iz dizajnerskog dijagrama.

IDE (integrisano razvojno okruženje- integrated development environment) omogućuju rad sa izvornim programom (source code) sadrže u sebi editore izvornog koda

• Kompilaciju i detekciju greški u okviru editora

• Integracija editora sa version-control alatima i build-alatima i debugging alatima, alatima za testiranje,

• Rezime programa tj. sažeto vidjenje programa: npr. lista imena klasa ili logička struktura programa

Skakanje u programu na definicije klasa, podprograma i varijabli

• Skakanje na mesta gde se koriste klase, podprogrami i varijable

• Formatiranje programa

• Interaktivna pomoć

• Provera zagrada (početak-kraj blokova sa zagradama)

Neki programeri tvrde da oko 40% vremena provode editujući izvorni kod programa

Refaktoreri omogućuju lake izmene programskog koda

#### CASE alati automatizuju metode za dizajn, dokumentovanje i pravljenje strukrurisanog programskog koda u željenom programskom jeziku.

CASE (Computer aided software engineering) označava metode za razvoj informacionih sistema pomoću automatizovanih alata koji se mogu koristiti u toku procesa razvoja softvera. Takodje, može da označi upotrebu softvera za automatizovan razvoj softvera tj. programskog koda. CASE funkciju uključuju analizu, dizajn, i programiranje.

CASE alati automatizuju metode za dizajn, dokumentovanje i pravljenje strukrurisanog programskog koda u željenom programskom jeziku. CASE alati podržavaju aktivnosti softverskog procesa, kao što su zahtevi, dizajn, razvoj programa i testiranje. Prema tome, CASE alati obuhvataju:

Dizajn editore, Kompajlere , Debuggers (Detektori skrivenih greški), Itd.

ALATI ZA KONSTRUISANJE SOFTVERA (CONSTRUCTION TOOLS)

Trenutno postojeći dizajnerski alati uglavnom su grafički alati koji prave dizajnerske dijagrame.

Trenutno postojeći dizajnerski alati uglavnom su grafički alati koji prave dizajnerske dijagrame. Ovi dizajnerski alati se ponekad ugrade u CASE alat sa širim funkcijama a ne samo dizajnerskim.

Grafički dizajnerski alati omogućuju da se prikaže dizajn preko UML dijagrama, arhitektonskih blok dijagrama, hijerarhijskih dijagrama, klasnih dijagrama. Neki alati se koriste samo za jednu vrstu dijagrama, a neki alati za više vrsta dijagrama. S jedne strane, ovi dizajnerski alati su samo specijalizovani paketi za crtanje. Ako se koristi obični grafički paket za crtanje ili papir i olovka, isto se može uraditi kao i pomoću ovih specijalizovanih alata, ali ovi specijalizovani alati omogućuju veću efikasnost i tačnost. Dizajnerski alat omogučuje lako pomeranje izmedju višeg i nižeg nivoa apstrakcije, jer je dizajn jedan iterativan proces na više nivoa apstrakcije. Takodje, dizajnerski alat proverava konzistentnost dizajna, i neki alati mogu da kreiraju programski kod direktno iz dizajna tj. direktno na osnovu dokumenta koji opisuje dizajn.

Alati za kodiranje izvornog programskog koda, i za debugging (source-code tools): Jedna grupa alata omogućuje editovanje programskog koda. Npr. IDE obuhvata editor programskog koda. Neki programeri kažu da provode 40% vremena u pisanju tj. editovanju programskog koda. Tako da upotreba IDE je svakako opravdana. Alati za detekciju skrivenih greški, debugging, se isto nalaze u okviru IDE.

**„SOFTVERSKE SKELE“**

Ove pomoćne klase ili podprogrami tj- „softverske skele“ se mogu napraviti manje ili više realistični da zamenjuju ostatak programa.

TEST-DATA GENERATORS, I BAZE PODATAKA GREŠKI

Baza podataka greški, tj. Error Database, je moćan alat za testiranje.

Sledeći alati se koriste kod debugging-a:

• Test scaffolding (skele za testiranje)

• Alati za poredjenje različitih verzija programa („Diff tools“)

• Trace monitors

• Interaktivni debugger-s (Interactive Debuggers) za dijagnozu greški • kompajleri tj. njihove poruke o greškama

###### ALAT FINDBUGS

Alat FindBugs je alat koji upoređuje program sa bazom podataka sa greškama ( database of bugs).

### L5

#### RAZVOJNO TESTIRANJE

„Razvojno“ testiranje se sastoji od testiranja softverskih jedinica i testiranja softverskih komponenti.

Testiranje se obavlja od strane razvojnih softverista, ali i od strane specijalizovanih test-specijalista

Testiranje softverskih jedinica, gde se ispituje neka kompletirana klasa ili podprogram ili mali program, i to izolovano od ostalog dela sistema, pri čemu je ta jedinica napisana od strane jednog rogramera ili jednog tima programera.

Testiranje softverskih komponenti, gde se ispituje neki element programa, npr. klasa, paket, manji program, i to izolovano od ostalog dela sistema, ali koji je razvijen od strane nekoliko programera ili nekoliko timova programera.

Integracioni testovi je kombinovano ispitivanje dve ili više klasa, paketa, komponenti, podsistema, razvijen od strane nekoliko programera ili nekoliko timova programera. Sistemsko testiranje, gde se softver testira u finalnoj konfiguraciji, uključujući integraciju sa ostalim softverom i računarskom opremom

Testiranje u kompaniji koja razvija softver obuhvata testiranje softverskih jedinica, testiranje softverskih komponenti, i integracione testove.

**Sistemsko testiranje** ide posle integracije. Ponekad i **sistemsko testiranje se** obavlja u kompaniji koja je razvila softver.

**Integracija** tj. integraciono testiranje se radi u okviru testiranja, a posle „razvojnog“ testiranja. Dakle, **„razvojno“ testiranje** se sastoji os testiranja softvcerskih jedinica i testiranja softverskih komponenti.

**Testiranje i „dedefektovanje“,debugging, nije isto, naime,**

• testiranje omogućuje detekciju greški,

• a debugging, to je dijagnoza i korigovanje uzroka greške koja je već otkrivena.

#### SISTEMATSKI PRILAZ RAZVOJNOM TESTIRANJUPREPORUKE ZA TESTIRANJE

Evo od čega se sastoji jedan sistematski prilaz testiranju. Ovde se bavimo samo sa detekcijom greški, tj testiranjem. Jedan sistematski prilaz razvojnom testiranju maksimizira sposobnost nalaženja greški sa minimumom napora, i to:

• testirati **svaki relevantni softverski zahtev**, da bi se proverila implementacija zahteva za softver

• testirati **svaki problem koji je uočen u fazi dizajna**, i već tada u fazi dizajna napisati testslučajeve

• testirati **svaku liniju programa** pomoću tzv baznog testiranja

• testirati **tok-podataka**

• **koristiti listu-provere** gde se navode greške iz prošlosti, tj iz prethodnih projekata kao i iz prethodnih faza dotičnog projekta

**Razvojni softveristi obično smatraju da je pokrivanje svake linije programa adekvatno, medjutim bolji standard zahteva i testiranje svake putanje u programu.**

**Nesavršenosti razvojnog testiranja se mogu sabrati na sledeći način**

• Razvojni softveristi pre testiraju da li softver funkcioniše nego da traže slučajeve tj izuzetke kad softver ne funkcioniše

• Obično kod razvojnog testiranja se pokrije 80% testova, a vruje se da je pokrivenost 95%

• Razvojni softveristi obično smatraju da je pokrivanje svake linije programa adekvatno, medjutim bolji standard zahteva i testiranje svake putanje u programu i to kako za ispravne tako i neispravne vrednosti

Kod testiranja, korisno je posmatrati klasu kao „glass-box“ a ne „blackbox“ jer da bi se detaljno istestirala klasa potrebno je posmatrati izvorni kod i ulaze i izlaze.

Prakse kolaborativnog razvoja (**collaborative development**) u različitim formama su se pokazale da omogućuju pronalaženje velikog broja greški npr. približno isto koliko i testiranje, i ove metode kolaborativnog razvoja su upola jeftinije od testiranja. Testiranje koje obuhvata

• unit test,

• component test,

• integration test

obično detektuje ukupno oko **50-60%** postojećih greški. Dakle, **software quality assurance plan treba da uključi i kolaborativno konstruisanje da bi se otkrilo što više greški**, tj. greški koje se ne mogu otkriti tetstiranjem. Obično oko 50% ukupnog vremena razvoja softvera odlazi na debugging i testiranje, pri čemu je ovde u testiranje uključeno i ne-razvojno testiranje.

Ako razvijate nekoliko klasnih metoda, treba ih razvijati jednu po jednu a ne istovremeno.

Klasne metode ja mnogo lakše debugg-ovati jednu po jednu, nego istovremeno, jer ako dodajete jednu po jednu klasnu metodu nekoj klasi, onda bilo koja nova greška je

• ili usled greške u toj klasnoj metodi

• ili usled interakcije te nove klasne metode sa ostatkom klase

Kod testiranja neke klase postoje dve vrste:

• **black-box testiranje**, gde se gleda šta klasa radi sa gledišta korisnika te klase, i ne posmatra se klasa iznutra tj. iza interfejsa klase

• **glass-box testiranj**e tj. white-box testiranje, gde se gleda se analizira source-code klase i takođe ulazi i izlazi iz klase

ponekad se ignoriše glass-box testiranje i koristi samo black-box testiranje što je pogrešno

error

Postoji tri tehnika testiranja sofvera, a one su:

1. Black Box testiranje
2. White Box testiranje
3. Grey Box testiranje

BlackBox testiranje je metod softverskog testiranja u kojem unutrašnja struktura sistema koji se testira nije poznata testeru (tester je vidi kao crnu kutiju i ne zna kako je softver implementiran).

WhiteBox testiranje je metod softverskog testiranja u kojem su unutrašnja struktura i svi implementirani detalji sistema koje obuhvata testiranje, u potpunosti poznati testeru. Kao developer.

Testiranje kao tester.

GreyBox testiranje je metod softverskog testiranja koji kombinuje BlackBox i WhiteBox metode. U ovoj metodologiji unutrašnja struktura sistema je delimično poznata testeru a tester poznaje strukturu podataka i algoritme koji su korišćeni za razvijanje softvera.

#### SELEKCIJA TEST SLUČAJEVA

Zaustavni kriterijum može biti ili unapred specificiran broj slučajeva u svakom subdomenu testiranja ili se posmatra broj grešaka.

Testiranje softvera je izvršavanje tj. rad („egzekucija“) softvera sa test podacima. Jedan od osnovnih problema kod testiranja softvera je selekcija test-slučajeva (test-case selection). Postoje sledeće metode za selekciju test-slučajeva:

* Funkcionalna, koja se bazira na specifikaciji softverskih zahteva (software requirements specification)
* Strukturna, koja se oslanja na strukturu izvornog koda programa Data-flow, tj. testiranje toka podataka u programu
* Random tj. proizvoljno trestiranje

Sa druge strane, zaustavni kriterijum (stopping criterion) može biti ili unapred specificiran broj slučajeva u svakom subdomenu testiranja (gde se domen svih test-slučajeva deli na subdomene) ili se posmatra broj grešaka u softveru koji se pojavljuje kod testiranja, pa ako se ovaj broj smanji do neke male vrednosti koja je zadata.

**Funkcionalno**

Obično može da se generiše jedan test-slučaj za svaki različit tip izlaza iz programa.

**Strukturno**  
Kod testiranja svake instrukcije, traži se da se svaka instrukcija u programu izvrši bar jedanput od strane nekog test-slučaja u toku testiranja test-slučajevima.

STRUKTURNO OSNOVNO TESTIRANJE?

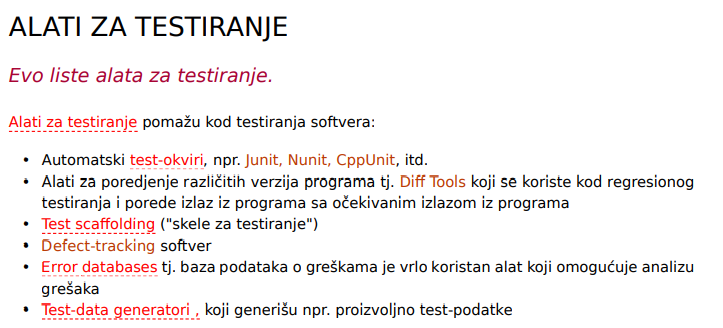
Ako je to logički iskaz, tj logiča instrukcija if ili while, onda je potrebno varirati testiranje tako da je kompletno istestiran taj iskaz

**Structured basis testing:** Ideja ovog prilaza je da svaku instrukciju u programu treba istestirati bar jednom. Ako je to logički iskaz, tj logiča instrukcija if ili while, onda je potrebno varirati testiranje tako da je kompletno istestiran taj iskaz. U nekim prilazima testiranju, testiraju se sve putanje u programu. To je slično sa prilazom strukturnog baznog testiranja, ali je potrebno dodati ideju o minimalnom skupu test-slučajeva.

**Minimalan broj putanja** tj test-slučajeva za strukturno bazno testiranje se može odrediti jednostavno na sledeći način:

startovati sa 1 za pravolinijsku putanju kroz podprogram

dodati 1 za svaki od sledećih ključnih reči, if,while,repeat,for,and, i or dodati 1 za svaki slučaj u iskazucase, a ako nema slučaj default onda dodati još 1



## L6

**ANATOMIJA GREŠKE**

Kada programeri pokušavaju da pronadju šta je pogrešno u njihovom programu, da pronadju grešku u programu, to se zove **debugging, ’isterivanje buba’**

Ima tri kategorije greške u programu,

• **sintaksne greške**, syntax errors,koje nastaju pogrešnim pisanjem komandi, npr PPRINT umesto PRINT, ili zaboravljanjem tačke-zareza, ;, • **izvršne grešk**e, run-time errors, gde greška nastaje ako program naidje na nešto neočekivano, npr. ako korisnik ukuca kao ulazni podatak neki negativan broj umesto pozitivnog broja koji označava godište neke osobe,

• **logičke greške**, logic-errors, gde greške nastaju ako program radi ali ne radi ono što se očekuje od njega, pa proizvodi nepredvidljive rezultate

Izvršne greške se otkrivaju tako što se program uporno testira na najraznovrsnije ulazne podatke.

dva glavna načina da se pronadju logičke greške su:

• stepping, koračanje kroz program,

• i watching, razgledanje prrograma.

**Stepping** predstavlja izvršavanje programa linija po linija, i posmatranjem kako program radi. U momentu kada se otkrije da program radi nešto pogrešno, vi onda znate tačno koja linija u programu pravi grešku.

Npr. za koračanje kroz neki program u Liberty BASIC, sledeće instrukcije se koriste:

• Debug,

• Step Into,

• Step Out, Itd.

**Debager**" tj.debugger je alat koji omogućuje da se program izvršava na jedan usporen način ("slow motion"), instrukcija po instrukciju, tako da omogućuje da se ispita izvršavanje programa a u cilju otkrivanja tj. detekcije i dijagnoze greški u programu. Kod manjih programa mogu se ubaciti printujuće komande, printing commands,

Debugger moze uraditi sledece:

• izvršavanje programa na kontrolisan/usporen način

• zaustavljanje programa u specificiranim tačkama

• ispitivanje vrednosti varijabli i promena vrednosti varijabli kod zaustavljanja programa

Kod nekih projekata dedefektovanje nosi 50% od ukupnog vremena razvoja softvera.

**Neefikasni prilazi dijagnozi greške su**

• pronalaženje greške pogadjanjem, pomoću slučajno tj haotično izabranih instrukcija štampanja u programu, i pomoću slučajnih tj haotičnih izmena u programu

• izbegavanje da se udubi duboko u problem

• već se površno pokušava ispraviti problem

**Stabilizacija** greške zahteva testiranje, i to traženje test-slučaja koji ne samo da proizvodi tu grešku, već sužavanje opsega test-slučajeva, tj utvrdjivanje opsega koji izaziva grešku

Tehnike za nalaženje greški su sledeće:

• Koristiti sve raspoložive podatrke da se napravi hipoteza o grešci

• Profiniti test-slučajeve koji proizvode grešku

• Ispitivati pojedine jedinice izolovano od ostatka programa

• Integraciju obaviti postepeno, u malim koracima

• Proveriti tipične greške

• Napraviti pauzu u traženju greške, da bi se slegli utisci

• Definisati vreme raspoloživo pre primene krajnjih mera

• Napraviti listu krajnjih mera

Primenom programa za dijagnozu greške, programa debugger, i tehnike koračanja, pauztiranja, i posmatranja, mogu otkriti uzrok neke greške, i zatim ova greška ispraviti.

Koristi se nekoliko načina da se identifikuje greška:

• **Stepping**, tj. ‘koračanje’ kroz program, gde debugger izvršava vaš program liniju po liniju, tako da se može locirati gde je greška

• **Breakpoints,** pauziranje programa u pojedinim tačkama, znači umesto da se ceo program ispituje linija po linija, ovde se definišu zaustavne tačke, pa se onda može ispitivati linija po linija, a ostatak programa se preskače kod ispitivanja

• **Watching**, **posmatranje,** koje omogfućuje da se ispita koje podatke memoriše vaš program u memoriji, tj. da li memoriše očekivane podatke

Build Tools: Cilj ovog alata je da minimizira vreme potrebno da se izgradi program koristeći trenutne verzije izvornih programskih fajlova.

**Kompajleri** (eng.compilers) konvertuju izvorni kod u izvršni kod

Ne treba totalno verovati kompajleru o lokaciji sintaksne greške.

Detektori sintaksnih greški: Kompajleri

Kompajleri su od velike koristi kod otkrivanja sintaksnih greški.

Medjutim ne treba totalno verovati kompajleru o lokaciji sintaksne greške, jer kompajleri znaju da pogreše i da netačno lociraju liniju programa gde je problematična sintaksna greška

##### Dijagnostički alat:

• ‘**Skele za program’** tj. Scaffolding Kao što je prethodno napomenuto, kod nalaženja defekta, nekad je neophodno , Izvući problematični deo izvornog koda, napisati dodatni kod za testiranje ovog problematičnog koda, tzv. ‘skele za program’, i onda egzekucija tog test-programa.

• **Okviri za testiranje** tj. Test frameworks- pomažu kod testiranja pomoću skela, naime test-frameworks alati omogućuju laku izradu skela i upotrebu tih skela za testiranje. Npr. alat JUnit,CppUnit, NUnit, itd.

• **Komparatori izvornog koda** (Source-code comparators):

Komparatori izvornog koda su korisni kod modifikacija programa u cilju korigovanja greške. Naime nekada je potrebno uporediti modifikovan program i početni program, da bi se neka promena preispitala ili izbacila, jer npr kod korigovanja programa, često se prave nove greške u novoj verziji programa, i ovi komparatori omogućuju pomažu kod detekcije novih greški, jer pomažu osvežavanju memorije programera

### L7

##### INTEGRACIJA SOFTVERA

Može doći do kolapsa u toku konstruisanja softvera, naime broj defekata može da izgleda nesavladljiv, napredak u integraciji može da bude nevidljiv.

**Integracija softvera je integracija softverskih jedinica/komponenti u jednu funkcionalnu tj operativnu celinu.**

Ako se izkonstruiše i integriše softver po pogrešnom redosledu, to ima negativne posledice: otežava kodiranje, otežava testiranje, otežava prepravke ( tj tzv debugging)

FAZNA INTEGRACIJA :

Problem sa faznom integracijom je da kad se klase integrišu da funkcionišu zajedno po prvi put, novi problemi se pojavljuju, i oni mogu biti bilo gde.

Do nedavno, fazna integracija je bila obavezna. Sastoji se od sledećih ustanovljenih faza:

• dizajnirati, kodirati, TESTIRATI, dedefektovati (debugging) svaku klasu, što se inače zove razvoj jedinica softvera tj unit development

• kombinovati klase u jedan sistem, što se zove sistemska integracija

• testirati i dedefektovati ceo sistem

Medjutim , problem sa faznom integracijom je da kad se klase integrišu da funkcionišu zajedno po prvi put, novi problemi se pojavljuju, i oni mogu biti bilo gde. Pošto se tipično radi o velikom broju klasa koje nikad nisu radile zajedno pre toga, uzroci mogu biti različiti, npr:

Nedovoljno testirana klasa, ili greška u interfejsu izmedju dve klase, ili greška usled interakcije dve klase. Medjutim, sve klase su sumnljive.

Zbog svega ovoga ovakva integracija se zove ‘ eksplozivna integracija’. Problemi mogu biti :Slabo sakrivanje podataka, Globalne varijable, Nedokumentovani interfejsi, Pretpostavke o operisanju greškama tj zaštita od grešaka

**INKREMENTALNA INTEGRACIJA :**

Kod inkrementalne integracije razvije se jedan manji deo sistema koji može da obavlja neku funkciju, npr najmanji funkcionalni deo, i zatim se taj deo testira i dedefektuje.

Inkrementalna integracija: Kod inkrementalne tj postepene integracije, ne kombinuju se svi delovi programa odjednom, već se ide postepeno, i to na sledeći način:

• Razvije se jedan manji deo sistema koji može da obavlja neku funkciju, npr najmanji funkcionalnideo, najteži deo, ključni deo, ili kombinacija ovih, i zatim se taj deo testira i dedefektuje (tj vrše popravke), i ovo će poslužiti kao jezgro tj kostur na koji se onda kače ostali delovi

• Dizajnirati, kodirati, testirati i debug-ovati novu klasu koja ne pripada prethodnom jezgru

• Integrisati novu klasu sa prethodnim jezgrom, i testirati i debug-ovati kombinaciju jezgra i nove klase, i osigurati da ova kombinacija funkcioniše pre nego se dodaju nove klase

PREDNOSTI INKREMENTALNE INTEGRACIJE

Lakša detekcija greški, jer greška je obično ili u novoj komponenti (interakciji ove komponente sa ostatkom sistema), ili je greška u vezi tj interfejsu izmedju nove komponente i ostatka progra

Kod inkrementalne integracije potrebno je odrediti redosled i napraviti vremenski plan integracije, i ovo ima uticaja na konstruisanje celog programa.

Ima nekoliko strategija integracije:

• integracija odozgo-nadole,top-down integration,

• integracija odozdo-nagore, bottom-up integration,

• sendvič-integacija, sandwitch-integation,

• ’antirizična’ integracija, risk-oriented integration,

• ’funkcionalna’ integracija, feature-oriented,

• T-integracija, T-shaped integration

INTEGRACIJA ODOZGO-NADOLE:

Konačno , ne može se primeniti integracija odozgo-nadole ako ne postoji klasa na vrhu. U mnogim sistemima, ne postoji vrh, tj struktura softvera nije kao na slici.

Kod integracije odozgo-nadole, interfejsi izmedju klasa moraju biti korektni. Većina greški nisu greške koje su unutar klase već one koje su usled pogrešne interakcije klasa. Ako su interfejsi tačno specificirani, mnogo je manje problema. Specifikacija interfejsa je uradjena ranije, kod dizajna i kodiranja, ali provera interfejsa kod integracije je veoma korisna.

INTEGRACIJA ODOZDO-NAGORE

Glavni problem integracije odozdo-nagore je da se glavne tj najvažnije klase i njihovi interfejsi, tj one klase i interfejsi na vrhu hijerarhije, ostavljaju za kraj integracije.

ANTIRIZIĆNA INTEGRACIJA

Kod antirizične integracije identifikuje se stepen rizika za svaku klasu. I onda se vrši prvo implemenacija ovih delova.Slika ilustruje antirizićnu integraciju.

"FUNKCIONALNA INTEGRACIJA"

Kod osobinske integracije, integracija se obavlja, tako što se klase dodaju u grupama koje formiraju odredjenu sobinu sistema. Slika prikazuje funkcionalnu tj „osobinsku“

SENDVIČ-INTEGRACIJA

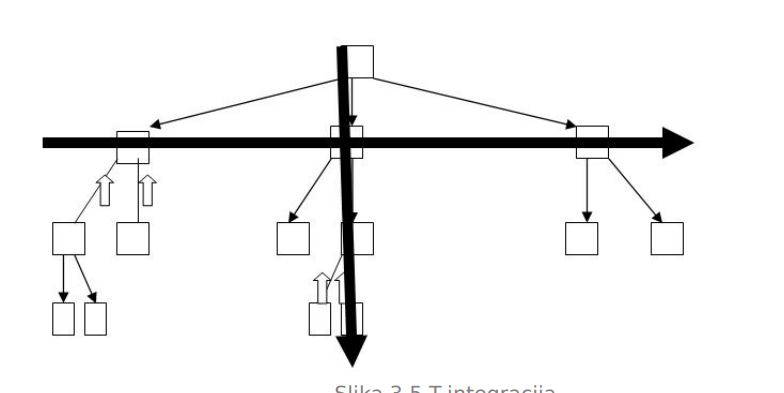
Pošto rigidna integracija odozdo-nagore ili rigidno odozgo-nadole nisu preporučljive, moguća je i tzv. sendvič integracija, gde se prvo ide po najvišem nivou, pa zatim najnižem, pa tek onda srednjim nivoima. Tu postoji niz varijacija, npr. da kod kretanja po najnižem nivou se neke manje važne klase preskoče, pa da se integrišu naknadno kod kretanja po višem nivou, što se može nazvati naizmenična integracija

NAIZMENIĆNA INTEGRACIJA

vidimo da se u okviru naizmenične integracije vrše obilasci i integracija klasa ili modula na različitim nivoima, uz mogućnost preskakanja nekih i kasnijeg njihovog naknadnog testiranja i integrisanja, u zavisnosti od konkretnog problema koji se rešava. U toku ovakvog alternativnog tipa integracije se takođe eliminišu uočeni problemi.

T –INTEGRACIJA

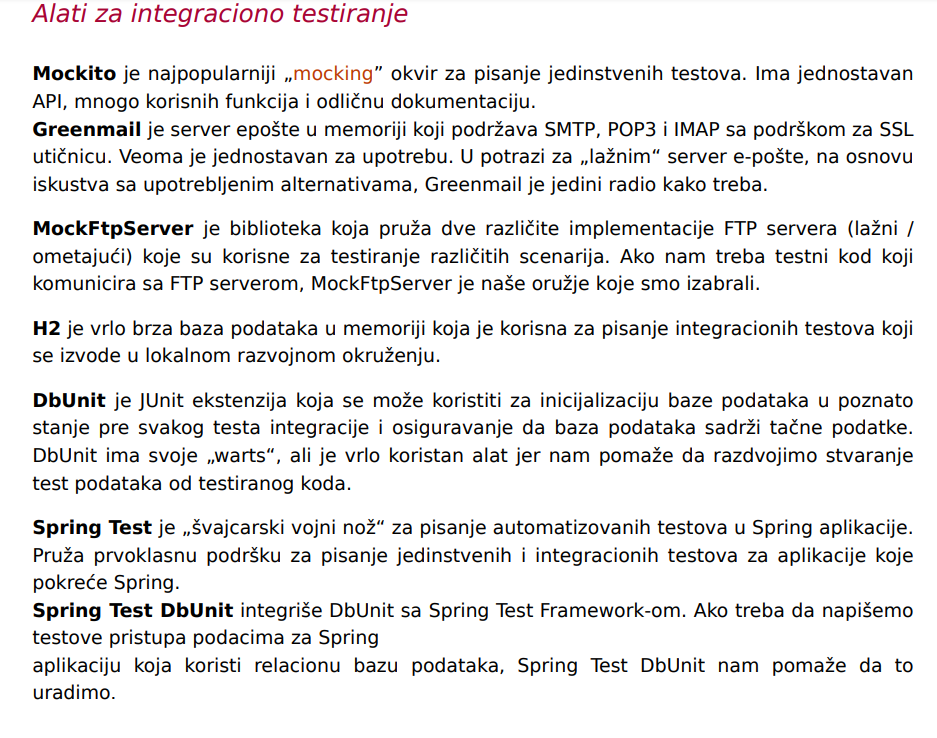
Kod ovog tipa integracije, jedan odredjeni vertikalni deo je izdvojen u cilju brzog razvoja i integracije. Ovaj deo se bira tako da može da ilustruje sistem i razvoj sistema od početka do kraja, tako da može da posluži za detekciju glavnih problema i proveru pretpostavki napravljenih kod dizajna. Pošto se izabrani vertikalni deo završi sa implementacijom, i svi uočeni problemi eliminišu, onda se ide u širinu i samo glavni delovi tj delovi na vrhu se razviju, odnosno glavna horizontala (horizontala na vrhu), kao što je ilustrovano na donjoj slici, dajući tako okvir za razvoj ostatka funkcionalnosti sistema. Ovaj prilaz se često kombinuje sa funkcionalnom integracijom i antirizičnom integracijom.



###### ŠTA JE TO INTEGRACIONO TESTIRANJE?

**Integraciono testiranje** uključuje pisanje "code snippets" za potrebe testiranja integhrisanih modula, pa je to u stvari white box testing. Ali, može se koristiti i black-box testiranje.

Postoje dva osnovna prilaza da bi se izvršilo integraciono testiranje: • Bottom up approach tj. odozdo-nagore • Top down approach. tj. odozgo-nadole



POREĐENJE INTEGRACIONOG TESTIRANJA SA JEDINIČNIM I FUNKCIONALNIM TESTIRANJEM

Ovde poredimo integraciono testiranje sa jediničnim testiranjem i funkcionalnim testiranjem.

Evo jednog jednostavnog primera ove tri vrste testiranja. Npr. za neki mobilni telefon, glavni delovi su 1)baterija i 2)SIM kartica.

**Jedinično testiranje - Unit testing** – baterija se npr. testira da li ima dovoljan kapacitet i da li je dovoljno sveža, itd.. A SIM kartica se proveri da li omogućuje aktivaciju. integraciono testiranje - **Integration Testing**– baterija i SIM kartica se ubacuju u telefon, i proverava da li telefon radi.

**Funkcionalno testiranje - Functional Testing**– funkcionalnost mobilnog telefona se detaljno proverava u smislu pojedinih opcija telefona, a između ostalog i upotreba baterije i SIM kartice u realnim uslovima upotrebe telefona.

###### L8

TEŠKOĆE KOD DIZAJNA

Dizajn je klizavi proces, jer se u tom procesu prave mnoge greške, koje se kasnije ispravljaju.

HIJERARHIJSKA STRUKTURA SOFTVERA:

Dizajn se obavlja na nekoliko različitih nivoa detaljnosti/abstrakcije.

Dizajn je potrebno obavljati na nekoliko različitih nivoa, različitih nivoa detaljnosti. Donja slika prikazuje te različite nivoe. Nivoi su sledeći:

• Softverski sistem

• Podela na podsisteme/pakete

• Podela na klase u okviru paketa

• Podela na podatke i podprograme u okviru klase

• Interni dizajn podprograma

**Nivo 1- Softverski system (sistemski nivo)** :

Ovo je nivo celokupnog sistema. Mnogi programeri sa ovog nivoa odmah idu na nivo 3 tj. dizajniranje klasa, i kasnije možda dizajniraju nivo 2, tj. Deljenje sistema na podsisteme/pakete. Medjutim, obično, korisno je ipak razmišljati o podeli na podsisteme

**Nivo 2, Podela na podsisteme :**Na ovom nivou dizajna potrebno je identifikovati podsisteme, tj. odlučiti kako podeliti sistem na podsisteme i kako da ovi podsistemi komuniciraju.

Od posebne važnosti je ovde pratiti pravila kako treba da podsistemi komuniciraju izmedju sebe. Ako svaki podsistem komunicira sa svim ostalim podsistemima onda nema koristi od podele na podsisteme.

Preko **restrikcije tj. minimizacije komunikacije** izmedju podsistema dolazi se do optimalne podele na podsisteme. Recimo da podelimo system na podsisteme, kao što je dole prikazano. Ako svaki sistem komunicira sa svakim preosalim podsistemom imamo slliku sa mnogo komunikacija. Medjutim potrebno je minimizirati komuniciranje medju podsistemima. Ako se koristi princip minimizacije/restrikcije komunikacije izmedju pojedinih podsistema onda se dobije redukovano komuniciranje izmedju podsistema kao što je ilustrovano dole.

**Nivo 3 – Podela na klase** :Dizajn na ovom nivou obuhvata identifikaciju svih klasa sistema, tj. podelu sistema/podsistema na klase. Ovo je kritična faza dizajna.

Takodje se specificiraju interfejsovi medju klasama, tj. Interfejsovi svake klase u odnosu na druge klase. Znači definiše se kako interaktuju klase izmedju sebe.

Glavna aktivnost dizajna ovde je da se za svaku individualnu klasu koja je formirana da se na sledećem nivou, tj. niovou 4, omogući individualni dizajn klase. Za svaku klasu se specificira odredjeni početni broj atributa, koji je dovoljan da se kasnije ide na individualni dizajn klase. Važno je praviti razliku izmedju objekata i klasa. Jedan objekat je bilo koji konkretni entitet koji postoji u programu u toku operisanja programa. Klasa je statična kategorija, dok je objekat dinamička kategorija, sa konkretnim vrednostima datim atributima

**Nivo 4 - Podela na podprograme i podatke,** individualni dizajn klase :Dizajn na ovom nivou obuhvata podelu svake klase na podprograme, i podatke. Interfejsi medju klasama specificirani na nivou 3 će odrediti neke od podprograma. Dizajn na nivou 4 će specificirati listu podprograma za dotičnu klasu. Ovde se mogu pojaviti hijerarhijski povezani podprogrami. Definisanje podprograma koji pripadaju jednoj klasi obično zahteva da se preispitaju interfejsi medju klasama, tj. vraćanje na Nivo 3. Nivo 4, tj. dekompozicija i dizajn na ovom nivou, je obično dodeljena programeru koji vrši kodiranje.

**Nivo 5, Interni dizajn podprograma :** Dizajn na nivou podprograma analizira detaljno kako funkcioniše konkretni podprogram, i to radi individualni programmer. Obično se to sastoji od pisanja pseudokoda, pretraživanje biblioteka algoritama, organizacija paragrafa koda, i kodiranja.

**FAZE DIZAJNA?**

Sledeće su faze softverskog dizajna: arhitektonski, dizajn podataka, interfejsni dizajn i proceduralni. Dizajn se sastoji od sledećih faza:

**• Arhitektonski dizajn** (Architectural design) – Ova faza proizvodi strukturne jedinice (skup klasa)

**• Dizajn podataka** (Data design) - Ova faza rezultira u strukture podataka (definiše listu tj. skup podataka za svaku klasu).

**• Interfejsni dizajn** (Interface design) – Ova faza specificira interfejse (skup ulaznih i izlaznih podataka) izmedju strukturnih jedinica.

**• Proceduralni dizajn** (Procedural design) – Ova faza specificira algoritme za svaki metod tj funkciju (klasne metode).

###### O.O. ANALIZA I DIZAJN

Izlazi iz OO analize nisu do kraja razvijeni kada se koriste kod OO dizajna, što znači da se OO analiza i OO dizajn obavljaju paralelno, iterativno i u medjusobnoj vezi.

**Sledeći su izlazi OO analize tj. ulazi OO dizajna:**

• Konceptualni model je rezultat OO analize i on opisuje objekte u problemskom domenu

• Use-case modelovanje

• Sistemski sekvencijalni dijagrami, koji za različite use-case scenarije prikazuju dogadjaje (i redosled dogadjaja) koje spoljni akteri proizvode

OO (objektno-orijentisan) dizajn je jedan pristup softverskom dizajnu.U ovom pristupu softverski system se planira tj. projektuje kao skup objekata koji su u medjusobnoj interakciji.

**A sledeći su izlazi (izlazni rezultati) OO dizajna:**

• Sekvencijalni dijagrami, gde se sistemski sekvencijalni dijagrami pretvaraju (proširuju) da bi se dodali objekti koji upravljaju dogadjajima u sistemu (sekvencijalni dijagrami se sastoje od vertikalnih linija koje prikazuju objekte a horizontalne linije predstavljaju poruke izmedju objekata I to po redosledu po kom nastaju)

• Klasni dijagram koji prikazuje statičku strukturu sistema

Postoji čitav niz metoda softverskog dizajna baziranih na objektima (klasama).

U moderno vreme, na velikim projektima, glavnu ulogu igraju

• Naslednost klasa

• Polimorfizam funkcija

**Detaljni dizajn** je dizajn pojedinih delova softvera npr. moduli/ klase (the design of low-level modules/classes), i on uključuje:

• dizajn podataka,

• dizajn interfejsa, i

• dizajn algoritama tj. klasnih metoda (proceduralni dizajn).

**DOKUMENTACIJA DIZAJNA**

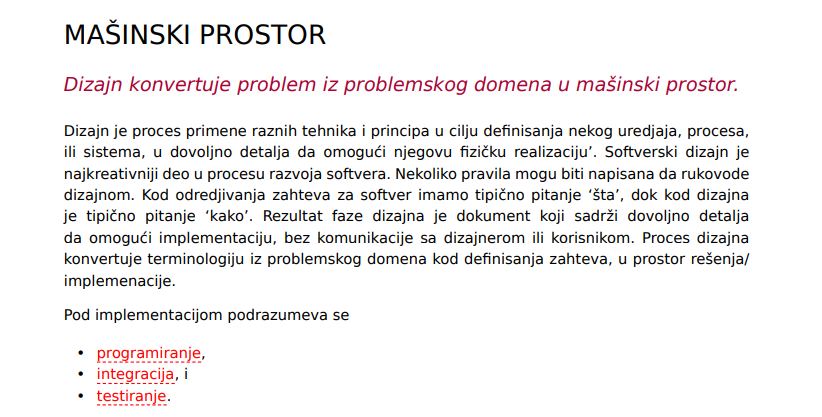
Kod objektno-orijentisanog razvojasoftvera, razvija se objektni tj. klasni model, gde u fazi zahteva se identifikuju objekti tj. klase relevantni za fazu zahteva (tzv. "problemski domen”) tj. relevantni za korisnika softvera, i tada je cilj modelovanje problemskog domena a ne dizajna (arhitektonskog dizajna i detaljnog dizajna) ili implementacije. Modeluju se entiteti koji su bitni za problemski domen a ne entiteti (klase) koji su bitni za rešenje problema (dizajn, implementacija).

Dakle, softverski razvojni cilkus obuhvata faze : faza zahteva, faza dizajna i faza implementacije.

**KOMPONENTNI DIZAJN**

Komponentni dizajn, ili proceduralni dizajn, se obavlja posle dizajna arhitekture i dizajna podataka i dizajna interfejsa.

Krajnji cilj je da se model dizajna prevede u operativni softver. Softverski detaljni dizajn, obuhvata komponentni tj. proceduralni dizajn, kao i dizajn podataka, i dizajn intefejsa. Neki autori poistovećuju detaljni i komponentni dizajn. Nivo abstrakcije modela komponentnog dizajna je relativno visok, u poredjenju sa nivoom abstrakcije operativnog softvera tj. izvornog koda, koji je nizak. Ovaj razmak može biti velik i izazovan i težak da se prevazidje. Ako se ne obavi kvalitetno ovaj komponentni dizajn, mogu se uvesti greške u softver koje je kasnije teško pronaći. Vrlo je važan princip da je mnogo bolje izbegavati greške nego ih posle tražiti. Dobri i efikasni programeri su oni koji minimiziraju pravljenje grešaka u fazi pravljenja softvera, i ne troše vreme na eliminaciju nepotrebnih grešaka. U tom smislu je komponentni tj. proceduralni dizajn veoma važno orudje za eliminaciju grešaka u početnoj fazi pravljenja softvera.



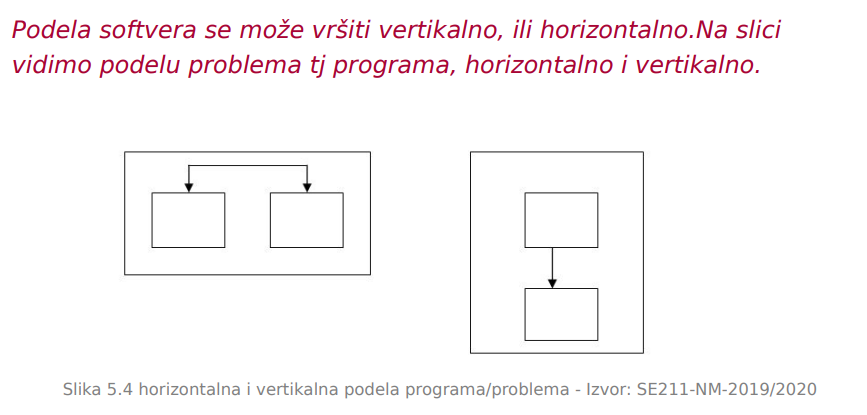
**Ciljevi dizajna obuhvataju, definisanje klasa i njhovih atributa, definisanje metoda klasa, definisanje relacija izmedju klasa, odredjivanje nasledje klasa, podela na podsisteme, itd.**

ITERACIONA TEHNIKA

Dizajn je iterativni proces. Slikovito opisano, ne ide se od A do B jedanput, već se posle dolaska u B vraćate ponovo u A, da bi ponovo išli ka B.

MODULARNA TEHNIKA

Podela softvera se može vršiti vertikalno, ili horizontalno.Na slici vidimo podelu problema tj programa, horizontalno i vertikalno.



Modularnost ( Modularity )– Ovo je strukturni prilaz gde se softver deli u manje delove, pri tome se svi delovi integrišu da bi se postigli zahtevi za softver. Modularna tehnika: Problem je potrebno podeliti

• na različite delove,

• i na različite aspekte

• i onda pojedine delove analizirati i dizajnirati.

• I zatim, iterirati.

#### Reupotreba softvera ima dva lica:

• **“construction for reuse”** , Konstruisanje za reupotrebu • i**“construction with reuse.”** konstruisanje sa reupotrebom

• prvi vid znači kreiranje reupotrebljivih softverskih resursa, a drugi vid je reupotreba softverskih resursa kod konstruisanja softvera

• reupotreba često podrazumeva prelazak granica između projekata i kompanija, dakle reupotrebljivi resursi se mogu koristiti u više projekata i više kompanija

**Do reupotrebe retko dolazi spontano, već je potrebno izvršiti identifikaciju reupotrebe softvera**

**Reupotreba softvera je jedan od velikih potencijala o.o. softvera.** Međutim, do reupotrebe retko dolazi spontano, već je potrebno izvršiti identifikaciju reupotrebe softvera. Prvi korak je nešto što se zove:

domain analysis tj. domenska analiza

UNS PODSISTEMI

Sistem je izdeljen na podsisteme, odnosno na samostalne softverske jedinice.

# L9

1. **O.( OBJEKTNO ORJENTISANI ) SOFTVERSKI DIZAJN**

**Softverski dizajn je nedeterministička primena skupa heurističkih pravila.**

o. softver se razlikuje od konvencionalnog tj. tradicionalnog softvera, i on nudi neke pogodnosti, npr. pojednstavljene zahteve, dizajn, implementaciju složenih sistema.

o.o razvoj se sastoji od:

• Modelovanja problemskog domena preko objekata koji predstavljaju relevantne entitete

• Kapsulizacija (encapsulating) funkcija sa podacima

• Rekorišćenjem objekata u okviru jednog softvera, ili više softvera

• Minimiziranjem razlike izmedju problemskog domena i softverkog rešenja problema.

• UML je standardna notacija/jezik za o. o. modele. Specifikacija UML-a se može naći na Internetu, npr. www.omg.org, ili pretraživanjem koristeći skraćenicu ‘UML

ITERATIVNI DIZAJN KLASE

**Interni dizajn klase** je iterativni/ponavljajući process, gde se u više navrata, pomoću principa profinjavanja, idući odozgo na dole (ili pak odozdo na gore), kompletira dizajn klase.

**Da bi se postigla optimalna organizacija softvera može se proces dizajna iterirati na dva načina,**

• jedan način je da se iterira organizacija klasa tj. podela na klase (top-level organization of classes),

• a drugi način je da se iterira na nivou svake klase, da se poveća nivo detaljnosti za svaku klasu.

**Sledeći koraci se koriste kod dizajna objekata:**

• Identifikacija objekata i njihovih atributa (podataka)

• Odrediti sve operacije tj. metode objekata koje se obavljaju na sopstvenim objektima

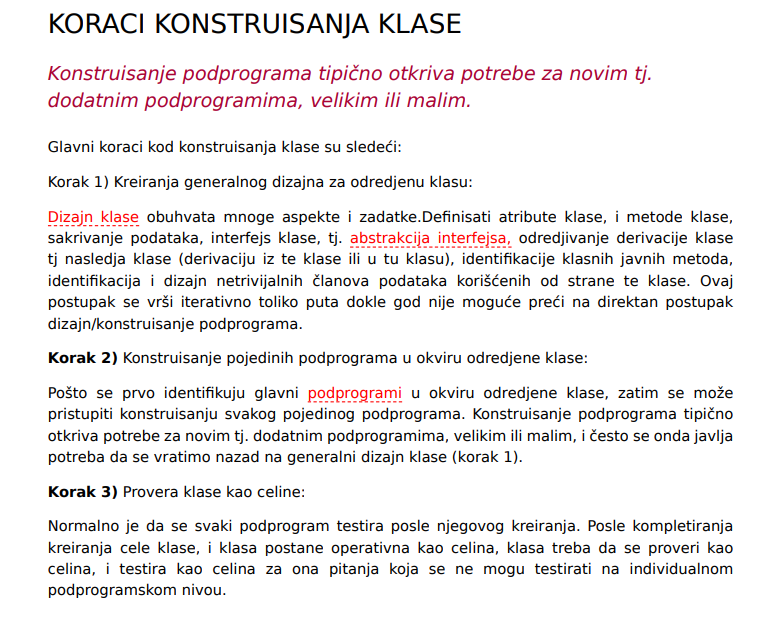
• Identifikovati interakcije objekata,

• Definisati koji deo objekta se prikriva, tj. koji deo objekta je javan, a koji skriven (vidljivost objekta)

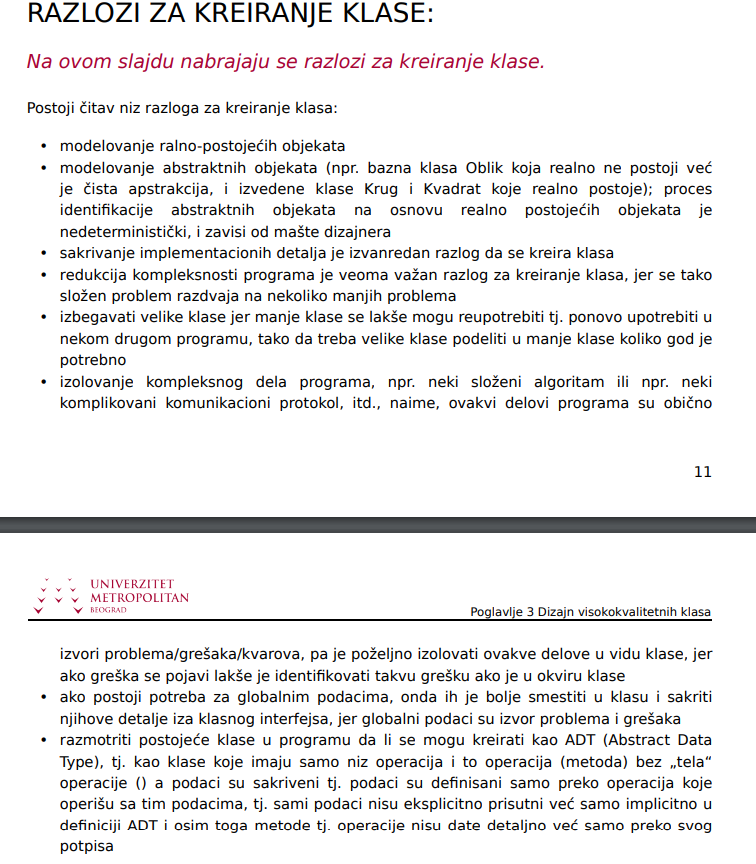
• Detaljno definisati interfejse pojedinih objekata ( tzv. javne interfejse, public interfaces)

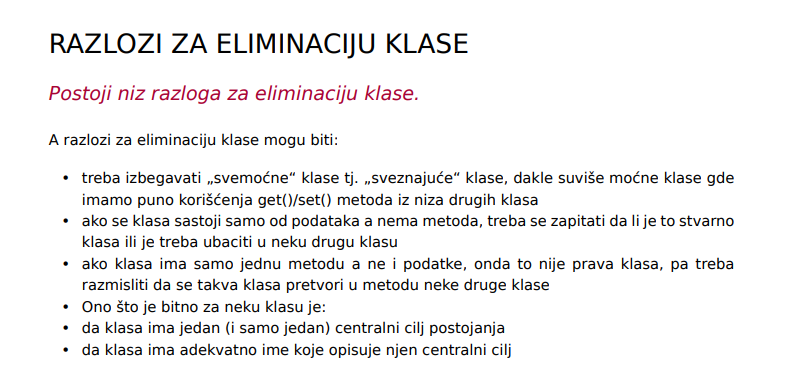
• Identifikovati nasledje objekata tj. klasa, i tzv. zaštićene interfejse

**U toku faze analize i faze dizajna softvera, mogu se identifikovati softverski objekti (klase) koji se razlikuju od realno postojećih objekata, ali realno postojeći objekti su dobra početna tačka.**



KLASNA RELACIJA „IMATI”: Neko praktično pravilo je da ako klasa ima više od oko 7 (7±2) članova-atributa, treba razmotriti da se klasa dekomponuje u nekoliko manjih klasa.





IMPLEMENTACIJA (MAPIRANJE) KLASNOG I SEKVENCIJALNIH DIJAGRAMA Dakle, klasni dijagram i sekvencijalni dijagram se koriste kao mape po kojima mogu da se onda pišu kodirajuće instrukcije, da još detaljnije opišu sistem koji razvijamo.

**Implementacioni klasni dijagram** prikazuje sve implementacione detalje, a to su : vidljivost , tip i početne vrednosti atributa, parametre metoda, sa tipovima parametara, kao i tip povratnih vrednosti metoda. Na osnovu implementacionog klasnog dijagrama kodiraju se direktno klase, atributi, metode, relacije izmedju klasa, kao što je gore ilustrovano.

Sekvencijalni dijagram se koristi :

• Da odredi redosled programskih instrukcija

• da se implementiraju razni use-case scenariji.

DETALJNI DIZAJN ASOCIJACIJA: Kod dvosmerne asocijacije klasa, obe klase imaju atribute koje označavaju onu drugu klasu tj. klasu sa kojom žele da razmenjuju poruke.

DETALJNI DIZAJN OPERACIJA Tokom detaljnog dizajna operacija, vrši se dizajn algoritama koji će se koristiti kod pojedinih operacija.

KLASNI DIJAGRAM Kod o.o. razvoja softvera, i definicija problema i rešenje problema se izražava pomoću tzv. objekata. Medjutim, kod rešenja problema, teži se da objekti prerastu u tzv. klase.

Klasni dijagram može biti konceptualan, zatim može dalje biti razvijen u fazi analize, ili još dalje rezvijen u dizajn fazi(arhitektonski dizajn i detaljni dizajn faze) za potrebe implementacije.

###### L9

### Dizajn i konstruisanje podprograma

Kod kreiranja ovih potprograma korisno je primeniti sistematski prilaz tj. primeniti tehniku dizajna podprograma, provera dizajna, pisanje (kodiranje) podprograma, i provera izvornog koda.

**Postupak izgradnje (konstruisanja) potprograma:**

Mnogi potprogrami u okviru odredjene klase su potprogrami za pristup sopstvenoj klasi (get/set metode), koji se mogu jednostavno i direktno implementirati, kao i pristup ostalim klasama. Medjutim, ostale potprograme je komplikovanije implementirati (napisati izvorni kod), i kod kreiranja ovih potprograma korisno je primeniti sistematski prilaz tj. primeniti tehniku dizajna potprograma, provera dizajna, pisanje (kodiranje) potprograma, i provera izvornog koda. Koraci sadržani kod kreiranja podprograma su prikazani na donjoj slici.

**Konstruisanje potprograma klase (routine) sastoji se od:**

• Dizajna potprograma

• Kodiranja potprograma

• Provera potprograma

• Ponoviti prethodne korake po potrebi

VISOKI DIZAJN POTPROGRAMA

Prvi korak dizajna potprograma obuhvata sledeće:

Definisati/iskazati problem koji će dotični potprogram da rešava, i to sa dovoljno detalja da se omogući kreiranje potprograma.

Ovaj posao je možda već obuhvaćen dizajnom na visokom nivou (high-level design)**,i ovaj visoki dizajn opisuje sledeće:**

• Informacije koje će se sakrivati u potprogramu

• Ulazi u i izlazi iz potprograma

• Preduslovi potprograma , npr. fajlovi otvoreni ili zatvoreni, varijable inicijalizovane ili u zadatim opsezima, itd.

• Postuslovi potprograma, npr. fajlovi otvoreni ili zatvoreni, varijable inicijalizovane ili u zadatim opsezima, itd.

**DIZAJN POTPROGRAMA POMOĆU PPP**

Dizajn potprograma obuhvata profinjavanje (pisanje pseudokoda na niskom nivou apstrakcije), počev od pseudokoda na visokom nivou apstrakcije i spuštati se postepeno na najniži nivo.

Dizajn potprograma obuhvata:

• Imenovanje potprograma

• Plan za testiranje potprograma

• Pretražiti standardne softverske biblioteke u cilju korišćenja već raspoloživih potprograma ili delova potprograma

• Predvideti moguće greške u programu

• Razmisliti o efikasnosti tj. optimizaciji programa • Pretražiti raspoložive algoritme

• Pisanje pseudokoda na visokom nivou abstrakcije, tj bez ulaženja u detalje

• Dizajn podataka tj. razmisliti koje tipove podataka koristiti u programu, npr. arrays, itd.

• Proveriti prethodni pseudokod (na visokom nivou apstrakcije), i zatim dati još nekom da proveri pseudokod

• Profinjavanje (pisanje pseudokoda na niskom nivou apstrakcije), počev od pseudokoda na visokom nivou apstrakcije i spuštati se postepeno na najniži nivo apstrakcije

• Iteriranje pseudokoda,tj. probati razne ideje za dizajn potprograma i ove ideje opisati u pseudokodu, počev od visokog nivoa apstrakcije pa do niskog nivoa apstrakcije

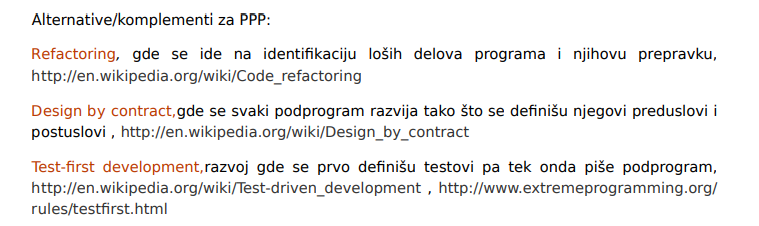
• Ići na kodiranje tj. programiranje (pisanje podprograma) tek kada je proces iteriranja pseudokoda gotov i sve ideje o dizajnu podprograma iscrpljene

PPP

PPP je verovatno najbolji metod za kreiranje klasa i podprograma, ali postoje i drugi pristupi koji se mogu koristiti kao alternative ili komplementi (dopune) za PPP.

Prethodni postupak dizajna/konstruisanja potprograma naziva se PPP (Pseudocode Programming Process), gde se vrši pisanje pseudokoda na visokomnivou apstrakcije i na niskom nivou apstrakcije i njegovo iteriranje.

PPP je verovatno najbolji metod za kreiranje klasa i potprograma, ali postoje i drugi pristupi koji se mogu koristiti kao alternative ili komplementi (dopune) za PPP.



### “REFAKTORISANJE PROGRAMA”(CODE REFACTORING):

Refaktorizacijom se poboljšava “čitljivost” programa (lakše razumevanje programa) i smanjuje kompleksnost programa, u cilju lakšeg održavanja i budućih promena programa (novih verzija).

###### POSTUPAK KONSTRUISANJA POTPROGRAMA

Pošto je završeno dizajniranje potprograma (routine), može se pristupiti konstruisanju u više koraka.

Ovaj dijagram se može varirati, menjajući redosled nekih blokova. Objašnjenje za dijagram:

• Ulaz u proces je pseudokod;

• Prvi blok odozgo: napisati deklaraciju podprograma

• Drugi blok: napisati prve i poslednje instrukcije, i prevesti dotični pseudokod u komentare

• Treći blok: popuniti code ispod svakog komentara (tj. Pseudokoda)

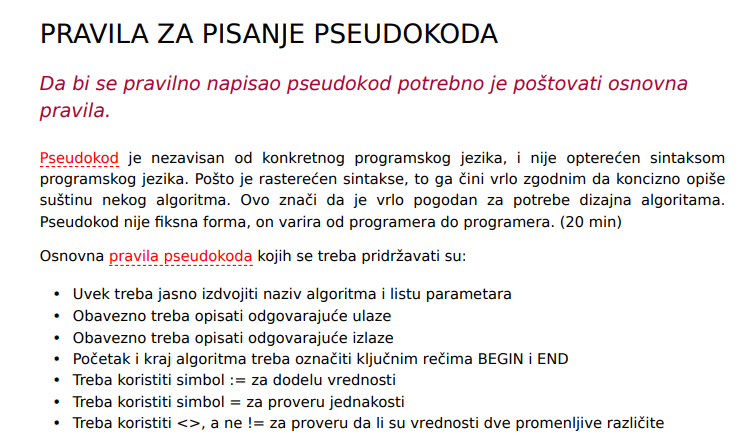
• Četvrti blok: proveriti programski code podprograma

• Peti blok: doterivanje podprograma

##### PROGRAM DESIGN LANGUAGE – PDL

Pseudokod se jednostavno sastoji od serije blokova teksta na Srpskom/ Engleskom, označenih serijskim brojevima, radi lakše organizacije programa.

Pravila pseudokoda



# L11

ATRIBUTI DIZAJNA –DEFINICIJA

Definišu se atributi dizajna: **kohezija, povezanost i abstrakcija**. Dizajn poseduje sledeće atribute:

**Apstrakcija (eng.Abstraction)**

• -Jedan predmet je abstraktan ako se nevažni detalji izostave. U umetnosti, abstrakcija pokušava da prikaže sliku sa samo nekoliko detalja.

• -Abstrakcija kod softverskog dizajna omogućava dizajneru npr da se fokusira na suštinska pitanja zanemarujući nepotrebne detalje nižeg nivoa (low-level details). Dobra apstrakcija krije nepotrebne detalje.

**Kohezija (eng.Cohesion)**

• Neki material je kohezivan ako se čvrsto drži u celini.

• Neka kompjuterska procedura je kohezivna ako svi delovi procedure su u vezi sa svakim izlazom.

• Klasa je kohezivna ako svi klasni atributi se koriste od strane svih klasnih metoda.

• Tj. u jednom modulu postignuta je kohezija, ako je sve unutar modula povezano medjusobom.

• Visoka kohezija (unutar nekog modula) je poželjna.

**Povezaost (eng.Coupling)**

• -Povezanost je mera interkonekcije izmedju modula.

• -Dva modula su povezana ako promena jedne varijable u jednom mdulu izaziva promenu u drugom modulu.

• -Obično, poželjna je najniža povezanost izmedju modula.

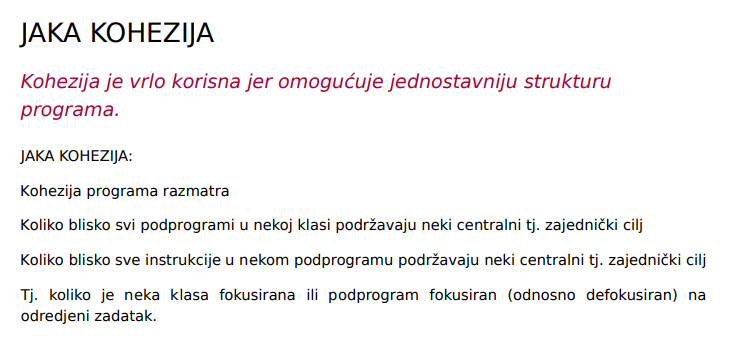
**Mogu se klasifikovati vrste kuplovanja**

**“Simple-data-parameter”** kuplovanje tj. **“jednostavno parametarsko” kuplovanje**: dva modula su “jednostavno-parametarski” kuplovana ako svi podaci koji se razmenjuju izmedju njih su prmitivni tipovi podataka (int, float, itd) i svi se razmenjuju tj. prenose preko liste parametra funkcije f(x1, …..,xN); ova vrsta kuplovanja je normalna i prihvatljiva

**“Simple-object” kuplovanje tj. “jednostavno-objektno” kuplovanje** označava povezanost modula sa nekim objektom gde modul “instancira” tj. “instantiates” taj objekt; ova povezanost je takodje prihvatljiva i normalna

**“Object-parameter” kuplovanje tj. “objektno-parametarsko”** kuplovanje gde su dva modula kuplovana na ovaj način ako Objekt1 zahteva od Objekta2 da mu preda Objekt3 (Objekt1 i Objekt2 razmenjuju podatke o Objekt3); ova vrsta kuplovanja je jača nego kad Objekt1 i Objekt2 razmenjuju primitivne tipove podataka jer zahteva da Objekt2 koristi Objekt3

**“Semantičko” kuplovanje**: ovo je kuplovanje izmedju dva modula gde jedan modul ne koristi neki sintaksni element drugog modula već neko “semantičko” znanje o dešavanju u drugom modulu, i ovo kuplovanje je opasno jer promena programa u korišćenom modulu može da razbije program u korisničkom modulu tako da se ne može detektovati pomoću kompajlera: npr.



Sledeće vrste kohezije su manje dobre tj. manje jake od funkcionalne kohezije:

**Sekvencijalna kohezija:** podprogram sadrži više operacija koje moraju biti izvršene po odredjenom redosledu i pri tome koriste zajedničke podatke od koraka do koraka (operacije su medjusobno povezane tj. proračun jedne operacije se koristi kod proračuna sledeće operacije)

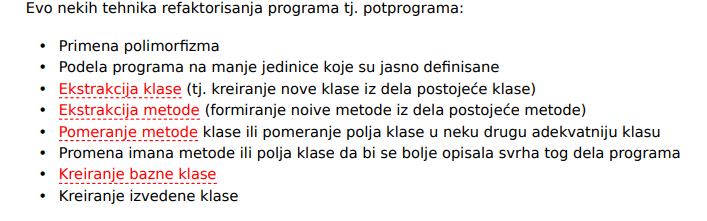
**Komunikaciona kohezija:** podprogram sadrži više operacija koje koriste iste podatke ali nisu povezane ni na kakav drugi način (operacije nisu u medjusobnoj vezi osim što koriste iste podatke)

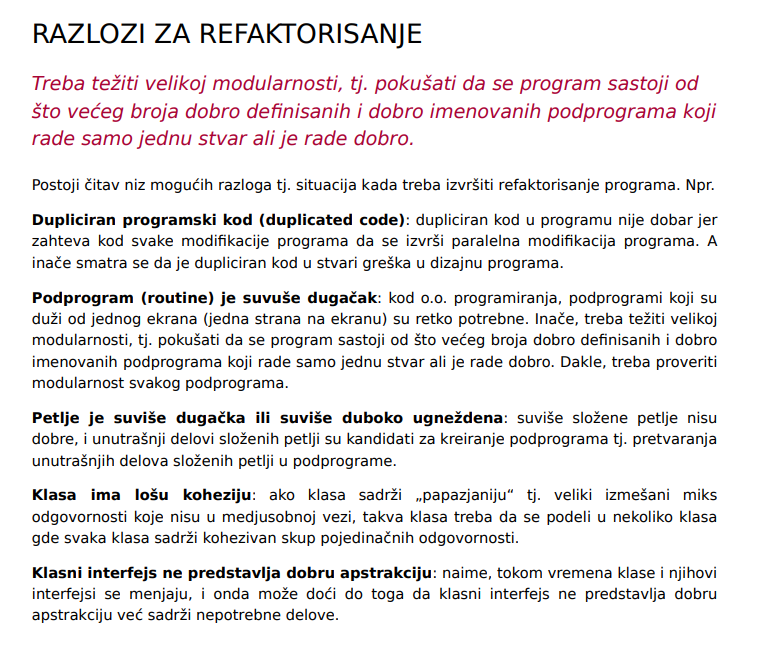
**Temporalna (temporal) kohezija:** više operacija se obavlja u isto vreme i to je razlog što su smeštene zajedno u isti podprogram

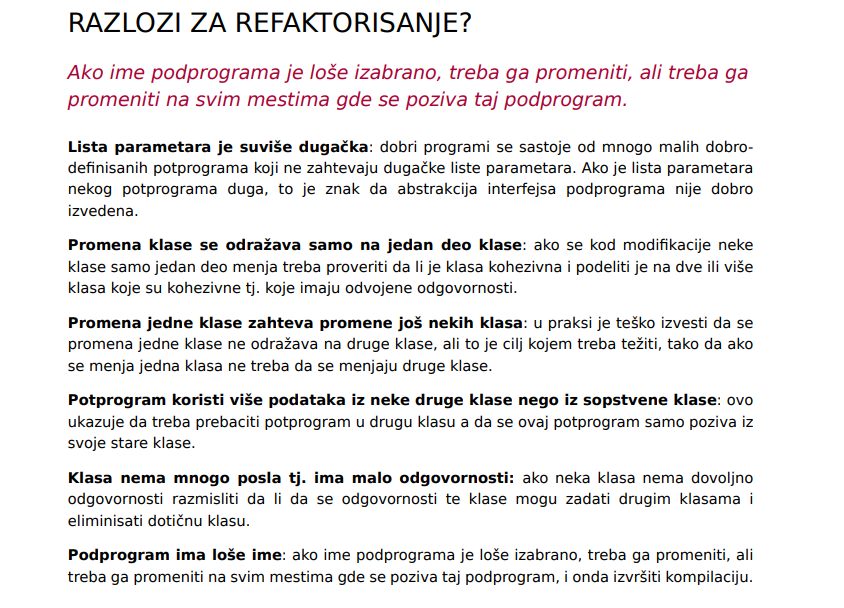
DOBRA APSTRAKCIJA:

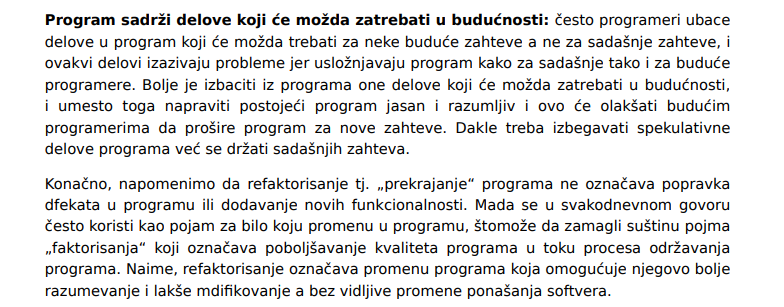
Dobar interfejs podprograma (funkcije) ili dobar interfejs podsistema tj. paketa je abstrakcija koja omogućuje da se fokusirate na interfejs i da ignorišete detalje.

**Refaktorisanje programa** tj. potprograma je proces/tehnika restruktuiranja postojećeg kompjuterskog programa je interna promena strukture programa bez promene njegovog spoljnjeg ponašanja. Tako se poboljšava “čitljivost” programa (lakše razumevanje programa) i smanjuje kompleksnost programa, u cilju lakšeg održavanja programa i budućih promena programa (novih verzija). To je postepeno poboljšanje dizajna programa, u cilju lakšeg rada sa tim programom.









**Postavlja se pitanje koje su moguće strategije refaktorisanja.**

**Postavlja se pitanje kada treba vršiti refaktorisanje, i evo odgovora:**

• Ako se doda potprogram u program, onda treba izvršiti refaktorisanje tako što se provere podprogrami koji su vezi sa dodatim potprogramom i izvršiti njihovo refaktorisanje

• Ako se doda klasa u program, onda treba izvršiti refaktorisanje tako što se refaktorišuklase koje su u vezi sa dodatom klasom

• Ako se eliminiše neki defekt u programu, treba izvršiti refaktorisanje tako što proveritiostatak programa na slične defekte

• Refaktorisati složene module

# L12

## SAMOPROVERE PROGRAMA

**Samoprovere programa** su deo programa, tj izv. kod, obično podprogram ili makro-funkcija, koji omogućuje program da sam sebe proverava u toku izvršavanja. Ako je neka samoprovera potvrdjena kao ispravna, to znači da sve se dešava kao što se očekivalo. Ako medjutim, ta samoprovera je potvrdjena kao neispravna, onda znači da je neočekivana greška otkrivena u programskom kodu.

**ASSERTIONS-PRIMERI**

Svaki program može da sadrži razne primere objava programa, o specifičnim pretpostavkama koje treba dokumentovati koristeći objave programa.

**ZAŠTITA PROGRAMA**

Ima tri načina da se zaštitimo od loših ulaznih podataka.

* Proveriti vrednosti svih podataka koji se unose od spolja, npr učitavanje podataka iz neke datoteke, od korisnika, mreže, ili nekog drugog eksternog interfejsa, proveriti da li podaci spadaju u dozvoljeni opseg
* Proveriti da li su numeričke vrednosti u okviru tolerancije, i dužina tekstualnih veličina da li je dovoljno kratka, ili da li je tekstualna varijabla adekvatna ili ne,
* Proveriti vrednosti svih ulaznih parametara za neki podprogram, i ovo je od iste važnosti kao i provera eksternih ulaznih vredosti, a razlika je u tome da ovi podaci dolaze iz drugog podprograma a ne od spolja

**Barikade** omogućuju da se ograniči šteta u programu, tj širenja kvara na ostale delove programa. Npr mogu se predvideti interfejsi kao granice prema delovima koji su osigurani.

Validacija, tj. validacija ulaza, Validation, to je tehnika gde se sprečava nastajanje izuzetka preko provere i korigovanja ulaznih podataka koji se unose u program.

#Jedna tehnika validacije je da se u program ubace validacione funkcije, i pomoću njih se vrši validacija, dakle provera i korigovanje ulaznih podataka.

$Ali, postoji i niz validacionih funkcija, koje već su razvijene i mogu se koristiti, tzv. utility-functions (tj. “komunalne“ funkcije), koje se koriste za tipične probleme koji se često javljaju.

# L13

#### KARAKTERISTIKE KVALITETA SOFTVERA

Imamo "eksterne" i "unutrašnje" karakteristike softvera, a „unutrašnje“ karakteristike su osobine koje su relevantne za softveriste.

Postoji čitav niz karakteristika softvera koji opisuju kvalitet softvera, npr.:

• korektnost tj.correctness , upotrebljivost (eng.usability), efikasnost (eng.efficiency), pouzdanost (eng.reliability), integritet (eng.integrity) softvera ( tj. zaštićenost od nepravilnog/neautorizovanog upada u softver), adaptivnost (eng.adaptability), tačnost (eng.accuracy), robusnost (eng.robustness).

Ovo su „eksterne“ karakteristike, tj. karakteristike koje su bitne za korisnike softvera.

A „unutrašnje“ karakteristike su osobine koje su relevantne za softveriste, npr.:

• Maintainability (tj. lakoća modifikovanja softvera),

• Fleksibilnost tj. eng.flexibility,

• portabilnost (eng.portability),

• reupotrebljivost (eng.reusability),

• čitljivost tj. eng.readability (ćitljivost izvornog koda),

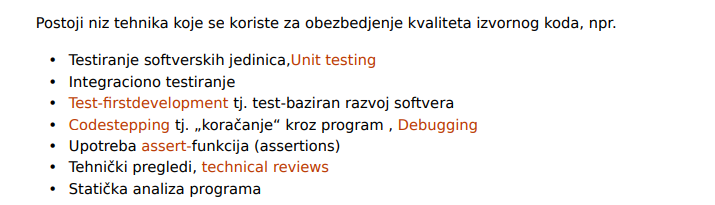
• testability, (da li se može izvršiti unit-test sistema i system-test sistema i verifikacija sistemskih zahteva)

• razumljivost tj. understandability (razumljivost organizacije celokupnog sistema)

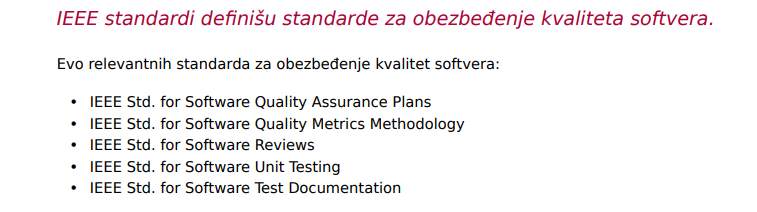
**Obezbeđenje kvaliteta softvera (software quality assurance) j**e planirani i sistematski program aktivnosti u cilju obezbeđenja željenih karakteristika sistema.

Ima niz tehnika za povećanje kvaliteta softvera: setting explicit quality objectives, setting explicit quality - assurance activities, strategija testiranja tj. testing strategy (kao glavni metod za procenu kvaliteta i povećanja kvaliteta),informal technical reviews, formal technical reviews, razvoj prototipa sistema (iskustvo pokazuje da razvoj prototipa sistema mogu da poboljšaju kvalitet softvera).

**Setting explicit quality objectives:** jedna moćna tehnika za postizanje kvaliteta softvera je eksplicitna specifikacija ciljeva kvaliteta, a na osnovu željenih karakteristika softvera, prethodno navedenih kao „eksterne“ i „interne“. Naime teba izabrati one karakteristike koje softver treba da ima na osnovu široke liste mogućih karakteristika. Dakle suziti listu karakteristika softvera na one karakteristike na kojim treba da se insistira u konkretnom projektu.



Neke tehnike povećanja kvaliteta su bolje u detekciji defekata , i različite metode detektuju različite vrste defekata.



**Code review (ili tzv. peer review)** tj. pregled ili provera kodiranja je sistematsko ispitivanje izvornog koda. Cilj je da se nadju greške i eliminišu greške koje se nisu videle u inicijalnoj razvojnoj fazi razvoja softvera, i tako se postiže

Code review može imati različite forme, npr.

• **Programiranje u paru** (par softverista programira zajedno)

**• Neformalni pregledi**

**• Formalne inspekcije**

Što se tiče **automatskog testiranja** (automated testing), postoje dva prilaza:

• **Code-driven testing,** gde se interfejsi (obično javni interfejsi) klasa, modula ili biblioteka se testiraju sa nizom ulaznih argumenata u cilju validacije vraćenih rezultata

• **GUI (graphical user interface) testing**

**Glavna tehnika za postizanje kvaliteta je**

• formalna inspekcija (formal inspections)

• tehnički pregled (technical reviews)

Formalni prilaz daje bolje rezultate od neformalnog prilaza.

Metrika koja se obično koristi kod inspekcije softvera je: errors-found/KLOC tj. nađene-greške/(1000 linija koda).

A efikasnost inspekcije se može meriti kao errrors-found/hour-spent tj. broj nađenih greški po utrošenom času.

### Formalna

Deo formalne inspekcije je i "checklist", tj. lista-provere, koja specificira detaljno šta treba da se proveri tokom inspekcije.

Formalna inspekcija je formalna, isplanirana/zakazana aktivnost gde prvo kreatori softvera prezentuju softver a onda izabrana grupa kolega ocenjuje tehničke aspekte softvera

Formalna inspekcija se izvodi rutinski za svaki softver, i pri tome se definišu specifične uloge pojedinim osobama (inspection roles) i definišu se specifični koraci inspekcije (inspection steps) .

Ipak, iako ima varijacija, sledeće inspekcione uloge se tipično koriste: • moderator, osoba koja sprovodi i organizuje inspekciju i izveštava rezultate inspekcije

• reader, šef stručnog tima inspektora

• recorder, tj. "zapisničar", vrši zabeleške o inspekciji i opisuje detaljno svaki defekt

• producer, "proizvođač", osoba koja je proizvela softver, ova osoba odgovara na pitanja tokom inspekcije, i popravlja nađene greške, i na kraju izveštava moderatora o popravkama

**A sledeći inspekcioni koraci se obično koriste:**

• overview, proizvođač proizvoda daje kratak opis proizvoda

• preparation, inspekcioni tim studira proizvod, i pri tome se često koristi check-list koja ukazuje na bitna pitanja

• inspection meeting,

• rework, proizvođač pregleda izveštaj sa sastanka i koriguje proizvod • follow-up, gde moderator pregleda izveštaj i korekcije, i ako je zadovoljan onda je inspekcija završena, a ako nije zadovoljan, onda se može zahtevati ponovorework ili ponovnu inspekciju

Tehnike kolaborativnog konstruisanja predstavljaju u stvari proces gde rad jednog programera se daje drugom programeru u cilju pronalaženja i eliminacije grešaka. To je inače praksa medju programerima da jedan programer ode kod kolege i zamoli ga da pregleda parče programa koji je napisao, i kaže mu koji problem ima tj. koji je problem uočio u tom novom parčetu programa koji je napisao. Ali kod kolaborativnog konstruisanja ova praksa se razvija do nivoa da se formalizuje i postaje zvanična i formalno definisana tehnika sa definisanim protokolom. To su sledeće tehnike:

• **Pair programmin**g tj. programiranje-u-paru

• **Formal inspection tj.** formalna inspekcija

• **Neformalni pregled** tj.informal review

• **Document reading** tj. čitanje dokumentacije

NEFORMALNI PREGLED

Neformalni pregled programa je popularna tehnika. Nije strogo definisano šta je to „neformalni pregled“, tako da svaka neformalna vrsta pregleda spada u tu kategoriju, npr.

• **Walk-through**s ( "brzi pregled")

• **code reading** tj. "čitanje program

**"Brzi pregled" (walk-through)** je po pravilu organizovan od strane samog autora dizajna ili programera koji implementira dizajn. Forma brzog pregleda na kraju poprima oblik radnog sastanka, gde se diskutuje dizajn ili programski kod.

**Čitanje programa (code reading)** je alternativa za brzi pregled, i ovde se čita samo izvorni kod programa i traže greške. Na osnovu izvornog koda programa se analizira i dizajn programa.

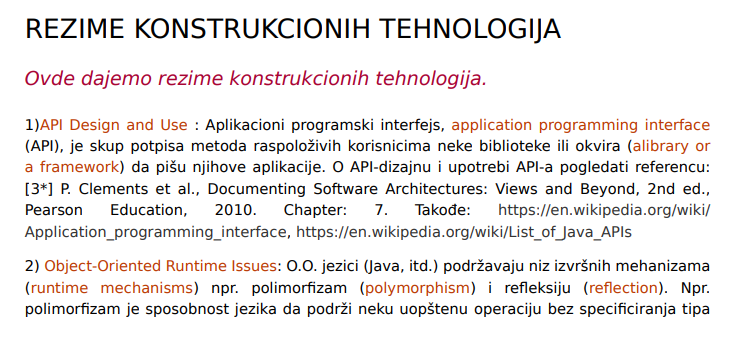
STATIC PROGRAM ANALYSIS

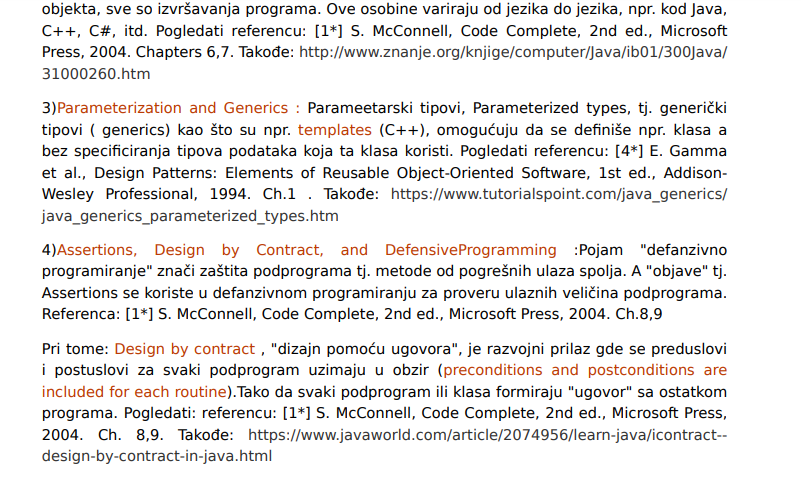
Statička analiza programa je analiza softvera koja se obavlja bez izvršavanja (egzekucije) programa.

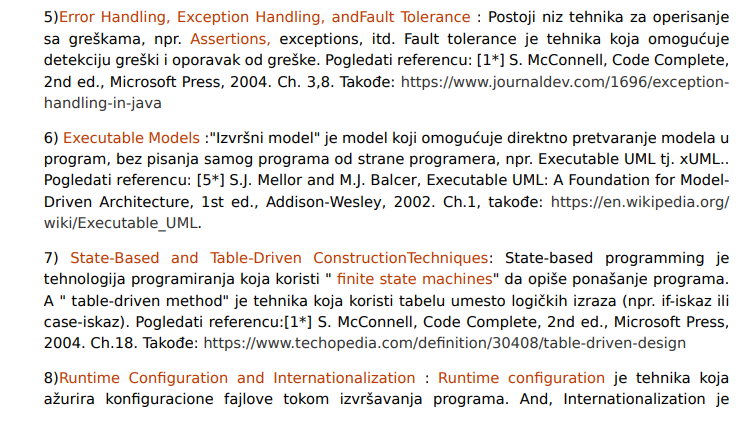
Dinamička analiza softvera je testiranje i evaluacija softvera u toku runtime. Dok statička analiza softvera je testiranje I evaluacija softvera pomoću proučavanja izvornog programa bez izvršavanja programa. Mnogi softverski defekti mogu se dtektovati pomoću statičke analize ili preko dinamičke analize. Ova dva pristupa se dopunjuju, jer ni jedan od ova dva prilaza ne otkriva sve greške.

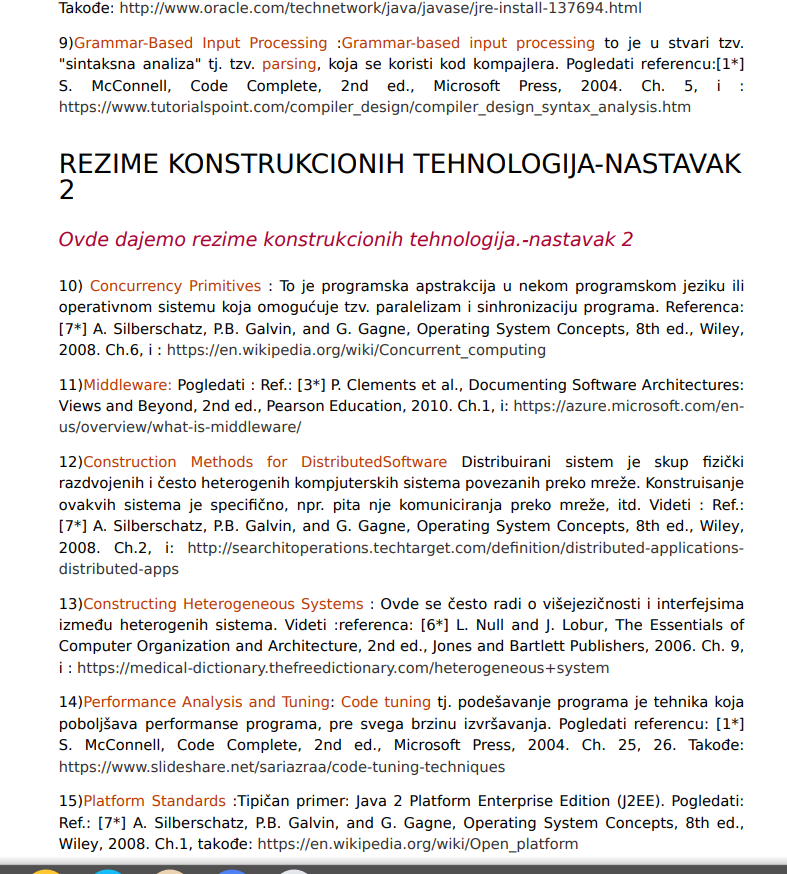
Obično se kompleksniji defekti koji su promakli kod statičke analize otkrivaju pomoću dinamičke analize. Kod statičke analize prednost je jer se istražuju sve moguće putanje programa I vrednosti varijabli, dok kd dinamičke analize se nalaze samo greške kod konkretne putanje izvršavanja I za konkretne vrednosti varijabli

# L14









## STRATEGIJA PODEŠAVANJA PROGRAMA -CODETUNING

(PODEŠAVANJE SOFTVERA)

Vrlo bitno je da obično je za sporost programa odgovorno nekoliko uskih grla u programu.

Performance tuning tj. code tuning tj. podešavanje performanci programa tj. podešavanje koda se bavi poboljšavanjem performanci programa, pre svega brzine programa. Na brzinu programa se može uticati na različite načine, i pod podešavanjem koda tj. code-tuning-om se podrazumeva modifikacija ispravnog koda tj. koda koji ispravno radi ali čija se modifikacija vrši da bi se postigla veća efikasnost tj. brzina programa.

Postavlja se pitanje**, kada treba vršiti code-tuning?** Odgovor je sledeći: ako se detektuje da program radi sporije nego što se očekivalo, ili ako program radi sporije nego što je to potrebno, onda treba primeniti **code-tuning tehnike**, kojih ima čitav niz, npr.

• compiler optimizations

• ponovno pisanje nekog ppodprograma ali u assembler-u

• upotreba in-line funkcija umesto običnih poziva funkcija

• optimizacija petlji

• itd

Primena optimizirajućih kompajlera (compiler optimization) može da uveća brzinu programa za 40%.

Većina code-tuning tehnika svaka ponaosob omogućuje povećanje brzine za oko 15-30%. Međutim, kada se sve tehnike primene zajedno, može se postići nekad povećanje brzine za 95% tj. može se brzina programa uvećati 20 puta.

Evo kako treba postupiti ako se otkrije da je neki program spor:

1) memorisati postojeću veziju programa,

2) izvršiti merenja brzine pojedinih delova programa da bi se otkrila uska grla

3) proučiti da li sporost programa potiče od neadekvatnog dizajna na visokom nivou, npr. lošeg izbora algoritama, i ako je to slučaj onda promeniti dizajn programa

4) proučiti koje code-tuning tehnike mogu da se primene na konkretnom primeru identifikovanih uskih grla u programu, i primeniti onda code-tuning tehnike

5) izvršiti merenje brzine, i ako nema povećanja brzine, onda se vratiti na prethodnu verziju programa

visokom nivou, npr. lošeg izbora algoritama, i ako je to slučaj onda promeniti dizajn programa