Razlozi nastanka objektno orijentisanog programiranja

Tendencija u razvoju primene računara je da se hardver razvija izuzetno brzo, pri čemu njegova cena stalno opada. Verovatno da razvoj u nijednoj drugoj oblasti ljudskog delovanja nije tako brz, kao što je slučaj kod računara. Dok su ciklusi poboljšanja u drugim oblastima prilično dugi, kod računarskog hardvera se svakih nekoliko godina snaga i brzina računara višestruko poboljšavaju.

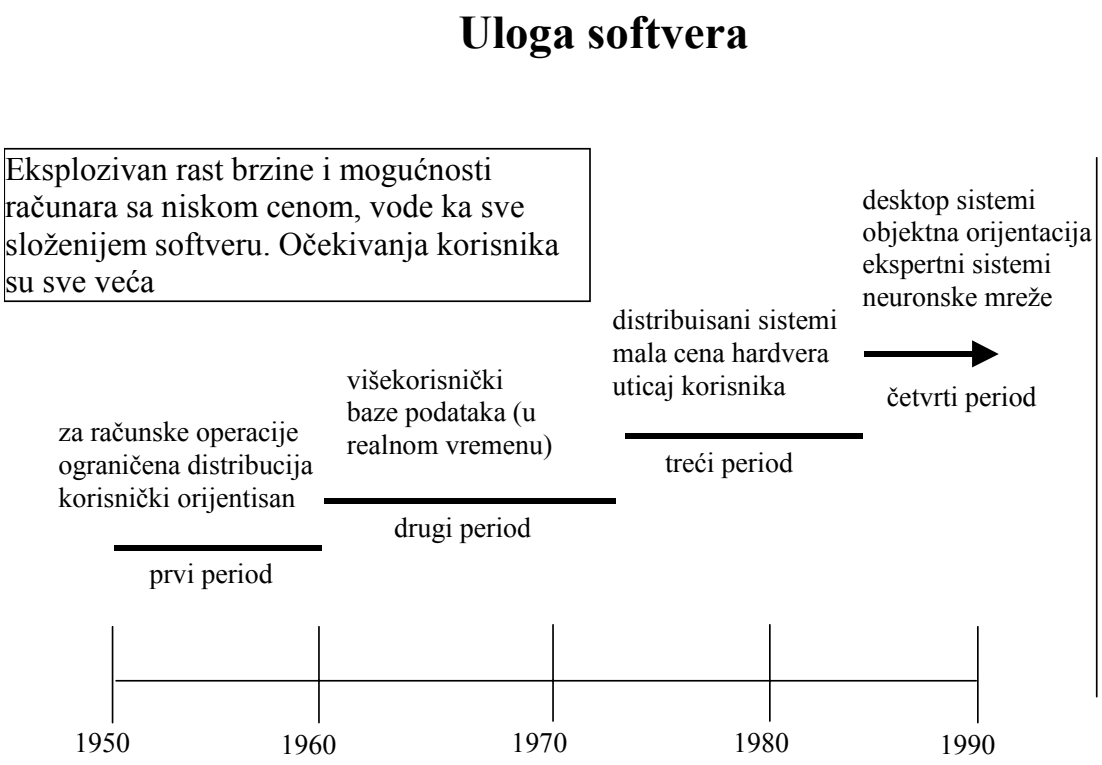
Dok je hardver bio relativno skup, privilegiju upotrebe računara imali su samo pojedinci i ustanove sa velikim budžetima. Računari su se primenjivali uglavnom za rešavanje relativno jednostavnih problema, kod kojih je bilo potrebno puno računice, ali sami problemi nisu bili preterano složeni.

Sa opadanjem cene i rastom popularnosti računara, a prvenstveno sa porastom njihove snage, raste i njihova upotreba u različitim oblastima ljudskog delovanja. Osamdesetih i devedesetih godina prošlog veka dolazi do brzog i eksplozivnog razvoja primene računara u svim oblastima. Kako je snaga računara rasla tako su rasli i zahtevi koji su se postavljali pred softver, koji na tim računarima radi. Programi su postajali sve složeniji i samim tim sve veći. Rasla su i očekivanja kupaca od softvera koji su nabavljali.

Posebne teškoće su se javile kod održavanja softvera. Kupci su stalno imali nove zahteve, vezane za poboljšanje ili popravku postojećeg softvera, a kako su sistemi postajali sve veći to je održavanje postajalo sve veći problem. Trebalo je snaći se u stotinama hiljada linija koda. Kada se i pronađe pravo mesto gde treba nešto promeniti, postavlja se pitanje u kakvoj je vezi taj kod sa ostalim delom programa. Održavanje je postalo vrlo teško i skupo.

Posebno pitanje je vezano za kvalitet softvera. Koliko program koji je isporučen zadovoljava ono što je kupac tražio. Koliko uopšte radi? Koliko je teško ili lako proširiti ga? Koliko je teško upotrebiti ga za rešenje nekog drugog problema i da li je to uopšte moguće?

Na sledećoj slici je grafički prikazan razvoj softvera tokom istorije?



U prvom periodu upotrebe računara softver je uglavnom bio usmeren ka konkretnim naručiocima, namenjen za rešavanje konkretnih problema. Program je radio samo kod onog ko ga je naručio i nije imao širu primenu.

U periodu od 1960-1970 počinje se sa razvojem višekorisničkih sistema, sistema za rukovanje bazama podataka, softverom koji radi u realnom vremenu.

Sedamdesete i osamdesete godine prošlog veka dovode do pojave distribuisanog softvera, sa niskom cenom hardvera i velikim uticajem zahteva korisnika (tržišta).

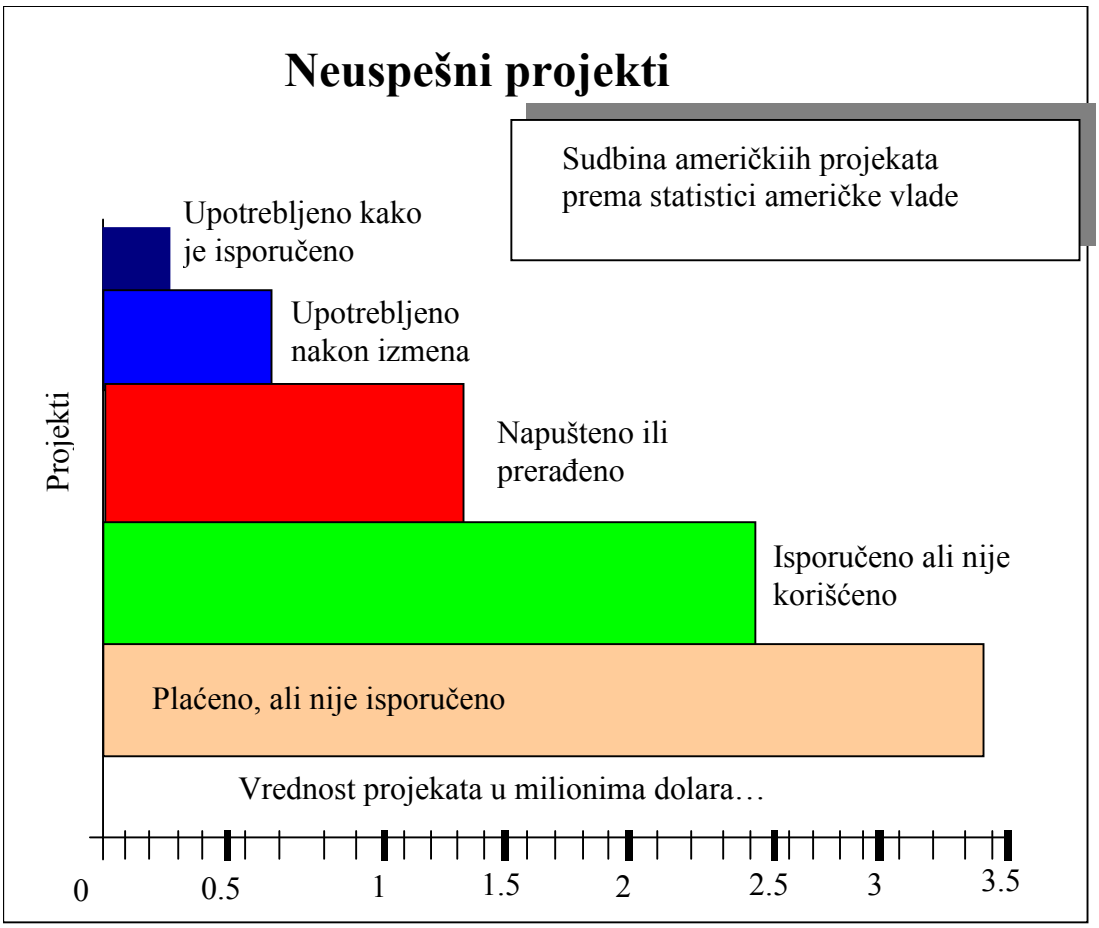
Poslednji period, koji traje i danas karakteriše intenzivna upotreba računara u svim oblastima. Pojavili su se desktop sistemi, koriste se objektno orijentisani metodi programiranja, primenjuje se veštačka inteligencija, a računari se i dalje razvijaju velikom brzinom.

U narednoj tabeli je pokazano kako se kretala cena hardvera i cena rada programera tokom godina.

	1965	1995
računar od 256 K	750 000 $	50 $
računar od 4 MB	?	1 000 $
Programer/godina	2 500 $	50 000 $
Softverski projekti	kasne	i dalje kasne
	imaju bagove	i dalje puni bagova
	nisu fleksibilni	nisu fleksibilni

Kao što se vidi iz ove tabele cena hardvera tokom godina značajno pada, dok cena softvera (preko cene rada programera) značajno raste.

Kako je složenost softvera rasla, rastao je i procenat neuspešnih projekata. Prema zvaničnoj statici američke vlade, u 1989 godini je raspodela uspešnih projekata izgledala ovako:



Softver postaje sve složeniji zato što su problemi koje rešava sve složeniji. Kod malih programa, su problemi koje treba rešiti mali, tako da je te programe mogla da kreira i održava jedna osoba.

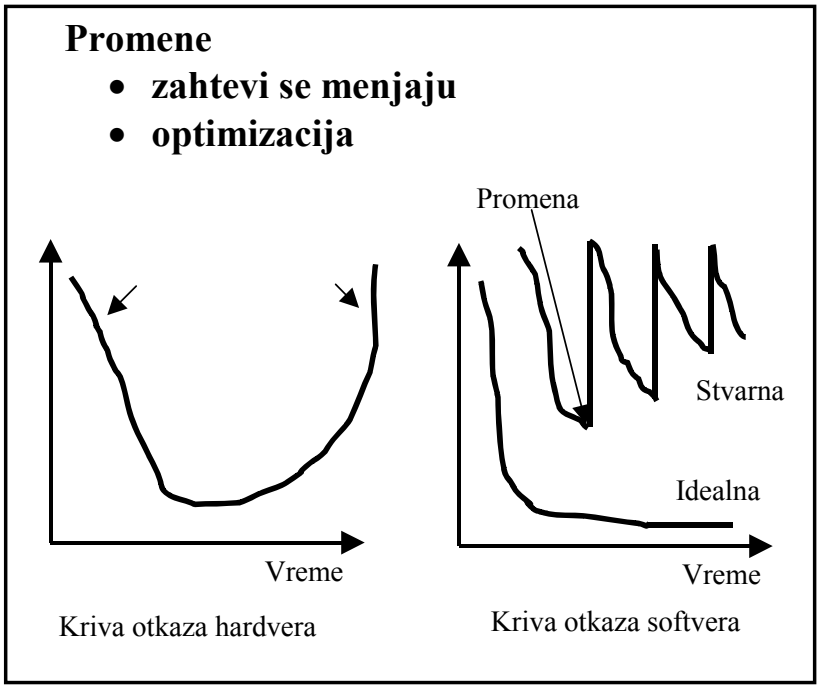
Veliki i složeni softverski sistemi imaju puno karakteristika, čije praćenje prevazilazi intelektualne kapacitete prosečne osobe. Jedna osoba nije više u stanju da stalno krpi i dopunjuje sistem, da održava zastarela programska okruženja ili da pažljivo kontroliše sve nastale promene.

Odgovor na ovo predstavljaju neke od osnovnih karakteristika OOP-a, a to su mogućnost ponovne upotrebe, mogućnost proširenja i kompatibilnost programa.

Osnovni razlozi nastanka neuspešnog softvera se mogu podeliti na:

- **složenost problema koji se rešavaju i stalna žurba tokom razvoja softvera (stalno se traži da program bude gotov za juče)** Postoji limit koliko iskusan programer može da napravi linija koda tokom jednog dana. Ako se od njega traži više od toga, taj kod ne može biti sasvim korektan. Objektno orijentisano programiranje ovde dolazi sa realnijim modelom problema koji se rešava, tako da i produktvnost programera unekoliko povećava. OOP sa sobom donosi i mogućnost ponovne upotrebe komponenti, što takođe može da u velikoj meri poboljša produktivnost.
- **saradnja članova tima**. Na razvoju velikih softverskih sistema moraju raditi timovi ljudi. Postavlja se pitanje kako funkcionišu veze između njih, odnosno veze između delova programa koje oni prave. Rad u timu ubrzava razvoj programa, ali donosi velike probleme kod integracije softvera, jer treba uklopiti ono što su oni uradili.

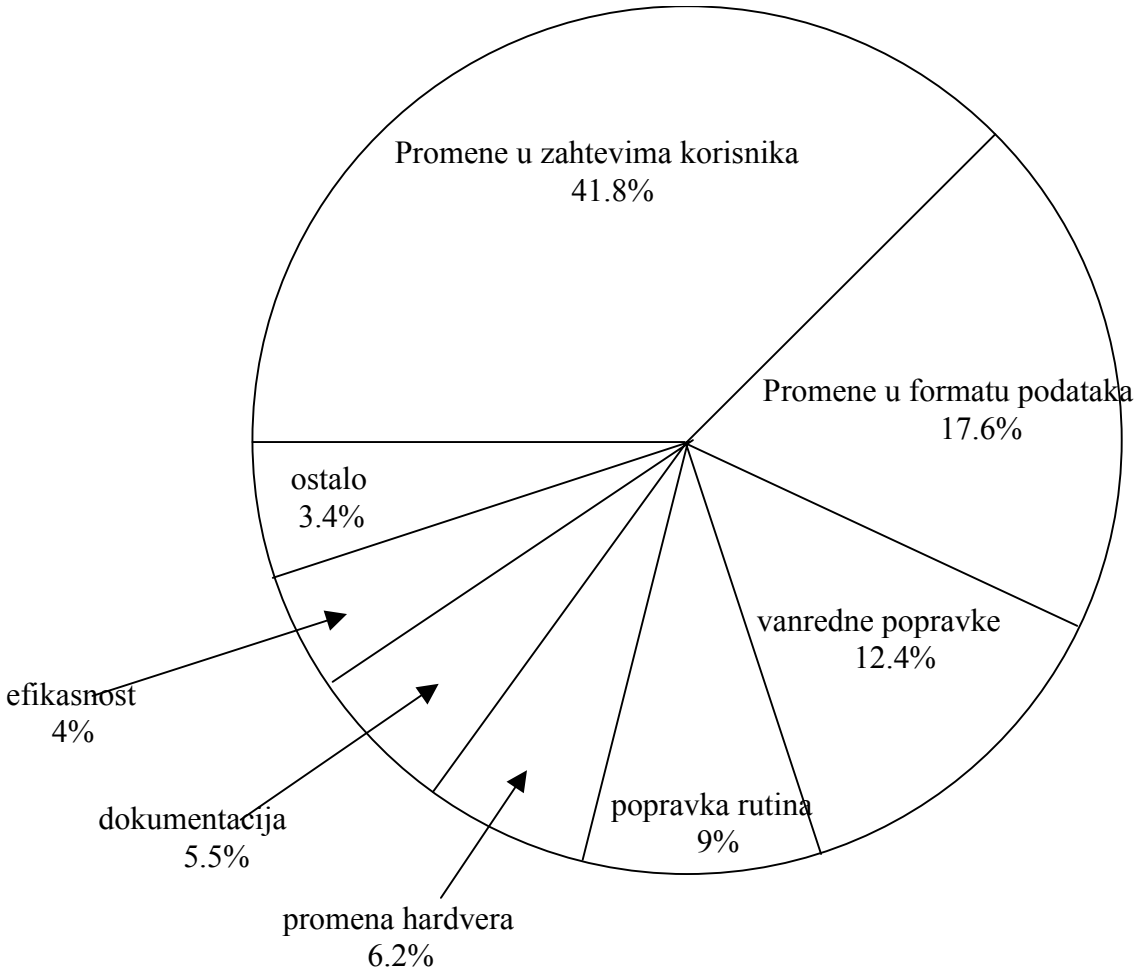
- **promene**. Softver je fluidan proizvod. Od njega se stalno traže neke promene. Korisnici se stalno postavljaju nove zahteve, na koje programeri treba da odgovore. Ti zahtevi često znaju da budu u protivurečnosti sa onima koji su prvobitno postavljeni. Krive otkaza hardvera i softvera su prikazane na sledećoj slici.



Neka istraživanja pokazuju da održavanje softvera odnosi 70% cene tog proizvoda.

Šta je uopšte održavanje softvera? To je ono što se dešava posle isporuke proizvoda. Softverski proizvod se ne troši usled upotrebe, tako da se ne održava na isti način, kao što je slučaj sa televizorom, ili automobilom. Održavanje se u suštini odnosi na dve vrste aktivnosti. Jedna vrsta je modifikacija programa. Kako se menjaju specifikacije računarskih sistema, koje odražavaju promene spoljašnjeg sveta, tako moraju da se menjaju i sami sistemi. Druga vrsta aktivnosti se odnosi na otklanjanje grešaka. To su greške koje nisu ni smele da postoje u sistemu i koje se otkrivaju tek u njegovoj eksploataciji.

Grafički proces održavanja, odnosno učešće pojedinih elemenata u ukupnom održavanju može prikazati na sledeći način:



Objektno orijentisano programiranje može posebno da utiče na promene zahteva korisnika i promene u formatu podataka.

Osnovne karakteristike promene zahteva korisnika se ogledaju pre svega u tome da se uglavnom radi o istim entitetima, da između tih entiteta ostaju slične veza (is-a, has-a), a da se menjaju interakcije između tih entiteta. Promene u modelu podataka su često minorne. Kako se OOP tu uklapa?

Odgovor je pre svega u mogućnosti ponovne upotrebe softvera, koju OOP propagira. Osnovne prednosti ponovne upotrebe softvera su da se time smanjuje vreme razvoja softvera, a da se istovremeno poboljšava kvalitet softvera.

Vreme razvoja se smanjuje time što programeri ne razvijaju ono što je već neko drugi razvio. Možda su to čak i oni sami uradili u nekom svom prethodnom projektu. Kod objektno orijentisanog programiranja je vrlo jednostavno klasu koja je negde funkcionisala, uklopiti u novi projekat.

Kvalitet softvera je takođe poboljšan jer su komponente koje se ugrađuju testirane i dokazano kvalitetne (takve bi trebalo uzimati). Ako bi programer sam pravio taj kod morao bi da izvesno vreme potroši i na njegovo testiranje, a da nema garancije da bi to bilo kvalitetno, kao komponenta koja je nabavljena od nekog ko se bavi samo tom oblašću.

Kolika je važnost ponovne upotrebe softvera za rešavanje novih problema, može se videti i iz činjenice da je američka vlada finansirala poseban projekat koji se bavi razvojem softverskih komponenti, koje se dalje koriste kod razvoja velikih softverskih sistema.

OOP može znatno da utiče i na promene u formatu podataka (pogledajte sliku vezanu za cenu održavanje softvera).

Ako se procedure i funkcije shvate kao glagoli (radnja), a podaci kao imenice, onda se strukturno programiranje može kvalifikovati kao programiranje kod kojeg imenice podržavaju radnju, dok se objektno orijentisano programiranje može shvatiti kao programiranje kod kojeg glagoli podržavaju imenice.

Kod tradicionalnog programiranja se problem rešava tako što se identifikuje algoritam (funkcije) potreban za rešavanje. Zatim se izvrši funkcionalna dekompozicija i definiše tok kontrole programa. Tek nakon toga se identifikuju strukture podataka, koje su potrebne da bi funkcije uradile posao za koji su napravljene.

Kod objektno orijentisanog programiranja rešavanje problema počinje sa identifikovanjem apstraktnih objekata (podatak) koji predstavljaju problem. Tek nakon toga se identifikuju apstraktne operacije koje ti objekti treba da podrže, odnosno definiše se interfejs objekata. Problem se na kraju rešava nizom poziva koji se upućuju objektima.

Objektno orijentisano programiranje se ovde uklapa preko jednog od osnovnih principa, a to je proširivost. Osnovni mehanizam za ostvarivanje proširivosti je nasleđivanje. Jedna klasa je istovremeno zatvorena celina, jer joj se pristupa uglavnom preko interfejsa, tako da je klijent koristi onakvu kakva jeste, ali je klasa istovremeno proširiva, jer naslednik klase ima sve njene osobine, a može dodati i svoje, nove, karakteristične.

OOP ima mnogobrojne prednosti, ali kao i svuda u životu postoje i određeni nedostaci. Jedan od nedostataka je svakako potpuno drugačiji način razmišljanja, koji programeri treba da usvoje. To ponekad nije lako, posebno za one koji su navikli na tradicionalno, strukturno programiranje. Pored ovog, većina programa pisanih u objektno orijentisanim jezicima radi sporije od programa pisanih u tradicionalnim jezicima. To je danas sve manji problem, pošto se hardver razvija ogromnom brzinom, tako da i objektno orijentisani jezici dobijaju zadovoljavajuće karakteristike.

Uvod u programski jezika Java

Java je programski jezik koji je poslednjih godina postao jedan od vodećih jezika. Ona se razvija već desetak godina, tako da je danas to zreo programski jezik, sa preležanim dečjim bolestima.

Java kao alat za programiranje

Danas svi govore o Javi kao alatu koji je pogodan za skoro sve namene u razvoju softvera. Skoro da ne postoji računarski časopis koji u svakom broju nema po neki tekst o Javi i njenim mogućnostima.

Postavlja se pitanje kakvo je to dramatično poboljšanje Java donela? To verovatno nisu promene u samom programskom jeziku, već više promene u Javinim bibliotekama. Tokom vremena je Sun Microsystems, koji razvija Javu, promenio skoro sve u tim bibliotekama, počev od imena funkcija, preko načina kako radi grafika (promena modela rukovanja događajima), pa do dodavanja novih mogućnosti, kao što je štampanje, koje u prvoj verziji Jave nije postojalo. Rezultat je mnogo korisnija programska platforma, koja je mnogo bolja od prvih, ranih verzija Jave.

Prednosti programskog jezika Java

Jedna od prednosti koja je očigledna je programsko okruženje koje obezbeđuje nezavisnost platforme. Isti kod možete da koristite na Windowsu, Solarisu, Linuxu, Macintoshu itd.

Druga prednost je u tome da je sintaksa Jave slična sa sintaksom jezika C++, tako da programeri koji su koristili C i C++ mogu lako da pređu na novu platformu. Naravno da programeri koji, na primer, rade u Visual Basicu, mogu doći do zaključka da ta sintaksa i nije tako sjajna.

Java je jezik koji je potpuno objektno orijentisan, mnogo više nego što je slučaj sa jezikom C++. Sve što u Javi postoji (osim nekoliko osnovnih tipova, kao što su brojevi) je objekat. Kreiranje novog dijalekta jezika C++, što se možda može zaključiti iz prethodno izloženog nije, međutim, bilo dovoljno. Ključna stvar je da je u Javi mnogo lakše napisati kod koji nema bagova, nego što je slučaj sa jezikom C++.U Javu su dodate mogućnosti koje eliminišu mogućnost kreiranja koda sa nekim greškama koje se najčešće javljaju. Šta je u pitanju:

- **Kreatori Jave su eliminisali ručno zauzimanje i oslobađanje memorije**. Memorija se u Javi automatski čisti. Programeri ne moraju da brinu o zauzeću memorije.
- **U Javi su uvedeni čisti nizovi, a eliminisana je aritmetika pokazivača**. Više ne morate da brinete o duplom zauzimanju memorije, kao kada ste radili sa pokazivačima.
- **Eliminisana je mogućnost greške kod zamene iskaza dodele**, **sa proverom ekvivalentnosti** (ovo se često dešavalo u C++-u).
- **Izbačeno je višestruko nasleđivanje**. Višestruko nasleđivanje je zamenjeno interfejsima. Interfejsi pružaju uglavnom one prednosti koje daje i višestruko nasleđivanje, ali se eliminiše složenost koja nastaje kod hijerarhija sa višestrukim nasleđivanjem.

Autori jezika Java su napisali jedan članak u kome su pokušali da objasne ciljeve koje su želeli da ostvare novim jezikom. Osnovni atributi jezika koji se pominju u tom članku su:

Jednostavnost

"Želeli samo da napravimo sistem u kome će se lako programirati, bez suvišne obuke. Sistem je trebao da se drži savremenih rešenja. U skladu sa tim, iako smo zaključili da je C++ nije najpogodniji, ipak smo Javu napravili tako da liči na C++, da bi sistem bio razumljiviji. Java sa druge strane izostavlja mnoge koncepte iz C++-a koji su dovodili do zabune. "

I zaista sintaksa Jave predstavlja prečišćenu verziju C++-a. Nema datoteka zaglavlja, pokazivača, struktura, unija, preklapanja operatora, virtuelnih osnovnih klasa itd.

Drugi aspekt jednostavnosti je da su programi mali. Jedan od ciljeva Jave je da se omogući kreiranje softvera koji može samostalno da radi na slabim mašinama. Interpreter i klase za podršku zauzimaju oko 40 Kb. Standardne biblioteke i podrška za niti traži još dodatnih 175 Kb.

Jezik je objektno orijentisan

Objektna orijentacija je tokom poslednjih dvadesetak godina dokazana, tako da je nezamislivo da neki od savremenih programskih jezika ne koriste takav pristup. Objektno orijentisane karakteristike Jave se ponovo mogu porediti sa jezikom C++. Osnovna razlika je u već pomenutom višestrukom nasleđivanju. Takođe i mehanizam refleksije, kao i serijalizacija objekata, omogućavaju mnogo lakšu implementaciju perzistentnih objekata.

Java je distribuisani jezik

U Javi postoji velika biblioteka rutina za rad sa TCP/IP protokolima, kao što su HTTP i FTP. Aplikacije pisane u Javi mogu da pristupaju objektima i da otvaraju objekte na mreži, preko URL-a. Ovo se radi isto tako jednostavno kao kada se pristupa lokalnom sistemu datoteka. Mrežne mogućnosti Jave su istovremeno vrlo velike i lake za upotrebu. Svako ko je programirao za Internet u nekom

drugom programskom jeziku može da kaže koliko je u Javi jednostavno otvoriti vezu preko soketa. Pored toga postoji i tehnologija servleta, koja je vrlo efikasna kod Java programa koji rade na serverima. Tu je i metod za poziv udaljenih metoda, koji se koristi za komunikaciju između distribuisanih objekata.

Pouzdanost

Kompajleri za Javu detektuju mnoge probleme, koje je kod drugih programskih jezika moguće otkriti tek u trenutku izvršenja programa. Ponovo se najveći problemi javljaju kod pokazivača, koji se koriste u jezicima C i C++, a kojih u Javi nema.

Sigurnost

Jedan od ciljeva prilikom kreiranja programskog jezika Java je bio rad u mrežnom i distribuisanom okruženju. U skladu sa tim u jeziku je puno rađeno na implementaciji sigurnosti. Kreatori Jave su pokušali da naprave sistem koji nije osetljiv na viruse i razne druge napade.

I pored prethodno navedenog u sigurnosnom mehanizmu koji je postojao u Javi 1.0 su vrlo brzo otkriveni propusti. Ljudi su nastavili da pronalaze bagove vezane za sigurnost i u svim narednim verzijama Jave. Dobra stvar u svemu tome je da Sun odmah nakon otkrivanja nekog od takvih bagova počinje da radi na njegovom otklanjanju. Pravi potez u tom smeru je bio i što je interna speceifikacija Java interpretera javno publikovana, tako da ljudi mogu lakše da pronalaze bagove i tome informišu kompaniju Sun.

Java je jezik koji ne zavisi od arhitekture računara

Java kompajler generiše nakon kompajliranja generiše datoteku u formatu koji ne zavisi od arhitekture. Kompajlirani kod se može izvršiti na različitim procesorima, naravno ako je prisutan sistem za izvršenje Java koda. Kompajler ovo postiže putem generisanja bajtkoda, koje nemaju ničeg sa arhitekturom konkretnog računara, već se te instrukcije itnerpretiraju na bilo kojoj mašini i u letu se prevode u instrukcije koje dotišna mašina razume.

Ovo nije nova ideja. Pre mnogo godina ta ideja je primenjena kod implementacije Pascala. U ovakvim situacijama osnovni problem postaju performanse. Autori Jave su napravili odličan posao razvojem bajtkoda, koji dobro radi na većini arhitektura koje se primenjuju kod današnjih računara.

Prenosivost

Za razliku od jezika C i C++, kod Jave nema aspekata specifikacije koji "zavise od implementacije". Veličine primitivnih tipova pdoataka su unapred definisane, kao i ponašanje aritmetike koja ih koristi.

Šta ovo znači? Promenljiva tipa int u C-u, na primer, može da biti i 16-o bitni i 32-bitni ceo broj. Kod Jave je to uvek 32-o bitni ceo broj. Jedino ograničenje je da tip int mora imati najmanje onoliko bajtova koliko ima tip short, a ne sme imati više bajtova od tipa long. Takođe se i stringovi, koji takođe mogu dovesti do problema, se čuvaju u standardnom Unicode formatu.

Biblioteke unapred pripremljenih klasa su deo sistema koji definiše prenosive interfejse. Na primer, postoji apstraktna klasa Window, koja ima posebne implementacije za UNIX, Windows i Macintosh računare.

Svako ko je nekada pokušao da napravi program sa prozorima, zna koliko je truda potrebno da se napravi program koji podjednako dobro izgleda na Windowsu, Macintoshu i različitim varijantama UNIX-a. Prve verzije Jave su predstavljale herojski pokušaj da svakvi problemi reše, ali je rezultat bila

biblioteka koja je retko davala prihvatljive rezultate. Ipak je to bio početak puta u pravom smeru. Novije verzije Jave su u tome otišle znatno dalje.

Java je jezik koji se interpretira

Interpreter za Javu može da izvršava bajt kod na bilo kojoj mašini, za koju postoji napravljen interperter.

Visoke performanse

Performanse koje postiže interpreter u većini slučajeva su sasvim zadovoljavajuće. Ako se desi da je potrebna veća brzina izvršenja programa, može se koristiti drugi oblik kompilacije, tzv. just-in-time (tačno na vreme) kompilacija. Ovo funkcioniše tako što se bajtkod jednom kompajlira, taj rezultat se kešira, a onda se poziva kada kasnije zatreba. Brzina koja se ovim dobija je značajno veća nego ako se koristi običan interpreter za Javu.

Java je jezik u kome se lako programira rad sa više niti

Ko je ikada pokušao da pravi program koji radi sa više niti, u nekom drugom programskom jeziku, pa proba da to uradi i u Javi, biće prijatno iznenađen. U Javi je sa nitima vrlo lako raditi. Niti u Javi imaju mogućnosti i da koriste višeprocesorske sisteme, ako operativni sistem to može da podrži.

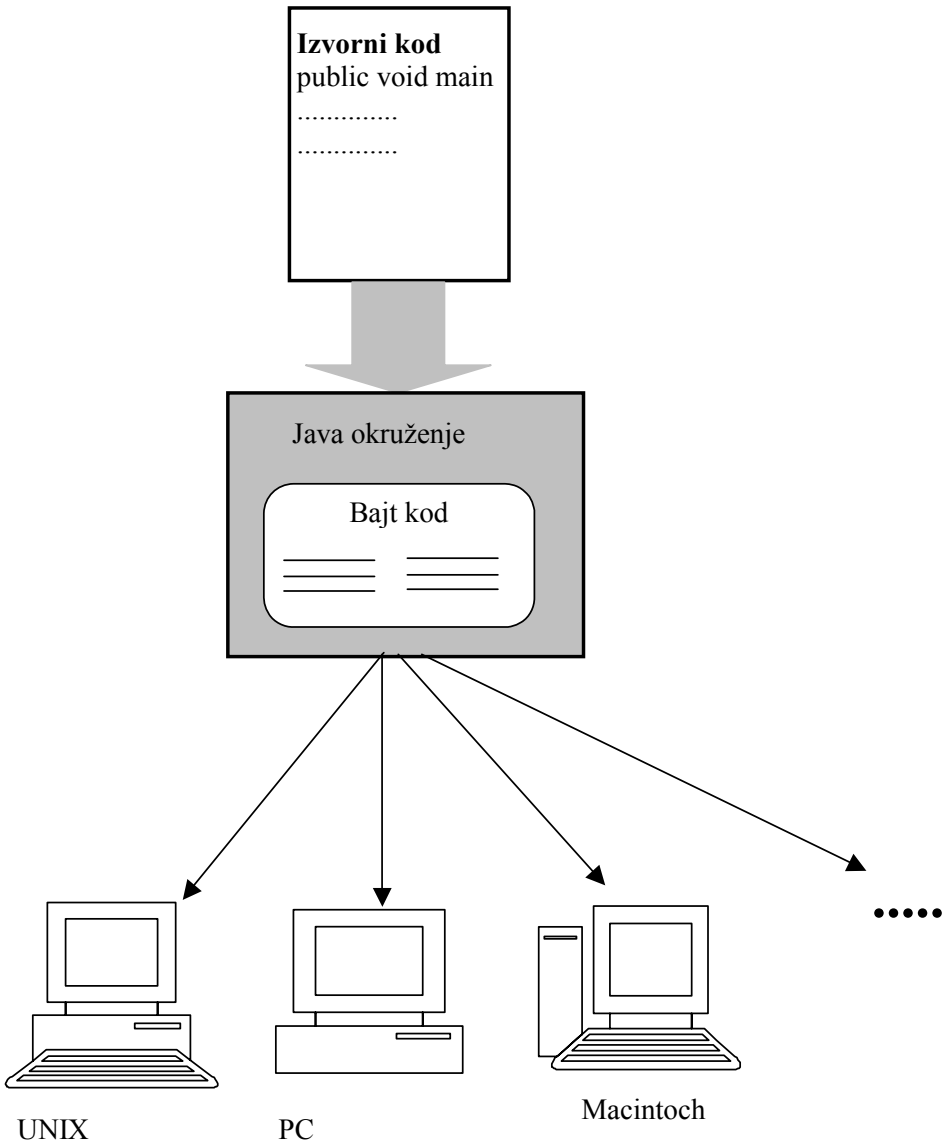
Virtuelna mašina za Javu

Šta je to što Javu razlikuje od ostalih jezika. Jedna od osnovnih karakteristika je to što je Java istovremeno i jezik koji se interpretira i jezik koji se kompajlira. Izvorni kod pisan u Javi se prevodi u jednostavne binarne isntrukcije, koje su slične sa mašinskim kodom koji procesor može da razume i izvrši. Razlika je u tome, što su kod jezika kao što su C ili C++ to instrukcije za konkretan model procesora. Kod Jave se izvorni kod kompajlira u univerzalni format. To su instrukcije za vrituelnu mašinu.

Kompajlirani Java bajtkod, se izvršava preko interpretera. Poseban sistem za izvršenje programa obavlja sve uobičajene aktivnosti koje radi i pravi procesor, ali to radi u sigurnom, virtuelnom okruženju. Sistem je taj koji kreira i manipuliše primitivnim tipovima podataka, on učitava i poziva nove blokove koda. Najvažnije od svega je da on to radi u skladu sa javno publikovanom specifikacijom, koju može da implementira bilo ko ko želi da napravi Java virtuelnu mašinu. Virtuelna mašina i specifikacija jezika zajedno šine kompletnu specifikaciju. U Javi ne postoji ništa što nije definisano ili što zavisi od implementacije.

Interpreter za Javu je relativno jednostavan i mali. On se može implementirati na svim platformama. Kod većine sistema interpreter je pisan u brzim jezicima, kao što su C ili C++. Interpreter se može pokrenuti kao posebna aplikacija ili se može ugraditi u druge programe, kao što je slučaj kod pretraživača za web.

Ovo znači da je kod pisan u Javi implicitno prenosiv (portabilan). Isti bajtkod se može izvršavati na bilo kojoj platformi koja ima okruženje za izvršavanje Jave. Ne morate da pravite alternativne verzije svojih aplikacija za različite platforme i ne morate da krajnjim korisnicima šaljete izvorni kod.



Slika: Okruženje za izvršenje programa pisanih u Javi

Pored sistema za izvršenje programa pisanih u Javi, u Javi postoji određeni broj osnovnih klasa, koje sadrže metode koji su različiti za različite arhitekture. Ovi metodi služe kao most između virtuelne mašine i spoljašnjeg sveta. Oni su implementirani u nekom jeziku koji se kompajlira i koji radi na određenoj platformi. Ti metodi se koriste za pristup mreži, sistemu prozora, ili sistemu datoteka. Ostatak Jave je pisan potpuno u Javi, pa je prema tome i prenosiv na sve platforme.

Pisanje programa u Javi

Pre nego što se napiše prvi program u Javi na računaru mora da se instalira odgovarajuće okruženje za kompilaciju i izvršenje Java programa. Konkretna uputstva za instalaciju i podešavanje će biti data na vežbama.

Prvi program u Javi

Sledeći kod predstavlja prvi program koji se obično daje kada se počinje sa nekim programskim jezikom. To je program koji na ekranu treba da ispiše tekst zdravo.

```
class Zdravo {
```

```
  public static void main ( String[] args ){
    System.out.println("Zdravo!");
  }
}
```

Ime ove datoteke mora biti Zdravo.java. To ime mora odgovarati imenu klase. Kod definisanja imena mora se voditi računa o velikim i malim slovima. Ime klase i ime datoteke moraju i po tome odgovarati jedno drugom.

Prvi red programa

```
class Zdravo
```

govori da je u pitanju izvorni program koji definiše klasu po imenu "Zdravo". Klasa je deo programa. Mali programi se obično sastoje od samo jedne klase. (Kasnije ćemo detaljnije definisati klase.) Kada kompajler prevede ovaj program dobija se datoteka sa imenom Zdravo.class.

Neke klase mogu da sadrže puno redova koda. Sve što pripada jednoj klasi mora da se postavi između otvorene i zatvorene velike zagrade ({ i }).

Ime klase (a time i ime datoteke sami određujete). Ime se mora sastojati od slova ili brojeva. Prvi karakter mora biti slovo, a unutar imena ne smeju postojati praznine. Obično imena klasa počinju velikim slovima, ali to nije obavezno. Ime izvorne datoteke obavezno ima ekstenziju .java (piše se malim slovima).

Sve što program radi mora biti između prve i poslednje zagrade. Obično se u jednoj datoteci nalazi samo jedna klasa, ali se u istoj datoteci može naći i više klasa.

Red

```
public static void main ( String[] args )
```

pokazuje gde program počinje. Reč main znači da je ovo glavni metod, odnosno metod sa kojim virtuelna mašina počinje izvršenje programa. Taj glavni metod u ovom slučaju sadrži samo jedan iskaz

```
System.out.println("Zdravo!");
```

Ovaj iskaz štampa tekst koji je dat u okviru dvostrukih navodnika, na ekran.

Izvorni kod ovog programa ste mogli da otkucate u bilo kom editoru teksta. Sledeći korak je kompilacija programa, odnosno njegovo prevođenje na bajtkod. Pod pretpostavkom da radite sa komandne linije, kompilacija bi mogla da izgleda ovako:

```
C:\TEMP>javac Zdravo.java
  compiling: Zdravo.java
```

Komanda javac predstavlja poziv kompajlera koji prevodi izvorni kod. Tom prilikom se dobija datoteka Zdravo.class. Ako sada želite da izvršite kompajlirani kod, treba da uradite sledeće:

```
C:\TEMP>java Zdravo
Zdravo!
C:\TEMP>
```

Sintaktičke greške

Vrlo je verovatno da će nešto poći naopako. Evo jednog programa sa namerno napisanom greškom.

```
Class Zdravo{
  public static void main ( String[] args )  {
    System.out.println("Zdravo!");
  }
```

```
}
```

Ključna reč class je ovde napisana sa velikim početnim slovom. Ovo je sintaktička greška. Sintaktičke greške su "gramatičke greške" kod rada sa nekim programskim jezikom. Ako se ovaj program kompajlira prikazaće se sledeće:

```
C:\Temp>javac Zdravo.java

  compiling: Zdravo.java
Zdravo.java:1: Class or interface declaration expected.
Class Zdravo
^
1 error
```

Kompajler je pokušao da prevede izvorni kod u bajtkod, ali se zbunio, jer je naišao na veliko slovo C tamo gde to nije očekivao. Poruka o grešci koju je dao, nije baš najjasnija. Te poruke nikad nisu potpuno jasne. Ipak ova geška bar pokazuje u kom redu se kompajler zbunio. U ovom slučaju kompajler nije napravio novu izvršnu datoteku, pošto je prevođenje zaustavljeno.

Da program ponovo proradio moramo da ispravimo grešku. Vratićemo se u editor i promeniti veliko slovo C na malo. Sada treba ponoviti ciklus kompilacije i izvršenja.

Ciklus promene, prevođenja i izvršenja

Ovde smo na malom primeru pokazali šta treba uraditi u slučaju da se u programu javi sintaktička greška. Kako je u pitanju ciklus koji ćete, tokom programiranja, ponavljati nebrojeno puta, evo kratkog rezimea. Sledeće treba uraditi sve dok program ne bude radio kako treba:

1. Unesite program uz pomoć nekog editora teksta.
2. Upamtite program na disku.
3. Kompajlirajte program pomoću komande javac.
4. Ako postoje sintaktičke greške, vratite se na korak 1.
5. Pokrenite program pomoću komande java.
6. Ako program ne radi kako treba, vratite se na korak 1.
7. Ako sve radi kako treba, završili ste.

Logičke greške

To što je program uspešno preveden (kompajliran) ne mora da znači da će on ispravno raditi. Greške koje se manifestuju u radu programa, koje niste očekivali se nazivaju logičkim greškama, ili bagovima u programu. Ako ste na primer dobili zadatak da odštampate tekst Zdravo, a napišete sledeći program, program će se iskompajlirati, ali nećete ostvariti ono što ste trebali.

```
class Zdravo{
  public static void main ( String[] args )  {
    System.out.println("Zdravo Perice!");
  }
}
```

Logičke greške se naravno mnogo češće javljaju u programima koji su duži od naših primera. Logičke greške je mnogo teže otkriti i one se uglavnom otkrivaju tek kasnije kada program počne da živi u praksi. Zbog toga je vrlo važno pre realne instalacije programa izvršiti njegovo testiranje. Testiranje treba da rade i programeri, ali i krajnji korisnici u realnim uslovima. Testiranje je posebna priča, o kojoj će kasnije biti više reči.