Usporedna analiza objektivno-orijentiranih programskih jezika

Buršić, Petra

Sveučilište Jurja Dobrile u Puli
Odjel za informacijsko-komunikacijske tehnologije

PETRA BURŠIĆ

USPOREDNA ANALIZA OBJEKTNO-ORIJENTIRANIH PROGRAMSKIH JEZIKA

Završni rad

Pula, rujan, 2017. godine

Sveučilište Jurja Dobrile u Puli
Odjel za informacijsko-komunikacijske tehnologije

PETRA BURŠIĆ

USPOREDNA ANALIZA OBJEKTNO-ORIJENTIRANIH PROGRAMSKIH JEZIKA

Završni rad

JMBAG: 0303054152, redovni student
Studijski smjer: Informatika

Predmet: Programiranje
Znanstveno područje: Društvene znanosti
Znanstveno polje: Informacijske i komunikacijske znanosti
Znanstvena grana: Informacijski sustavi i informatologija
Mentor: doc. dr. sc. Tihomir Orehovački

Pula, rujan, 2017. godine

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani _____, kandidat za prvostupnika _____ovime izjavljujem da je ovaj Završni rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Završnog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, _____, _____ godine

IZJAVA
o korištenju autorskog djela

Ja, _____ dajem odobrenje Sveučilištu Jurja Dobrile
u Puli, kao nositelju prava iskorištavanja, da moj završni rad pod nazivom _____
_____
_____ koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.
Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, _____

Potpis
_____

Sadržaj

1. Uvod

U ovome radu obrađena je tema objektno-orijentiranog programiranja, odnosno usporedna analiza objektno-orijentiranih programskih jezika.

Temeljni principi objektno-orijentiranog programiranja su enkapsulacija, nasljeđivanje i polimorfizam.

Enkapsulacija jest princip implementacije koji skriva podatke od okoline, dakle opis javnog sučelja i implementacije. Nasljeđivanje je svojstvo kojim klase imaju mogućnost naslijediti određena svojstva, odnosno strukturu i ponašanje druge klase. Polimorfizam pruža dinamičko vezanje poruka prema definicijama metoda, također implementirano kroz nasljeđene klase.

Dok su za usporedbu odabrani programski jezici Smalltalk, C++, Java i Python.

Smalltalk je odabran kao jedan od prvih, pa sa time i zastarijelih, objektno-orijentiranih programskih jezika koji čini temelj suvremenih jezika. U svoje vrijeme predstavljao je revolucionaran jezik koji uvodi objektno-orijentiranu paradigmu koja je zatim znatno pojačana i unaprijeđena u složenijim jezicima poput C++-a.

Dakle, C++ predstavlja modernijeg predstavnika kategorije koji je relativno složen jezik, no vrlo efikasan. Java predstavlja noviji jezik s automatskim upravljanjem memorije čija je glavna značajka pouzdanost. Python je jezik izgrađen funkcionalnim i objektno-orijentiranim principom što ga čini vrlo elegantnim za korištenje i pogodnim za brzi razvoj softverskih sustava.

Programiranje je upravo najzanimljiviji i najkreativniji dio interakcije čovjeka sa računalima, međutim pri odabiru programskog jezika potrebno je analizirati svojstva pojedinih kandidata i objektivno analizirati pozitivne i negativne strane istih kako bi se odabrao programski jezik koji najviše odgovara potrebama projekta.

U obzir treba uzeti brzinu razvoja, jednostavnost razumijevanja i pisanja programskog kôda, pouzdanost programskog jezika te brzinu procesnog vremena programa što je detaljnije opisano u nastavku rada.

Drugo poglavlje govori o evoluciji objektno-orijentiranih programskih jezika kroz vremensku liniju. Opisani su glavni koncepti programiranja koji prethode objektno-orijentiranu paradigmu. Nadalje se uvodi suvremeni princip objektno-orijentiranog programiranja s temeljnim značajkama.

U četvrtom poglavlju opisuje se razvoj i glavna svojstva Smalltalk-a, jezika iz kojeg potječe objektno-orijentirana paradigma.

U petom poglavlju opisuje se C++, zatim u šestom Java, te u sedmom poglavlju opisane su glavne prednosti i mane jezika Python.

Osmo poglavlje predstavlja krajnju usporedbu spomenutih programskih jezika sa pozitivnim i negativnim stavkama koje ih čine pogodnima za specifične vrste zadataka.

U devetom poglavlju iz analize se zaključuje da je nužno poznavanje pojedinih značajka programskih jezika kako bi se pri izradi projekta odabrao onaj jezik koji je objektivno najpogodniji za određeni zadatak.

2. Evolucija objektno-orijentiranih programskih jezika

Viši programski jezici (eng. High level languages) pojavili su se početkom 60-ih godina prošlog stoljeća, a to su FORTRAN, COBOL i BASIC.

FORTRAN je prvi programski jezik visokog poretka i namijenjen je za matematičke vrste problema, dok je COBOL namijenjen poslovnim problemima.

U sedamdesetim i osamdesetim godinama 20. stoljeća pojavljuju se novi programski jezici kao što su Wirthov Pascal i Stroustrupov C++.

„Radi se o modifikacijama viših programskih jezika, unaprijeđenim u skladu s razvojem novih tehnologija i metoda programiranja, grafičkog sučelja te – u novije vrijeme – interaktivnog i objektnog pristupa programiranju i računalima uopće.“ (Lipnjin, 2004, str. 17).

Programiranje je najkreativnija strana interakcije sa računalima. Često programeri vrše odabir programskog jezika s obzirom na svoja subjektivna vjerovanja i iskustva, što nije uvijek vjerodostojno. Programeri moraju odabrati programski jezik u skladu sa zadatkom kojega je potrebno riješiti. Dakle, potrebno je na objektivan način usporediti i analizirati različite strane programskih jezika te utvrditi prvo što je potrebno ostvariti samim zadatkom i koje su značajke najbitnije, kao na primjer sigurnost ili brzina, a zatim koji programski jezik odgovara svim zahtjevima.

Programski su jezici kroz godine razvili bolju standardnu sintaksu koja uvodi nove značajke u unutarnjoj strukturi podataka samih računalnih programa kao što su programski stog (eng. stack) i hrpa (eng. heap).

Nadalje, objašnjen je razvoj i evolucija dominantnih koncepta programskih jezika kao što su: strukturno programiranje, funkcionalno programiranje i logičko programiranje.

Strukturno programiranje uvodi mnoge osnovne koncepte programskih jezika koje podižu razinu istih na viši nivo. Cilj strukturnog programiranja je smanjiti ili potpuno eliminirati korištenje skoka GOTO. Skokovi su kontrolne strukture čije se korištenje eliminra radi štetnosti nad samom hijerarhijskom strukturom programa. Skok GOTO je bezuvjetni skok koji čini program teškim za razumijevanje ukoliko čini isti manje logičnim, odnosno teškim za održavanje. Ipak, neki se znanstvenici slažu s

korištenjem GOTO skoka ukoliko je isti rješenje za rukovanje iznimkama što je iznad bilo koje logične strukture programa.

Strukturno programiranje predstavlja prvi programski koncept koji se bazira na logičkim modelima te uvodi kontrolne strukture, strukture podataka i odozgo prema dolje (eng. top-down) programski dizajn (Lovrenčić i sur., 2009).

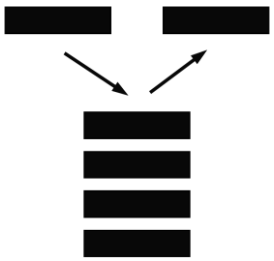
Teorem strukturnog programiranja govori da je svaki programski zadatak moguće logički programirati koristeći samo kontrolne strukture strukturnog programiranja, a to su sekvenca, iteracija i selekcija.

Sekvenca, slijed ili programski blok predstavlja niz instrukcija obrade koje se izvode slijedno i završavaju znakom točka zarez, ';'. Jedna sekvenca čini pravilan program koji ima jednu ulaznu i jednu izlaznu liniju koje povezuje put te nemaju nedostupnih dijelova poput već spomenutih skokova (Radošević, 2007).

Selekcija ili grananje jest kontrolna struktura kojom se odabire i izvršava određena sekvenca s obzirom na nekakav logički uvjet. Kao primjer mogu se navesti selekcija tipa „if“ i „if-else“, odnosno grananje u dva ili više smjera, i selekcija tipa „switch“, odnosno grananje u više smjerova sa uvjetovanim slučajevima (eng. case). Dok iteracija ili petlja postavlja izvršavanje određene sekvence više broj puta kao što su iteracija tipa „for“ s brojačem, te iteracija sa logičkim uvjetom „while“ sa uvjetom na početku petlje i „do-while“ sa uvjetom na kraju petlje.

Jedan od osnovnih koncepta strukturnog programiranja jesu potprogrami ili funkcije, odnosno logičke cjeline unutar programa sa imenom, argumentima, lokalnim podacima i povratnim vrijednostima. Za pamćenje povratnih podataka, odnosno adresa istih, koristi se struktura stoga (Ibidem).

Stog je struktura podataka koja radi po principu „LIFO“ (eng. Last in, First out) ili „FILO“ (eng. First in, Last out), odnosno zadnji element koji ulazi u strukturu je ujedno i prvi element koji će izaći iz nje ili prvi element koji je ubačen unutar strukture jest ujedno i element koji će zadnji izaći iz iste. Operacije ubacivanja i izbacivanja elemenata iz strukture vrši se funkcijom ubacivanja (eng. Push) i vađenja (eng. Pop). Struktura stoga prikazana je na slici broj jedan u nastavku.



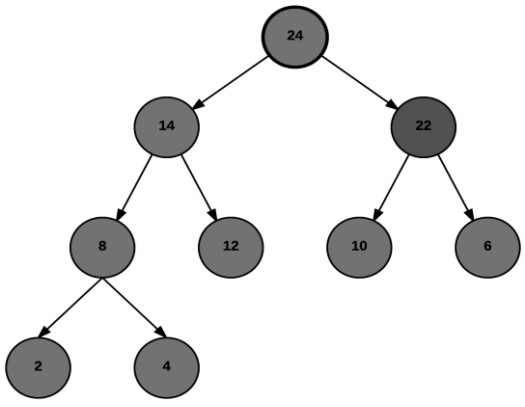
Slika br.1: Stog

Poziv potrprograma, unutar programa, vrši se pomoću strukture stoga.

Strukturno programiranje uvodi novo razvijeni pristup koji se naziva rekurzija. Rekurzija je svojstvo potprograma da može pozivati sam sebe te sadrži sidreni izraz koji omogućuje izlazak iz rekurzivne funkcije, odnosno sprečava beskonačno pozivanje iste što znatno usporava vrijeme izvršavanja programa (Ibidem).

Osim rekurzije uveden je i novi tip varijable, odnosno varijable koje sadrže memorijsku adresu drugih podataka pod nazivom pokazivači. Prema tome razvijena je još jedna struktura podataka pod nazivom hrpa ili gomila.

Hrpa je uređeno potpuno binarno stablo gdje roditelj uvijek mora biti veći od svoje djece, ili uvijek manji, ovisno o implementaciji. Na slici broj dva prikazan je primjer hrpe u kojoj je svaki čvor najveći u podstablu kojemu je on korijen (Kusalić, 2010).



Slika br.2: Hrpa

Sljedeći dominantan koncept programskih jezika jest funkcionalno programiranje.

Većina računalnog koda bavi se standardnim kalkulacijama, sortiranjem i drugim standardnim akcijama koje se ponavljaju u svakom računalnom programu. Ideja je potaknuti programere da razmišljaju na drugačiji način, odnosno programeri više ne razmišljaju o tome *„Kako to učiniti?“* već *„Što učiniti?“.* Drugim riječima, ideja je izgraditi sistem sa standardnim algoritmima za standardne akcije koje bi omogućile programeru da jednostavno opiše ono što hoće ostvariti, i ostavi odabir algoritma prevoditelju (Lovrenčić i sur., 2009).

Funkcionalno programiranje se može promatrati kao stil programiranja čija osnovna računalna metoda jest primjena funkcija na argumente. Ime dolazi iz činjenice da program napisan u takvom jeziku se sastoji potpuno od funkcija. Glavna (eng. main) funkcija je definirana u pogledu drugih funkcija koje se definiraju u pogledu još funkcija (Ibidem).

U suvremenom računalnom svijetu uključene su ideje i koncepti nastali upravo u funkcionalnom programiranju poput apstrakcije podataka, generičnost, polimorfizam i preopterećenje, te kasnije usvojeni u objektno-orijentiranom konceptu programiranja i implementirani u proceduralnom programiranju.

Sljedeći dominantan koncept programskih jezika jest logičko programiranje koje sadrži nove koncepte koji su već definirani u funkcionalnom programiranju i govori da podatci ne moraju biti eksplicitno definirani u programu već se deriviraju, a zatim se definira nova organizacija dinamičke baze podataka koja je dio programa.

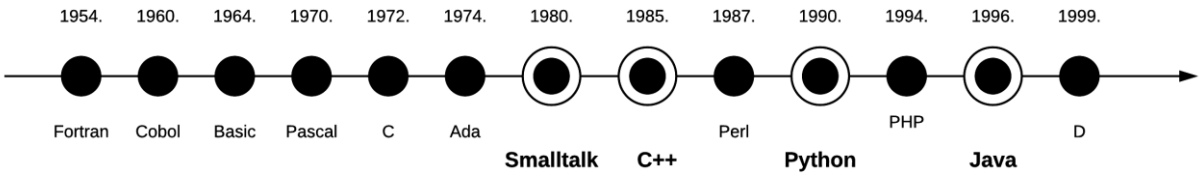
Akronim „FOL“ (eng. First Order Language) predstavlja logičke jezike prvog reda gdje ne moraju sve formule imati jedinstvenu interpretaciju te postoji problem u definiranju negacije. Dok u logičkom programiranju interpretacija za sve formule mora biti definirana što znači da način na koji je negacija definirana u logičnom programskom jeziku je bazirano na pretpostavci koja govori da sve što nije istinito, je laž. Definicija jednakosti u logičnim programskim jezicima govori da dva pojma su jednaka ako i samo ako je tako definirano u programu eksplicitno prema činjenici ili implicitno prema pravilu (Ibidem).

U nastavku se obrađuje proceduralno-objektni model, odnosno principi objektno-orijentiranog programiranja.

3. Principi objektno-orijentiranog programiranja

Proceduralno-objektni model rastavlja program na niz zatvorenih cjelina koje objedinjuju operacije i podatke u objekte prema čemu je novi koncept dobio ime 80-ih godina 20. stoljeća, a radi se o objektno-orijentiranom pristupu (eng. Object-oriented approach).

Objektno-orijentirani princip je inspiriran jezikom Smalltalk-80 kojeg je zamislio Alan Key 1969. godine, a počeo se upotrebljavati tek 1980. godine. Bjarne Stroustrup je oblikovao C++, odnosno objektnu ekstenziju jezika C. Jezik C++ se počeo primjenjivati 1985. godine. Kasnije su se razvili: Pearl autora Larry Wall-a 1987., zatim Python autora Guido van Rossum 1990. i Java u vlasništvu firme Sun Microsystems koji je objavljen 1996. godine (Lipnjin, 2004).



Slika br.3: Vremenska crta programskih jezika

Terminologija „Object Oriented Programming“ (OOP) i „Object Oriented Language“ (OOL) na hrvatskom je jeziku prevedeno kao „Objektno programiranje“ i „Objektni jezik“ međutim bitno je naglasiti da nije sam postupak objektan, već se sa objektima radi. Dakle, hrvatski prijevodi za „Object Oriented Programming“ (OOP) i „Object Oriented Language“ (OOL) glase „Objektno Orijentirano Programiranje“ (OOP) i „Objektno Orijentiran Jezik“ (OOJ).

Objektno orijentirani programski jezici ne definiraju strukture podataka i funkcije odvojeno, već ih prilažu sve u objekte.

Objekti su statički entiteti u programima, iako oni sadrže funkcije pod nazivom metode koje se koriste za procesiranje atributa, odnosno za procesiranje podataka objekta. Razlog tome je činjenica da su objekti građeni oko podataka koje isti sadrže. Definicija objekta se naziva klasa. Klasa definira tip podataka objekta pojedine klase i metode koje ih procesiraju. Objekti ne sadrže bilo koje metode i to je razlog zašto

objektno-orijentirano programiranje se mora promatrati kao način za organizaciju podataka u računalnom programu (Lovrenčić i sur., 2009).

Iako je početak objektno-orijentiranih programskih jezika bio kasnih 1960.-ih, kada su razvijeni Simula i Smalltalk, realna ekspanzija započela je kasnih 1970.-ih i 1980-ih godina kada je razvijen jezik C++.

Dakle, objektni pristup programiranja podržava objekte, odnosno spoj podataka, operacija i metoda, koji predstavljaju strukturu programa i apstrakciju, odnosno sposobnost programa da zanemari detalje i pažnju usmjeri na ono što je bitno (Lipnjin, 2004).

Osnovna svojstva objekto-orijentiranih programskih jezika jesu enkapsulacija (eng. encapsulation), nasljeđivanje (eng. inheritance) i polimorfizam (eng. polymorphism).

Enkapsulacija ili učahurivanje jest spajanje, odnosno zajedništvo strukture podataka s metodama. Ovo svojstvo sadrži i skrivanje ili privatnost podataka (eng. data hiding) koji je detaljnije opisan u potpoglavlju „4.1 Enkapsulacija“.

Nasljeđivanje je svojstvo koje pruža mogućnost da se objekti definiraju kao nasljednici ili podređene klase te pretci ili super, odnosno nadređene klase. Glavna funkcija ovoga svojstva jest ponovna upotrebljivost. Super i podređene klase osim zajedničkih svojstva mogu imati i svoja vlastita dodatna svojstva kao što su podaci, operacije i metode.

Polimorfizam jest koncept koegzistencije entiteta s istim imenom unutar programa koji se preciziraju pri izvođenju programa.

Detaljnija razrada osnovnih svojstva objektno-orijentiranih programskih jezika obrađuju se u slijedećim podnaslovima u nastavku rada „3.1 Enkapsulacija“, „3.2 Nasljeđivanje“ i „3.3 Polimorfizam“.

3.1 Enkapsulacija

Enkapsulacija predstavlja princip kojim se atributi i metode spajaju u cjelinu pod nazivom klasa.

Općenito, klasa se sastoji od svojeg javnog sučelja i implementacije iste. Reprezentacija objekta je implementacija, te se među objektima interno vrši komunikacija.

„*Implementacija je najčešće skrivena od okoline kako bi se osigurala konzistentnost objekata. Klasa se, dakle, sastoji od opisa javnog sučelja i implementacije. To objedinjavanje javnog sučelja i implementacije naziva se enkapsulacija (eng. encapsulation).*“ (Motik i Šribar, 1997, str.215).

U deklaraciji same klase potrebno je dodati specifikatore pristupa koji će, sukladno sa svojim nazivom, određivati pravo pristupa za određene atribute i metode klase. Specifikatori mogu biti javni (eng. public), zaštičeni (eng. protected) i privatni (eng. private). Članovi javnog specifikatora dostupni su iz bilo kojeg dijela programa, članovi zaštičenog prava pristupa dostupni su iz osnovnih klasa ili naslijeđenih klasa, dok privatni specifikator prava pristupa označava da su članovi dostupni isključivo unutar deklaracije same klase. Naravno, za svako pravilo postoji iznimka ukoliko su pravila stvorena kako bi se rušila.

Postoji nekoliko prednosti skrivanja informacija, jedan od njih je povećana pouzdanost. Programske jedinice koje koriste određenu vrstu apstraktnih tipa podataka nazivaju se klijentima. Klijenti ne mogu izravno manipulirati temeljnim prikazima čime se povećava integritet takvih predmeta. Objekti se mogu mijenjati samo kroz navedene operacije (Sebesta, 2012).

Još jedna prednost skrivanja informacija je smanjenje raspona kôda i broj varijabli kojima programer mora rukovati prilikom pisanja programa ili čitanja dijela programa. Vrijednost određene varijable može biti promijenjena samo u djelu kôda ograničenog raspona, čime je lakše razumjeti kôd i manje izazovno pronaći izvore pogrešnih promjena.

Dakle, iako definicija apstraktnih vrsta podataka određuje da podaci koji su članovi objekta moraju biti skriveni od klijenata, pojavljuju se mnoge situacije u kojima klijenti

trebaju pristup tim podacima. Rješenje jest osigurati pristupne metode dohvaćanja (eng. get) i postavljanja vrijednosti (eng. set). Iste omogućuju neizravan pristup članovima klase koji su specificirani kao privatni a ipak moraju biti dostupni u drugim dijelovima programa, odnosno izvan same deklaracije klase. Skrivanje podataka predstavlja bolje rješenje od običnog deklariranja javnog specifikatora koji pruža pristup istima u svim dijelovima programa.

Apstraktni tipovi podataka, s enkapsulacijom i kontrolom pristupa, jesu kandidati za ponovnu upotrebu. Problem sa ponovnom upotrebom apstaktnih tipova podataka jest da, skoro u svim slučajevima, značajke i mogućnosti postojećeg tipa nisu sasvim u redu za novu upotrebu. Stari tip zahtijeva barem neke manje izmjene. Takve izmjene mogu biti teške jer zahtjevaju da osoba koja radi modifikacije razumije dio, ako ne i sve dijelove postojećeg kôda. U mnogim slučajevima, osoba koja radi izmjenu nije izvorni autor programa. Nadalje, u mnogim slučajevima, izmjene zahtijevaju promjene u svim programima klijenata (Ibidem).

Ako novi apstraktni tip podataka može naslijediti podatke i funkcionalnost nekog postojećeg tipa, a također je dopušteno izmijeniti neke od tih entiteta i dodati nove entitete, ponovna upotreba uvelike se olakšava bez potrebe za izmjenama na ponovnoj upotrebi apstraktnih podataka. Programeri mogu započeti s postojećim tipom apstraktnih podataka i dizajnirati modificirani potomak koji bi trebao odgovarati novom zahtjevu za problem. Nadalje, nasljeđe pruža okvir za definiranje hijerarhija povezanih klasa koje mogu održavati odnos potomaka u problemskom prostoru (Ibidem).

Tema nasljeđivanja obrađena je u sljedećem podpoglavlju u nastavku rada.

3.2 Nalјeđivanje

Klase su tipovi apstraktnih podataka u objektno-orijentiranim jezicima, a instance klase se nazivaju objekti.

Klasa koja je definirana nasljeđivanjem iz druge klase, izvedena je klasa ili podklasa. Klasa iz koje je izvedena nova klasa je njezina roditeljska klasa ili superklasa. Potprogrami koji definiraju radnje nad objektima klase nazivaju se metode. Metode klasa se ponekad nazivaju poruke. Cijela zbirka metoda nekog objekta naziva se protokolom poruke, ili sučeljem poruka objekta. Računanja u objektno orijentiranom programu određuju poruke poslane od objekta na druge objekte, ili u nekim slučajevima, u klase (Sebesta, 2012).

Izvedena se klasa razlikuje od svojih roditelja. Roditeljska klasa može definirati neke varijable i metode sa privatnim pristupom, što znači da neće biti vidljive u podklasi. Podklasa može dodati varijable i metode onima naslijeđenima iz roditeljske klase. Podklasa može mijenjati ponašanje jedne ili više njenih naslijeđenih metoda. Izmijenjena metoda ima isti naziv, a često je isti protokol kao i onaj koji je modificiran.

Nova metoda nadjačava naslijeđenu metodu koja se tada zove nadjačana metoda. Svrha je pružiti operaciju u podklasi koja je slična onoj u roditeljskoj klasi, ali je prilagođena upravo za taj tip objekta u podklasi.

Ukoliko je nova klasa zapravo podklasa neke roditeljske klase, radi se o jednostrukom nasljeđivanju. Ako klasa ima više od jedne roditeljske klase, proces se zove višestruko nasljeđivanje (eng. multiple inheritance).

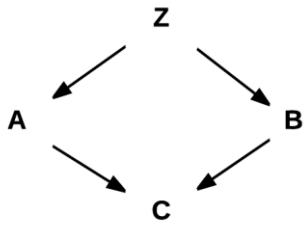
Svrha višestrukog nasljeđivanja je dopustiti novoj klasi da naslijedi svojstva, uključujući varijable i metode, od dviju ili više klasa.

Pitanje je da li svi jezici omogućuju više nasljeđa, te budući da je višestruko nasljeđivanje jako korisno, zašto ne bi dizajner jezika učinio takvo svojstvo mogućim. Očito postoje problemi vezani uz takvu vrstu nasljeđivanja.

Ako klasa ima dvije nepovezane roditeljske klase i iste nemaju, na primjer, istoimene metode, onda nema problema. Međutim, pretpostavimo da klasa pod nazivom C nasljeđuje iz obje klase A i B, i obje klase definiraju naslijeđenu metodu pod nazivom „ispis“. Problem se pojavljuje u trenutku kada klasa C mora referencirati jednu od

dviju verzija naslijeđene metode 'ispis'. Problem je dvosmislen i kompliciran kada obje roditeljske klase definiraju jednako imenovane metode i jedna ili obje moraju biti nadjačane u podklasi.

Drugi se problem javlja u slučaju kada je nova klasa C podklasa dvije roditeljske klase A i B dok su iste izvedene klase jedne roditeljske klase, Z. Ovaj je slučaj poznat pod nazivom dijamantno (eng. diamond) nasljeđivanje kao što je grafički prikazano na sljedećoj slici.



Slika br.4: Dijamantno nasljeđivanje

Korištenje višestrukog nasljeđivanja može lako dovesti do složenih programskih organizacija. Održavanje sustava koji koriste višestruko nasljeđivanje može biti ozbiljan problem iz razloga što dovodi do složenijih zavisnosti među klasama.

Dakle, jedan nedostatak nasljeđivanja kao sredstvo povećanja mogućnosti ponovne upotrebe jest da stvara zavisnost među klasama u hijerarhiji nasljeđivanja. Ovaj rezultat djeluje protiv jedne od prednosti apstraktnih vrsta podataka, odnosno da su neovisne jedna od druge. Naravno, sve apstraktne vrste podataka ne smiju biti potpuno neovisne. Ali općenito, neovisnost tipa apstraktnih podataka jedna je od njihovih najjačih pozitivnih svojstva. Međutim, može biti teško, ako ne i nemoguće, povećati ponovnu upotrebu apstraktnih vrsta podataka bez stvaranja zavisnosti između nekih od njih. Nadalje, u mnogim slučajevima, ovisnosti prirodno održavaju ovisnost u temeljnom problemskom prostoru (Ibidem).

Također, u slučajevima nasljeđivanja postavlja se pitanje alokacije i dealokacije objekata te mjesta iz kojeg su isti alocirani.

Objekti su zapravo apstraktni tipovi podataka kojima se alokacija može dodijeliti sa

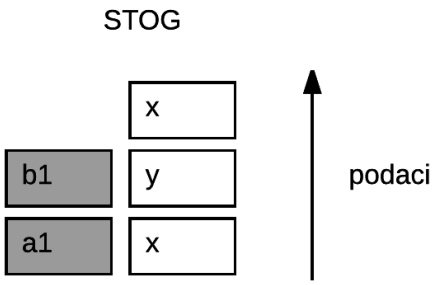
bilo kojeg mjesta kao što je stog za vrijeme izvođenja programa ili hrpa sa naredbom „new“. Dakle, ako se objekti nalaze na dinamičkoj hrpi isti imaju prednost stvaranja putem uniformne metode i dohvaćaju se pokazivačima i referentnim varijablama. Ako se objekti nalaze na dinamičkom stogu, postoji problem sa podtipovima, odnosno podklasama. Ako je klasa B dijete klase A, a B je podtip A, onda objekt tipa B može biti dodijeljen varijabli tipa A. Na primjer, ako je varijabla b1 objekt tipa klase B, a varijabla a1 objekt tipa klase A onda bi se takva dodjela mogla zapisati kao izjava prikazana na slici broj pet (Ibidem).

```
A a1;
B b1;

a1 = b1;
```

Slika br.5: Dodijela

Kao što je već rečeno, problem dodjele se pojavljuje kod dinamičkog stoga, te prethodni primjer može se vizualno pregledati u samoj memoriji stoga kao što je prikazano na slici broj šest.



Slika br.6: Dinamički stog

Ukoliko su a1 i b1 reference objekta koje se nalaze na dinamičkoj hrpi, tada ne postoje problemi ukoliko se jednostavni zadatak rješava pomoću pokazivača. Međutim, ako se a1 i b1 nalaze na dinamičkom stogu, dodijeljena im se vrijednost mora preslikati u prostor ciljnog objekta. Ako B dodaje podatkovno polje na ono što je naslijedilo od A, tada a1 neće imati dovoljno prostora na stogu za b1. Višak će

jednostavno biti skraćen što bi moglo zbuniti programere koji pišu ili koriste kôd. Ovakva se vrsta skraćivanja naziva sjeckanje objekta (eng. object slicing) (Ibidem).

```
class A{
    int x;
}

class B : A{
    int y;
}
```

Slika br.7: Nasljeđivanje klasa

Još jednu stavku vezanu uz skrivanje podataka čine ugniježđene klase. Što znači da, ako je nova klasa potrebna samo jednoj klasi i nema potrebe da se ista definira kako bi je i ostale klase mogle vidjeti, nova se klasa može ugnijezditi unutar klase koja istu koristi. U nekim je slučajevima nova klasa ugniježđena unutar podprograma umjesto izravno u drugoj klasi.

Klasa u kojoj je nova klasa ugniježđena zove se gnijezdo klasa. Najznačajniji problem kod dizajna povezan je sa klasičnim gniježđenjem, odnosno problem je vezan uz samu vidljivost upitne klase. Najme, postavlja se pitanje koji od objekta klase gniježđenja su vidljivi u gniježđenoj klasi te koji od objekta ugniježđene klase moraju biti vidljivi u klasi gniježđenja (Ibidem).

3.3 Polimorfizam

Treća osobina, nakon enkapsulacije i nasljeđivanja, objektno-orijentiranih programskih jezika je polimorfizam koji pruža dinamičko vezanje (eng. dynamic binding) poruka prema definicijama metoda. To se zove dinamičko slanje (eng. dynamic dispatch) (Sebesta, 2012).

Razmatra se sljedeća situacija. Postoji osnovna klasa, A, koja definira metodu „ispis“ koja ispisuje neke informacije povezane sa baznom klasom. Druga klasa, B, je definirana kao podklasa od A. Objekt ove nove klase također ima metodu „ispis“ koja je poput one iz klase A, ali ipak malo drugačija, zato što su podklasni objekti uvijek malo drugačiji. Tako podklasa nadjačava nasljeđenu metodu „ispis“. Ako klijent od A i B ima varijablu koja je referenca na objekte klase A, ta referenca također može ukazivati na klase objekta B, što čini polimorfnu referencu. Ako se metoda „ispis“ koja je definirana u objektu klase poziva kroz polimorfnu referencu, vrijeme rada sustava mora odrediti, tijekom izvršavanja, koju metodu treba pozvati, A ili B, određujući na koji tip objekta trenutno upućuje referenca.

Jedna svrha dinamičkog vezanja jest mogućnost da se softverski sustavi lakše produže tijekom razvoja i održavanja.

U nekim slučajevima, dizajn hijerarhije nasljeđivanja rezultira jednom ili više klasa koje su toliko visoko u hijerarhiji da njihova instanca ne bi imala smisla. Takve se metode nazivaju apstraktne metode ili čiste virtualne metode u C++-u. Klasa koja uključuje barem jednu apstraktnu metodu naziva se apstraktna klasa. Takva se klasa obično ne može instancirati jer su neke njegove metode deklarirane ali nisu definirane, odnosno nemaju svoje tijelo. Svaka podklasa apstraktne klase koja je instancirana mora osigurati definiciju, odnosno implementaciju svih naslijeđenih apstraktnih metoda.

Dakle, prema Motiki i Šribaru (1997), mogućnost promatranja objekta kroz različite klase u stablu nasljeđivanja se zove polimorfizam.

4. Smalltalk

Koncept objektno-orijentiranog programiranja ima svoje korijene u jeziku Simula 67, ali nije bio u potpunosti razvijen dok nije došlo do evolucije Smalltalk 80 jezika upravo 1980. godine. Doista, neki smatraju da je to temeljni model za čisto objektno-orijentirano programiranje.

Smalltalk je bio prvi programski jezik koji je potpuno podržavao objektno-orijentirano programiranje.

Alan Kay je u kasnim 60.-im godinama na Sveučilištu Utah-a vjerovao da će stolna (eng. desktop) računala biti korištena i od ne programera, te isti moraju imati prilagođeno korisničko sučelje (eng. user friendly interface). Kay je odredio da računalo treba biti vrlo interaktivno i koristiti sofisticiranu grafiku u svom korisničkom sučelju (Sebesta, 2012).

Smalltalk svijet je naseljen samo objektima. Svo računanje u Smalltalku se obavlja istom uniformnom tehnikom, odnosno slanje poruke objektu da se pozove na jednu od svojih metoda. Odgovor na poruku je objekt koji vraća traženu informaciju ili jednostavno obavještava pošiljatelja da je obrada zahtjeva dovršena.

Objekt se sastoji od privatne memorije i skupa operacija. Priroda samih objekata ovisi o vrsti, odnosno tipu podataka koju komponenta predstavlja. Objekti koji predstavljaju brojeve računaju aritmetičke funkcije. Predmeti koji predstavljaju strukture podataka pohranjuju i dohvaćaju informacije. Objekti koji predstavljaju pozicije i područja odgovaraju na upite o njihovom odnosu sa drugim pozicijama i područjima (Goldberg i Robson, 1983).

Poruka jest zahtjev za objektom koji će izvršiti jednu od njegovih operacija. Poruka određuje koja će se operacija izvršiti, ali ne kako se ta operacija treba provesti. Objekt kojemu je poruka poslana, određuje kako izvršiti traženu operaciju. Na primjer, dodavanje se vrši slanjem poruke objektu koji predstavlja broj. Poruka navodi da je željena operacija dodavanja i također određuje koji broj treba dodati prijemniku. Poruka ne navodi kako će se izvršiti dodavanje. Prijemnik određuje kako izvršiti dodavanje. Računanje se smatra intrinzičnom sposobnošću objekata koji se mogu ravnomjerno pozivati slanjem poruka (Ibidem).

U programskim jezicima poput C++-a moguće je izvršavati samo dijelove programa koji će izvršavati operacije i vraćati određeni rezultat. Radi se o potprogramima.

Temeljna razlika između poruke i potprogramskog poziva jest da se poruka šalje objektu podataka, konkretno na jednu od metoda definiranih za objekt.

Donošenje poruke zaista je različito od pozivanja potprograma. Potprogram obično obrađuje podatke koje je njegov pozivatelj proslijedio kao parametar ili se istome pristupa ne lokano, odnosno globalno. Poruka poslana objektu je zahtjev za izvršenje jedne od njegovih metoda. Barem dio podataka na kojima metoda treba raditi je sam objekt. Objekti imaju metode koje definiraju procese, odnosno metode koje objekt može izvesti na sebi. Iz razloga što su objekti apstraktne vrste podataka, ovo bi trebao biti jedini način za manipulaciju objektom. Potprogram definira proces koji može obaviti na bilo kojim podacima koji su mu poslani ili dostupni na ne lokalnoj, odnosno globalnoj razini (Sebesta, 2012).

Tada se pozvana metoda izvršava, često mijenjanjem podataka objekta na koji je poruka poslana. Podsustav se šalje kao poruka kôdu potprograma. Obično podaci koji se moraju obraditi u potprogramu šalju se istome putem parametra.

U Smalltalk-u apstrakcije objekata su klase koje su vrlo slične klasama u Simula 67. Instance klasa se mogu stvoriti i tek tada se stvaraju objekti programa.

Sintaksa Smalltalk-a se razlikuje od većine drugih programskih jezika, u velikoj mjeri zbog potrebe slanja poruka, a ne aritmetičkih i logičkih izraza i konvencionalnih kontrolnih izjava.

Mnogi misle o Smalltalk-u kao konačnom objektno-orijentiranom programskom jeziku. Bio je to prvi jezik koji uključuje potpunu podršku za tu paradigmu. Stoga je prirodno započeti istraživanje jezične podrške za objektno-orijentirano programiranje s Smalltalk-om.

Svi objekti imaju lokalnu memoriju, nasljednu sposobnost obrade, mogućnost komuniciranja s drugim objektima, te mogućnost nasljeđivanja metoda i varijabla instance predaka. Klase ne mogu biti ugniježđene u Smalltalk-u.

Svi objekti Smalltalk-a dodjeljuju se iz hrpe i upućuju na iste kroz referentne varijable, koje se implicitno dereferenciraju. Ne postoji izričita izjava o dealokaciji ili operaciji.

Sva odstupanja su implicitna, koristeći postupak sakupljanja smeća za reklamacije prostora u skladištu podataka.

U Smalltalk-u, konstruktori moraju biti izričito nazvani kada se objekt stvara. Klasa može imati više konstruktora, ali svaki mora imati jedinstveno ime.

Za razliku od hibridnih jezika kao što je C++, Smalltalk je dizajniran za samo objektno-orijentiranu paradigmu razvoja softvera. Osim toga, ona ne prihvaća ništa od izgleda imperativnih jezika.

Podklasa Smalltalk-a nasljeđuje sve instance varijable i klasne metode njezine nadklasa. Podklasa također može imati svoje instancirane varijable, koje moraju imati nazive koji su različiti od naziva varijabli iz svoje nadklase. Konačno, podklasa može definirati nove metode i redefinirati metode koje već postoje u klasi predaka. Kada podklasa ima metodu čije je ime i protokol isti kao i klasa predak, metoda podklase skriva se od klase predaka. Pristup skrivenoj metodi dobiva se prefiksnoj poruci s pseudovarijablom „super“. Prefiks uzrokuje traženje metoda za početak u superklasama, a ne lokalno.

Budući da entiteti u roditeljskoj klasi ne mogu biti skriveni od podklasa, sve podklase su podtipovi. Smalltalk podržava jednostruko naslijeđivanje i ne dopušta višestruko nasljeđivanje.

Smalltalk je učinio mnogo za pomicanje dva odvojena aspekta računalsta, a to su grafičko korisničko sučelje i objektno orijentirano programiranje. Sustav prozorčića koji su sada dominantna metoda korisničkih sučelja za softverski sustav su izrasli iz Smalltalk-a.

Danas je najznačajniji dizajn metodologije softvera i programski jezici koji su objektno-orijentirani. Iako podrijetlo nekih ideja o objektno-orijentiranim jezicima je došlo od jezika Simula 67, isti su svoje sazrijevanje postigli u Smalltalk-u. Jasno je da Smalltalk utječe na računalni svijet opsežno i dugoročno.

Na slici broj osam, u nastavku, prikazan je način implementiranja programskog kôda sa osnovnim principima objektno-orijentiranog programiranja u Smalltalk-u.

Ispis programskog kôda izgleda ovako:

„Kućni ljubimac

Bao bao!

Kućni ljubimac

Mijao mijao!“.

```
Object subclass: #Ljubimac.
Ljubimac class extend [
            new [
                <category: 'instance creation'>
            ]
            ispis [
                Transcript show: 'Kućni ljubimac'.
                Transcript cr.
            ]
        ]

Ljubimac subclass: #Pas.
Pas class extend [
            new [
                <category: 'instance creation'>
                super ispis
            ]
            ispis [
                Transcript show: 'Bao bao!'.
                Transcript cr.
            ]
        ]

Ljubimac subclass: #Macka.
Macka class extend [
            new [
                <category: 'instance creation'>
                super ispis
            ]
            ispis [
                Transcript show: 'Mijao mijao!'.
                Transcript cr.
            ]
        ]

bao := Pas new.
bao ispis.
mijao := Macka new.
mijao ispis.
```

Slika br.8: Implementacija programskog kôda u Smalltalk-u

U nastavku rada opisuje se suvremeni objektno-orijentirani programski jezik C++ koji nasljeđuje koncepte Smalltalk-a.

5. C++

Koncept objektno-orijentiranog programiranja predstavlja nadogradnju koncepta strukturnog programiranja čiji je predstavnik programski jezik C. Proširenje C-a predstavlja jezik C++ sa značajkama objektno-orijentiranog programskog jezika.

C++ se razvio od jezika C kroz niz modifikacija za poboljšanje imperativnih svojstva i dodavanje konstruktora za podršku objektno-orijentiranog programiranja.

Prvi korak od C-a do C++-a napravio je Bjarne Stroustrup u Bell Laboratoriju 1980. godine. Početne modifikacije u C-u uključuju funkcije provjeravanja i pretvorbe tipa parametra, te još neka svojstva klasa koje su povezane sa onima iz Simula 67 i Smalltalk-a. Također su uključene i izvedene klase, javna i privatna kontrola pristupa naslijeđenih komponenti, metode konstruktora i destruktora te prijatelja klasa. Zatim su dodani zadaci preopterećenja operatora, parametri i funkcije u redu (eng. inline) čije je tijelo definirano upravo kod poziva iste za veću brzinu obrade (Sebesta, 2012).

Korisno je uzeti u obzir neke ciljeve C-a s klasama, odnosno C++-a. Primarni cilj bio je pružiti jezik u kojem bi se programi mogli organizirati kao što su organizirani u Simula 67, to jest, s klasama i nasljeđivanjem. Do 1984., ovaj je jezik proširen uključivanjem virtualnih metoda koje omogućuju dinamičko povezivanje poziva metode s određenim definicijama metoda, istoimene metode i preopterećenje operatora, te reference. Ova verzija programskog jezika jest C++.

Najvažnije značajke dodane u C++-u, izdanje 2.0, bile su podrška za višestruko nasljeđivanje, odnosno klase sa više od jedne roditeljske klase, i apstraktne klase, zajedno sa nekim drugim poboljšanjima. Dok je izdanje 3.0 1990. godine dodao predloške koji pružaju parametrizirane vrste i rukovanje iznimkama.

Budući da C++ ima funkcije i metode, onda podržava i proceduralno i objektno-orijentirano programiranje.

Operatori u C++-u mogu biti preopterećeni, što znači da korisnik može stvoriti operatore za postojeće operatore na korisnički definirane vrste. C++ metode također mogu biti preopterećene što znači da korisnik može definirati više od jednog načina ili više od jednog imena pod uvjetom da su brojevi ili vrste njihovih parametara različiti.

Dinamičko vezanje u C++-u pruža virtualne metode. Ove metode definiraju operacije

koje su ovisne o vrsti, koristeći preopterećene metode, unutar zbirke klasa koje su povezane nasljeđivanjem. Pokazivač na objekt klase tipa A također može ukazati na objekte klase koje imaju klasu A kao predaka. Kada ovaj pokazivač ukazuje na preopterećenu virtualnu metodu, dinamički se odabire metoda tekućeg tipa (Ibidem).

Obje metode i klase mogu biti predlošci, što znači da oni mogu biti parametri. Na primjer, metoda se može napisati kao predložena metoda i omogućeno je da ima verzije za različite vrste parametara. Klase imaju istu fleksibilnost.

C++ podržava višestruko nasljeđivanje što također uključuje rukovanje iznimkama (eng. exception handling) gdje događaj ne omogućava normalan nastavak rada programa i zahtijevaju posebnu obradu (Radošević, 2007).

U vrijeme kada se prvi put pojavio C++, programiranje orijentirano prema objektu počelo je stvarati interes i široku rasprostranjenost. C ++ je jedini dostupan jezik koji je pogodan za velike komercijalne softverske projekte.

Na negativnoj strani, C ++ je vrlo velik i složen jezik i očito ima svojih nedostataka. Naslijednik je mnogih nesigurnosti C-a koje ga čine manje sigurnim od jezika kao što je Java.

C++ pruža dva konstrukta koja su vrlo slična jedno drugom, a to su klase i strukture koji izravnije podržavaju apstraktne vrste podataka. Strukture se obično koriste kada su uključeni samo podaci. C++ klase predstavljaju tipove podataka kao što je već spomenuto.

U C++-u, programska jedinica deklarira instacu klase i može pristupiti bilo kojem od javnih entiteta u toj klasi, ali samo putem instance klase. Podaci definirani u C++ klasi nazivaju se podatkovni članovi (eng. data members), a funkcije i metode definirane u klasi nazivaju se funkcijski članovi (eng. member functions). Podatkovni članovi i funkcijski članovi pojavljuju se u dvije kategorije, odnosno klase i instance. Članovi klase su povezani s klasom, a instancirani članovi povezani su s instancama klase. Sve instance klase dijele jedan skup funkcijskih članova, ali svaka instanca ima svoj skup podataka koji su članovi klase. Instance klasa mogu biti statičke ili dinamičke kao stog ili hrpa. Ako se radi o dinamičkom stogu onda se referencira direktno s vrijednostima varijabli. Ako se radi o dinamičkoj hrpi onda se referencira putem pokazivača. Instance klasa koje su na dinamičkom stogu nastaju elaboracijom

dealociranog objekta. Instance dinamičke klase hrpe stvaraju se operatorom za stvaranje nove instance „new“ i uništavaju se operatorom za brisanje „delete“. Klase dinamičkog stoga i hrpe mogu imati pokazivačke članove koji upućuju na dinamičku hrpu podataka, tako da iako je klasa instancirana kao dinamički stog, može uključivati članove podataka koji se odnose na dinamičku hrpu podataka (Sebesta, 2012).

Funkcijski član klase može se definirati na dva različita načina, odnosno potpuna definicija može se pojaviti u klasi ili samo u zaglavlju. Kada se funkcijski član nalazi i u zaglavlju i u tijelu to znači da se pojavljuje u definiciji klase, te se funkcijski član implicitno umeće. To znači da se kôd nalazi u pozivateljskom kôdu pa ne zahtijeva uobičajeni poziv i povratno povezivanje. Ako se u klasi pojavi samo definicija zaglavlja funkcijskog člana, njegova potpuna definicija pojavljuje se izvan klase i odvojeno se sastavlja. Postavljanje definicija funkcija člana izvan definicije klase odvaja specifikaciju od implementacije što jest cilj modernog programiranja (Ibidem).

C++ klasa može sadržavati i skrivene i vidljive entitete, što znači da su isti vidljivi ili ne vidljivi klijentima klase. Entiteti koje treba sakriti smješteni su u privatnoj klauzuli, a vidljivi ili javni entiteti pojavljuju se u javnoj klauzuli. Javna klauzula stoga opisuje sučelje za instanciranje klasa.

C++ omogućuje korisniku uključivanje funkcija konstruktora u definicije klase, koje se koriste za inicijaliziranje podatkovnih članova novo stvorenih objekata. Konstruktori se implicitno zovu kad je neki objekt tipa klase stvoren. Konstruktor ima isto ime kao i klasa čiji je objekti inicijaliziran. Konstruktori mogu biti preopterećeni, ali svakako svaki konstruktor klase mora imati jedinstveni profil parametra.

C++ klasa također može uključivati funkciju nazvanu destruktor koji se implicitno zove kada vrijeme trajanja klase završi. Kao što je ranije navedeno, instance klasa na dinamičkom stogu mogu sadržavati članove pokazivače koji se odnose na dinamičku hrpu podataka. Funkcija destruktora za takvu vrstu može uključiti operatera za brisanje na članovima pokazivača kako bi se raspodijelio prostor na hrpi na koju isti upućuje. Destruktori se često koriste kao pomoć pri uklanjanju pogrešaka, u kojima se jednostavno prikazuju ili ispisuju vrijednosti nekih ili svih podataka objekta članova klase, prije nego što su ti članovi raspoređeni. Ime destruktora je ime klase, kojem prethodi znak tilda (~). Ni konstruktori niti destruktori nemaju povratne vrste, niti koriste povratne izjave.

Jedan problem dizajna jezika koji proizlazi iz klasa, ali nema generaliziranu konstrukciju za enkapsulaciju je da ponekad kada su definirane operacije koje koriste dvije različite klase objekata, operacija prirodno ne pripada ni jednoj od tih klasa. Na primjer, pretpostavimo da imamo apstraktnu vrstu podataka za matrice i jednu za vektore i treba nam operacija množenja između vektora i matrica. Kôd množenja mora imati pristup podacima članova oba vektor i klase matrice, ali niti jedna od tih klasa nije prirodan dom za kôd. Nadalje, bez obzira na koji je odabran, pristup članovi druge klase je problem. U C++-u, takvim se situacijama može rukovati dopuštajući nefunkcijskim članovima da budu "prijatelji" (eng. friends) klase. Prijatelj funkcije imaju pristup privatnim subjektima klase u kojima su proglašeni kao prijatelji. Za operaciju umnožavanja matrica ili vektora, jedno C++ rješenje je definirati operaciju izvan matrice i klase vektora, ali definirati da bude prijatelj od oboje.

Prijateljstvo nije uvijek obostrano, ako to nije eksplicitno postavljeno.

Pored funkcija, čitave klase mogu se definirati kao prijatelji klasa. Onda su svi privatni članovi klase vidljivi svim članovima prijateljske klase.

Općenito, programi u C++-u su relativno brzi uspoređujući njemu bliske programske jezike te se koristi za posebno velike softverske projekte, no glavni nedostaci su složenost jezika što ga čini teškim za savladavanje i vrlo opsežnim te nesigurnim naspram jezicima poput Jave.

Na slici broj devet, u nastavku, prikazan je način implementiranja programskog kôda sa osnovnim principima objektno-orijentiranog programiranja u C++-u.

Kao što je već spomenuto u radu, kako bi u C++-u bio omogućen polimorfizam potrebno je definirati apstraktnu osnovnu klasu koja sadrži čistu virtualnu funkciju. Ovakva se klasa ne može instancirati, već se njezina virtualna metoda definira u podklasnim istoimenim metodama koje je nadglašavaju.

Ispis programskog kôda izgleda ovako:

„*Bao bao!*

Mijao mijao!“.

```cpp
#include<iostream>
using namespace std;

class ljubimac{
    public:
        virtual void ispis() = 0;
        virtual ~ljubimac() = 0;
};
ljubimac::~ljubimac(){}

class pas: public ljubimac{
    public:
        void ispis(){
            cout << "Bao bao!" << endl;
        }
};

class macka: public ljubimac{
    public:
        void ispis(){
            cout << "Mijao mijao!" << endl;
        }
};

int main(){

    pas *bao = new pas;
    bao->ispis();

    macka *mijao = new macka;
    mijao->ispis();

    cout << endl;

    delete bao;
    delete mijao;

    system("pause");
    return 0;
}
```

Slika br.9: Implementacija programskog kôda u C++-u

U nastavku rada opisuje se objektno-orijentirani programski jezik Java koji je kreiran iz primjera C++-a. U nekim je kriterijama Java nadmašila C++, međutim, to nije bilo dovoljno ukoliko je C++ i dalje puno brži od Jave.

6. Java

Java dizajneri započeli su s C++-om, uklonili su i promijenili neke konstrukcije i dodali nova svojstva. Rezultirajući jezik pruža mnogo snage i fleksibilnost C++-a, ali u manjem, jednostavnijem i sigurnijem jeziku.

1990. godine Sun Microsystems su utvrdili da postoji potreba za programskim jezikom za ugrađene potrošačke elektroničke uređaje kao što su tosteri, mikrovalne pečnice i interaktivni TV sustavi. Pouzdanost je važna karakteristika softvera u potrošačkim elektrnoničkim proizvodima i jedan je od primarnih ciljeva takvog jezika (Sebesta, 2012).

C++ podržava objektno-orijentirano programiranje, ali je procijenjen kao prevelikim i složenim jezikom, dijelom zbog toga što podržava proceduralno programiranje. Isto tako niti C niti C++ ne pružaju potrebnu razinu pouzdanosti. Dakle, dizajniran je novi jezik pod nazivom Java. Njegov dizajn bio je vođen temeljnim ciljem da pruža veću jednostavnost i pouzdanost od C++-a.

Iako je početna namjena Jave bila potrošačka elektronika, ipak jezik nije bio korišten za takve proizvode u svojim prvim godinama postojanja. Kada je 1993. godine „World Wide Web“ postao široko korišten, uglavnom zbog svojih novih grafičkih preglednika, Java je pronađen kao koristan alat za web programiranje. Konkretno, Java „applet“, koji su relativno mali Java programi koji se interpretiraju u web preglednicima i čiji izlaz može biti uključen u prikazani web-dokument, postali su vrlo popularni sredinom kasnih 1990-ih. U prvih nekoliko godina popularnosti Jave, web je bio najčešća primjena.

Java je objektno-orijentirani programski jezik što znači da svaki program sadrži barem jednu klasu koja omogućava izvršavanje željenih operacija i predstavlja određeni tip podataka. Klase sadržavaju atribute i metode. Metode predstavljaju funkcije koje se izvršavaju unutar klase. Do Java 8 verzija, svaka je funkcija morala biti sadržana unutar neke klase (Fain, 2015).

Kako bi instancirali novu klasu, potrebno je deklarirati određeni tip klase i zatim se koristi operator „new“ za svaku novu instancu klase.

„Svi osnovni tipovi podataka imaju odgovarajuće klase omotače (eng. wrapper classes), koje sadrže korisne metode za rad sa odgovarajućim tipovima podataka.“ (Fain, 2015, str.26).

Takve klase općenito imaju dvije svrhe, odnosno sadrže određeni broj korisnih funkcija za rad sa osnovnim tipovima podataka. Na primer, klasa „Integer“ sadrži korisne metode, kao što su konverzija tipa „String“ u vrijednost „int“, pretvaranje „int“ u vrijednost „float“ i tako dalje. Klasa „Integer“ također omogućava definiranje minimalne i maksimalne vrijednosti određenog tipa. Nadalje, neke Java kolekcije ne mogu spremati vrijednosti osnovnih tipova podataka, kao što je „ArrayList“, tako da osnovni tipovi moraju biti prevedeni u objekte (Ibidem).

Java se temelji na C++-u, ali je posebno dizajniran da bude manji, jednostavniji i pouzdaniji. Poput C++-a, Java ima klase i primitivne vrste. U Javi, polja su slučajevi unaprijed definirane klase, dok u C++-u nisu. Mnogi korisnici C++-a grade klase omotnice za polja kako bi se dodala svojstva poput provjere indeksa, što je zapravo implicitno u Javi.

Java nema pokazivače, no njegove referentne vrste pružaju neke od sposobnosti pokazivača. Ove se reference koriste za upućivanje na klase. Svi su objekti raspoređeni na hrpi. Reference se uvijek implicitno odvajaju, kada je to potrebno. Tako se klase ponašaju više kao obične varijable.

Java ima primitivni „Boolean“ tip podataka koji se uglavnom koristi za kontrolne izraze, odnosno kontrolne izjave kao što su „if“ i „while“. Za razliku od C i C++-a, aritmetički izrazi se ne mogu koristiti za kontrolne izraze.

Jedna značajna razlika između Jave i mnogih njegovih prethodnika koji podržavaju objektno-orijentirano programiranje, uključujući C++, jest to što nije moguće pisanje samostalnih potprograma u Javi. Svi su potprogrami zapravo Java metode i definiraju se u klasama. Nadalje, metode se mogu pozivati kroz klase ili samo preko objekta. Jedna od posljedica toga jest da dok C++ podržava proceduralno i objektno usmjereno programiranje, Java podržava samo objektno-orijentirano programiranje.

Još jedna važna razlika između C++-a i Jave jest da C++ podržava višestruko nasljeđivanje izravno u svojim definicijama klase. Java podržava samo jednostruko nasljeđivanje klasa, nema višestrukog nasljeđivanja.

U Javi se relativno lako mogu stvarati procesi koji se izvršavaju istodobno. Isti se nazivaju niti (eng. threads).

Java koristi implicitnu dealokaciju pohrane podataka za svoje objekte, koja se naziva kolekcija smeća (eng. garbage collection). To oslobađa programera od potrebe za eksplicitnim brisanjem objekata kada isti više nisu potrebni. Programi napisani u jezicima koji nemaju kolekciju smeća često pate od onoga što se zove curenje memorije (eng. memory leaking), što znači da se prostor za pohranu dodjeljuje, ali nikad nije dealociran. To može očito dovesti do eventualnog iscrpljivanja svih dostupnih prostora za pohranu u skladištu podataka.

Za razliku od C i C++-a, Java uključuje korekcije asptraktnih tipova, implicitna vrsta konverzije, na primjer od "manjeg" do "većeg" tipa. Tako se pretvorba iz „int“ u „float“ prisiljava preko operatera zadataka, ali „float“ u „int“ ne.

Početna verzija Java prevoditelja, nazvan „Java Virtual Machine“ (JVM), doista je bio najmanje 10 puta sporiji od ekvivalentnih sastavljenih C programa. Međutim, mnogi su Java programi sada prevedeni na strojni kôd prije izvršavanja, koristeći pravovremenog „Just-in-Time“ (JIT) prevodioca. To čini učinkovitost Java programa konkurentnim onim programiranima na konvencionalno sastavljenim jezicima poput C++-a.

Korištenje Jave povećalo se brže od bilo kojeg drugog programskog jezika. U početku je to bilo zbog svoje vrijednosti u programiranju dinamičkih web-dokumenata. Jedan od razloga, izuzetne važnosti, brzog porasta Jave jest činjenica da programeri koji koriste Javu izuzetno vole svoj dizajn. Neki programeri misle da je C++ jednostavno prevelik i složen da bi bio praktičan i siguran. Java im je ponudio alternativu s mnogo snage C++-a, ali u jednostavnijem, sigurnijem jeziku. Drugi razlog jest da je prevodilac, sustavni interpreter, za Javu slobodan i lako ga je dobiti na mreži (eng. web). Java je danas široko korišten u različitim područjima primjene aplikacija.

Prije Jave 8 verzija, Java 7 pojavila se u 2011. godini. Od svog početka, mnoge su značajke dodane na jezik, uključujući popisivanje klasa, generičkih proizvoda i novog iteracijskog konstrukta.

Java podrška za apstraktne vrste podataka slična je onoj u C++-u. Postoje, međutim,

nekoliko važnih razlika. Svi objekti se dodjeljuju iz hrpe i pristupa im se putem referentnih varijabli. Metode u Javi moraju biti potpuno definirane u klasi. Tijelo metode mora se pojaviti s odgovarajućom metodom iz zaglavlja. Dakle, Java apstraktni tipovi podataka su i deklarirani i definirani u pojedinačnim sintaktičkim jedinicama. Java prevodilac može definirati u redu bilo koji način koji nije nadjačao. Definicije su skrivene od klijenata tako što se proglašavaju privatnima.

Umjesto da se specificiraju privatne i javne klauzule u definicijama klase, Java u svojoj modifikaciji pridružuje metode i varijable u definicijama. U Javi instance varijabla i metoda nemaju pristupn u specifikaciju već imaju paketni pristup.

Jedna je očigledna razlika, a to je nedostatak destruktora u Java verziji, uklonjena Java implicitnom zbirkom smeća.

Java podrška za apstraktne tipove podataka je slična onoj C++-a. Java jasno predviđa ono što je potrebno za dizajnirati apstraktne vrste podataka.

„U objektno-orijentiranim jezicima nasljeđivanje podrazumijeva mogućnost da se nova klasa definira na osnovu postojeće, a ne od početka. Svaka osoba nasljeđuje određene karakteristike od svojih roditelja. Sličan mehanizam postoji i u Java programskom jeziku. Specijalna ključna riječ 'extends' se koristi za ukazivanje da je neka klasa izvedena iz druge klase.“ (Fain, 2015, str.33).

Općenito, u objektno-orijentiranom programiranju, u izvedenim klasama mogu se definirati istoimene metode iz nadklasa, sa istim nazivom i listom argumenta. Radi se o specifikaciji, odnosno korištenje predefiniranih metoda iz nadklasa čiji izvorni kôd nije raspoloživ, ali je potrebno koristiti i specificirati funkcionalnosti metode. Takve predefinirane metode omogućuju polimorfizam.

Polimorfizam u Javi može biti u vremenu kompajliranja i u vremenu izvršavanja, odnosno mogu se preopteretiti statičke metode i poziv može krenuti preko referentne metode u nadklasi.

Na slici broj deset, u nastavku, prikazan je način implementiranja programskog kôda sa osnovnim principima objektno-orijentiranog programiranja u jeziku Java.

```java
public class Ljubimac
{
  public void ispis(){
    System.out.print("Kućni ljubimac!\n");
  }

  public static void main(String[] args)
  {
    Ljubimac c = new Ljubimac();
    c.ispis();

    Pas bao = new Pas();
    bao.ispis();

    Macka mijao = new Macka();
    mijao.ispis();

  }
}

public class Pas extends Ljubimac
{
  public void ispis(){
    System.out.print("Bao bao!\n");
  }
}

public class Macka extends Ljubimac
{
  public void ispis(){
    System.out.print("Mijao mijao!\n");
  }
}

```

Slika br.10: Implementacija programskog kôda u Javi

Ispis programskog kôda izgleda ovako:

„*Kućni ljubimac!*

Bao bao!

Mijao mijao!“.

Pozivi metoda su polimorfni i u jeziku Python, baš kao u Javi, što je obrađeno u nastavku rada.

7. Python

Python je relativno nedavno interpretiran u skladu s predmetom skriptnih programskih jezika. Njegov početni dizajner bio je Guido van Rossum na „Stichting Mathematisch Centrum“ u Nizozemskoj početkom 1990-ih. Sada se negov razvoj pridodaje Python Software Foundation-u.

Skriptni jezici su se razvili tijekom proteklih 25 godina. Rani skriptni jezici su korišteni sastavljanjem popisa naredbi, nazvanih skripta, u datoteku za tumačenje. Prvi od tih jezika, nazvan „sh“ za Shell, započeo je kao mala zbirka naredbi koja je interpretirana kao pozivi na sustav potprograma u kojima se izvršavaju uslužne funkcije, kao što su upravljanje datotekama ili jednostavno filtriranje datoteka. Pri tome su dodane varijable, izjave o kontroli toka, funkcije i razne druge mogućnosti, a rezultat je potpuni programski jezik.

Pythonova sintaksa ne temelji se izravno na bilo kojem uobičajenom jeziku. Sintaksa se provjerava pri pisanju i piše se dinamički. Umjesto polja, Python uključuje tri vrste struktura podataka, a to su popisi ili nepromjenjive liste koje se nazivaju torke (eng. tuples), i mješavine (eng. Hash) koji se nazivaju rječnici (eng. dictionary). Postoji zbirka metoda liste koja uključuje metode za dodavanje, umetanje, uklanjanje i sortiranje, kao i zbirka metoda za rječnike. Python također podržava razumijevanje popisa ili liste koja su nastala s Haskell-ovim jezikom.

Python je objektno-orijentiran, uključuje sposobnosti podudaranja uzoraka Perla, i ima rukovanje iznimkama. Sakupljanje smeća koristi se za povrat objekata kada više nisu potrebni. U Pythonu za podatkovne zbirke poput znakovnog niza (eng. string) i liste postoje odgovarajući operatori, funkcije i metode. Dakle, umjesto tipa podataka rabe se klase, a umjesto naziva koriste se jedinka klase ili objekti.

„*Naime u Pythonu postoje tzv. specijalne metode (eng. special methods) kojima se ostvaruju operatori. Tako se, primjerice, operator + ostvaruje metodom __add__(), a operator * metodom __mul__().*“ (Budin i sur., 2013, str. 3).

Specijalne metode se pišu s dvijema donjim crtama na početku i kraju iste, te osnovni brojači poput „int“ i „float“ također predstavljaju klase sa svojim specijalnim metodama.

„*Dakle, svi su tipovi podataka u Pythonu klase (eng. class), a sve pojedine vrijednosti su jedinke (eng. instances), odnosno objekti (eng. objects) klasa.*“ (Budin i sur., 2013, str. 4).

Metode se definiraju unutar klasa, odnosno metode predstavljaju definiciju klase. Prilikom kreiranja objekta iz klase postavlja se konstruktor koji se izvršava i postavlja početne vrijednosti. Na razini klase, atributi su globalni i mogu se dohvaćati iz bilo koje metode klase.

Python je sažeti jezik koji je čitak samim programerima ukoliko se isti piše na jednostavan način, upravo onako kako bi ga čovjek mogao razumljivo pročitati za razliku od ostalih popularnih programskih jezika.

Python ima visoki potencijal ponovne iskoristivosti upravo radi uporabe metoda i odgovarajućih implementacija.

Poput Smalltalka, Python je objektno-orijentirani jezik u kojemu se klase tretiraju kao objekti, no pored toga podržava i ostale koncepte programiranja poput funkcionalnog programiranja. Funkcije su objekti prvog reda što znači da se mogu pohranjivati u podatkovnim strukturama i prenose povratne vrijednosti potprograma (Kalafatić i sur., 2016).

Python ima široko područje primjene, ali u slučajevima kada procesorsko vrijeme nije kritično, odnosno u slučajevima kada bi najbolje bilo koristiti C++.

Imperativno programiranje (eng. imperative programming) ili imperativna programska paradigma jest pristup kojim se modelira i izgrađuje tehnička naprava prema ljudskim sklonostima kao što je slaganje Lego-kockica.

Python sadrži koncepte proteklih iz funkcijskog ili funkcionalnog programiranja koje u sebi ne sadrži naredbe i koriste čiste izraze (eng. pure expression) koje ne utječu na stanje programa, odnosno grade se isključivo funkcijskim izrazima.

Kao što je već spomenuto, u objektno-orijentiranom programiranju operacije se izvršavaju nad objektima i nazivaju se metode. Prema Kalafatiću, skup svih metoda koje se primjenjuju nad objektom nazivaju se sučeljem objekta (eng. interface). U Pythonu sučelja objekta nazivaju se protokolima (eng. protocol). Sučelje definira sposobnosti objekta, odnosno što se sa njime sve može napraviti.

U Pythonu ime objekta nema nikakvu informaciju vezanu za sami objekt, tip objekta je zapravo njegovo svojstvo, a ne ime. Prema Kalafatiću i sur.(2016), svi pozivi funkcija i metoda u Pythonu su polimorfni te se unaprijed ne može ustanoviti gdje će funkcijski poziv završiti.

„*Java je donekle slična Pythonu utoliko što su i u Javi svi pozivi metoda polimorfni. Međutim, Java je statički tipiziran jezik: svako ime u Javi jednoznačno i neraskidivo je povezano s odgovarajućim tipom koji definira sučelje koje to ime podržava (...) Zbog toga se u Javi puno češće koristimo nasljeđivanjem nego u Pythonu, a ponekad moramo i duplicirati kôd kako bismo uspjeli izraziti svoju namjeru.*“(Kalafatić i sur., 2016, str.329).

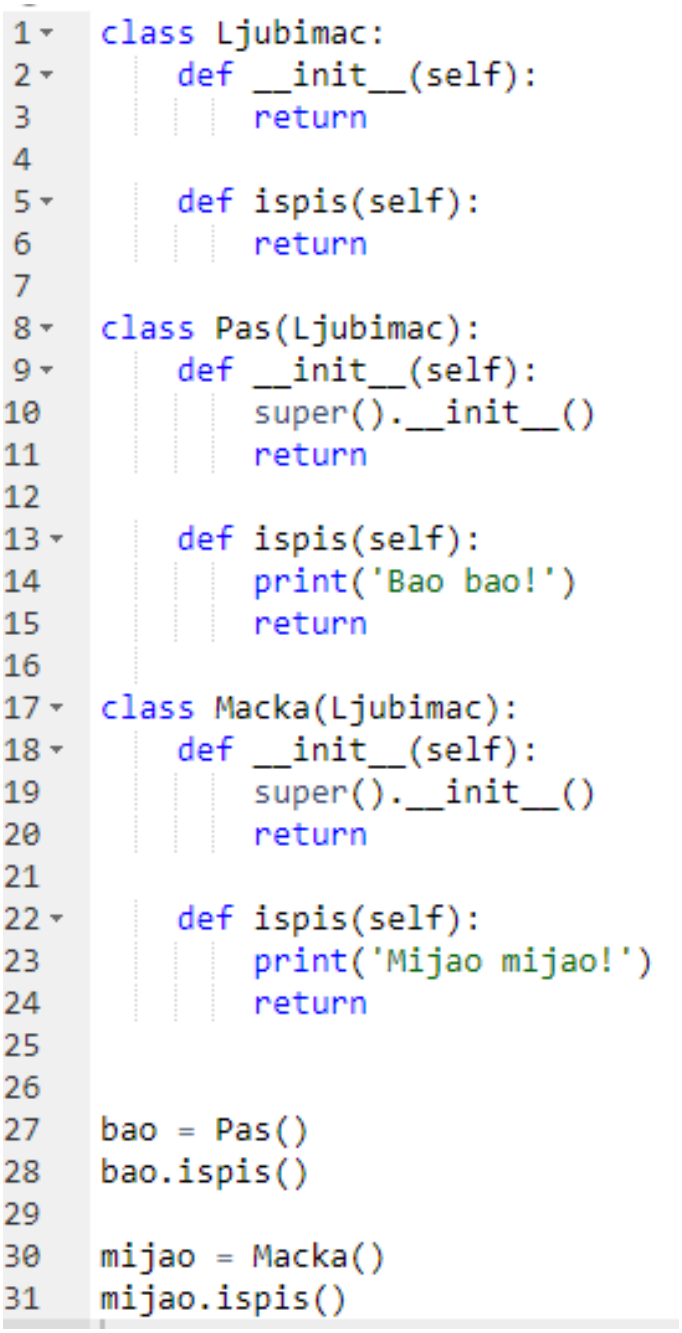
Na slici broj jedanaest, u nastavku, prikazan je način implementiranja programskog kôda sa osnovnim principima objektno-orijentiranog programiranja u jeziku Python.

Python ima svoje super metode, u ovome je slučaju korištena super metoda „__init__“ koja vrijedi kao konstruktor klase.

Ispis programskog kôda izgleda ovako:

„*Bao bao!*

Mijao mijao!“.



```python
class Ljubimac:
    def __init__(self):
        return

    def ispis(self):
        return

class Pas(Ljubimac):
    def __init__(self):
        super().__init__()
        return

    def ispis(self):
        print('Bao bao!')
        return

class Macka(Ljubimac):
    def __init__(self):
        super().__init__()
        return

    def ispis(self):
        print('Mijao mijao!')
        return


bao = Pas()
bao.ispis()

mijao = Macka()
mijao.ispis()
```

Slika br.11: Implementacija programskog kôda u Pythonu

U sljedećem poglavlju nalazi se usporedna analiza objketno-orijentiranog programiranja, odnosno usporedba Smalltalk-a, C++-a, Jave i Python-a.

8. Krajnja usporedba

Smalltalk je temeljni model za čisto objektno-orijentirano programiranje, međutim zastario je ukoliko se objektno-orijentirana paradigma znatno poboljšala u jezicima poput C++-a.

U Smalltalk-u se konstruktori izričito pozivaju i moraju imati jedinstveno ime poput metode. Podržava samo jednostruko nasljeđivanje, no sadrži oblik polimorfizma gdje podklasa može definirati istoimene metode koje već postoje u klasi predaka.

Najočitija razlika između Smalltalk-a i ostalih odabranih jezika jest slanje poruka na metode objekta što je različito od potprograma ostalih programskih jezika koji izvršavaju operacije kao zasebni dijelovi programa koji vraćaju neki rezultat.

Najbitnije svojstvo Smalltalk-a jest sustav prozorčića u korisničkom sučelju, danas dominantna struktura u svim softverskih sustavima.

Svojstvo objektno-orijentiranog programiranja koje uvodi C++ jest skrivanje podataka ili pravo pristupa što korisniku čini djelove programa nevidljivim zajedno s enkapsulacijom, odnosno spajanje strukture podataka sa funkcijama vezane uz klasu koje se onda nazivaju metodama klase.

Uz C++ objektno-orijentirano programiranje dobiva veliku važnost i rasprostranjenost te interes svih programera. Pogodan je za velike komercijalne softverske projekte, no, glavni nedostaci jesu njegova složenost i nesigurnost, dok je glavna vrlina njegova brzina.

C++ ujedno uvodi i višestruko nasljeđivanje te ugniježđene klase.

Java je programski jezik koji od C++-a nasljeđuje snagu i fleksibilnost, no u jednostavnijem i sigurnijem obliku. Najbitnija karakteristika jest upravo pouzdanost ukoliko je jezik zamišljen za uporabu u ugrađenim električnim uređajima iako je njegova najčešća primjena u mrežnom programiranju.

U Javi nije moguće pisanje samostalnih potprograma iz razloga što su svi potprogrami definirani unutar klasa što ih čini metodama klase. Za razliku od C++-a, Java podržava samo objektno-orijentirano programiranje i nema višestruko nasljeđivanje. Činjenica je da programeri koji koriste Javu izuzetno vole svoj dizajn.

Python podržava funkcionalno i objektno-orijentirano programiranje. Ima široko područje primjene no samo u slučajevima kada procesorsko vrijeme izvođenja nije presudno. U takvim bi slučajevima najpogodnije bilo koristiti jezik poput C++-a.

Kroz istraživanja pokazalo se da su programi u Javi sporiji od programa u C++-u uz veće zauzeće memorije.

Pisanje kôda u Pythonu poistovječuje se s slaganjem Lego-kockica što ga čini vrlo elegantnim, modernim i jednostavnim jezikom.

Neki tvrde da Python nije potpuno objektno-orijentiran jezik, no popularnost je stekao radi elegantnog izvornog kôda pa time i brzine razvoja što čini projekte razvijene u Pythonu vrlo uspješnima, no u domenama kada efikasnost programa nije presudna.

Python je jezik s izražajnim i jasnim kôdom. Jednostavno se čita, što nije uvijek slučaj u jezicima poput C++-a i Jave. Zbog primjene funkcionalnog programiranja, Python pruža visoki potencijal ponovne uporabe.

Java i Python su vrlo slični jezici utoliko što su svi pozivi metoda polimorfni u oba slučaja. U C++-u polimorfizam se primjenjuje samo kod metoda koje su označene ključnom riječi „virtual“. No, Java i C++ su statički tipizirani jezici, dok je Python dinamički tipiziran što znači da se u Javi i C++-u često koristi nasljeđivanje i ponavljanje istoga kôda (Kalafatić i sur., 2016).

U nastavku se nalazi tablica sa kriterijama usporedbe između ova četiri objektno-orijentirana programska jezika. Kriteriji su pouzdanost, složenost i procesorska brzina.

Usporedba se vrši ocjenama od jedan do pet. Pet predstavlja da je kriterij zadovoljavajuć, odnosno da je kriterij odlično zadovoljen. Dok jedan predstavlja da kriterij nije zadovoljen.

Naznake sa ocjenama prikazane su u tablici broj jedan, dok su kriteriji usporedbe sa zaključcima prikazani u tablici broj dva u nastavku rada.

Tablica br.1: Naznake usporedbe

Ocjena	Naznaka
1	nedovoljno
2	dovoljno
3	dobro
4	vrlo dobro
5	odlično

Iz analize se zaključuje da je Smalltalk svakako voljen jezik iz prošlosti, no zamijenjen je modernijom sintaksom i poboljšanjima nad glavnim objektno-orijentiranim principima. C++ je složen jezik kojega je teško razumjeti pa je često zamijenjen Javom većinom u mrežnom programiranju i u sustavima gdje je pouzdanost ključna, no po pitanju procesorske brzine C++ uvijek pobjeđuje. Python je jednostavan jezik i pogodan je za projekte koji iziskuju brz razvoj, no relativno je spor.

Tablica br.2: Kriterij usporedne analize

	Pouzdanost	*Složenost	Brzina	**Zaključak**
Smalltalk	4	3.5	3.5	Danas je zastario, ali je voljen jezik.
C++	4	5	5	Složeni, ali brzi.
Java	5	4	3.5	Pouzdan, ali nešto sporiji.
Python	4	3	3.5	Jednostavan i brz razvoj, ali spora procesorska brzina.
Zaključak	Java je najpouzdanija.	C++ je najsloženiji jezik.	C++ je najbrži jezik.	

*Složenost se odnosi na složenost same strukture programskog jezika, odnosno složenost pri pisanju programskog kôda. U ovom slučaju viša ocjena predstavlja negativnu stavku, dok niža ocjena predstavlja jednostavniju i razumljiviju strukturu programskog jezika što pojednostavljuje razvoj projekta.

9. Zaključak

Programiranje je najkreativnija strana interakcije čovjeka sa računalima.

Programskih jezika ima mnogo i svaki od njih ima svoje prednosti i nedostatke što ih čine pogodnima za određeni tip zadatka i nepogodnim za druge.

Objektno-orijentirano programiranje je najmoderniji koncept programiranja čije su najbitnije značajke enkapsulacija, nasljeđivanje i polimorfizam. Ove značajke imaju svoja svojstva koja imaju varijacije u svakom programskom jeziku ukoliko je svaki programski jezik jedinstveno implementiran.

Odabrani objektno-orijentirani programski jezici za usporedbu su Smalltalk, C++, Java i Python.

Smalltalk se smatra prvim programskim jezikom s čistim objektno-orijentiranim programiranjem. C++ je složeni jezik, no revolucionaran ukoliko se koristi za velike komercijalne softverske projekte i glavna mu je značajka brzina. Ujedno, C++ predstavlja jezik koji je činio objektno-orijentirani koncept popularnim i najzastupljenijim. Java je nešto sporija no vrlo pouzdana te koristi se za sustave elektroničkih uređaja i mrežne sustave. Dok je Python elegantan i moderan pa je prema tome jednostavan i zastupljen među programerima.

Za analizu uzeto je u obzir brzina razvoja u pojedinom objektno-orijentiranom programskom jeziku gdje se Python pokazao kao najbolji odabir, jednostavnost razumijevanja i pisanja programskog kôda odnosno složenost samog jezika gdje se C++ pokazao kao najsloženiji od predstavljenih jezika, pouzdanost programskog jezika gdje se ističe Java te brzinu procesnog vremena programa gdje se pokazalo da je jezik C++ svakako brži od jezika Jave.

Bitno je spomenuti da su odabrani jezici zapravo vrlo slični međusobno te imaju mnogo zajedničkih značajki. Razlike su male, ali opseg je dovoljan da utječe na način razvoja projekta.

Iz analize se zaključuje da je neophodno objektivno analizirati pojedine značajke jezika kako bi se odabrao najpogodniji objektno-orijentirani programski jezik koji najviše odgovara pojedinom projektnom zadatku.

Literatura

Budin L., Brođanac P., Markučič Z. i Perić S. (2013) Napredno rješavanje problema programiranjem u Pythonu. Zagreb, Element

Fain Y. (2015) Programiranje Java. Zagreb, Dobar plan

Goldberg A. i Robson D. (1983) Smalltalk-80: the language and its implementation. Palo Alto, Xerox Company

Kalafatić Z., Pošćić A., Šegvić S. i Šribar J. (2016) Python za znatiželjne. Zagreb, Element

Kusalić D. (2010) Napredno programiranje i algoritmi u C-u i C++-u. Zagreb, Element

Lipljin N. (2004) Programiranje 1. Varaždin, TIVA Tiskara Varaždin

Lovrenčić A., Konecki M. i Orehovački T. (2009) 1957-2007: 50 Years of Higher Order Programming Languages, JIOS, Vol. 33, No. 1, 79-150

Motik B. i Šribar J. (1997) Demistificirani C++. Zagreb, Element

Radošević D. (2007) Programiranje 2. Varaždin, TIVA Tiskara Varaždin

W. Sebesta R. (2012) Concepts of Programming Languages. New Jersey, Pearson

Popis slika

Popis tablica

Sažetak

Cilj ovoga rada je usporediti četiri objektno-orijentiranih programskih jezika kako bi se pokazale sličnosti i razlike koje čine programski jezik pogodnim za određeni tip projekta.

Vrlo je bitno odabrati programski jezik na objektivan način bez subjektivnih vjerovanja i prijašnjih iskustva.

Kronološki su odabrani jezici Smalltalk, C++, Java i Python s detaljnom razradom osnovnih principa objektno-orijentiranih programskih jezika, a to su enkapsulacija, nasljeđivanje i polimorfizam.

Smalltalk je odabran kao jedan od prvih pa sa time i zastarijelih objektno-orijentiranih programskih jezika koji čini temelj suvremenih jezika.

Kroz analizu se pokazalo da su programi u Javi sporiji od programa u C++-u uz veće zauzeće memorije, dok je Python popularnost stekao radi elegantnog izvornog kôda pa time i brzine razvoja što čini projekte razvijene u Pythonu vrlo uspješnima, no u domenama kada efikasnost programa nije presudna.

Ključne riječi: Smalltalk, C++, Java, Python, programiranje, objektno-orijentirano programiranje

Abstract

The aim of this thesis is to compare four object-oriented programming languages to show similarities and differences that make the programming language suitable for a particular type of project.

It is very important to choose a programming language in an objective manner without subjective beliefs and previous experiences.

Chronologically selected languages are Smalltalk, C ++, Java and Python with detailed elaboration of the basic principles of object-oriented programming languages, which are encapsulation, inheritance and polymorphism.

Smalltalk has been chosen as one of the first and therefore obsolete, object-oriented programming languages that make up the foundation of modern languages.

The analysis showed that programs in Java are slower than programs in C ++ with a higher memory occupation, while Python gained popularity for the elegant source code, hence the speed of development that makes Python 's projects highly successful, but in domains where the efficiency of the program is not decisive.

Key words: Smalltalk, C++, Java, Python, programming, object-oriented programming