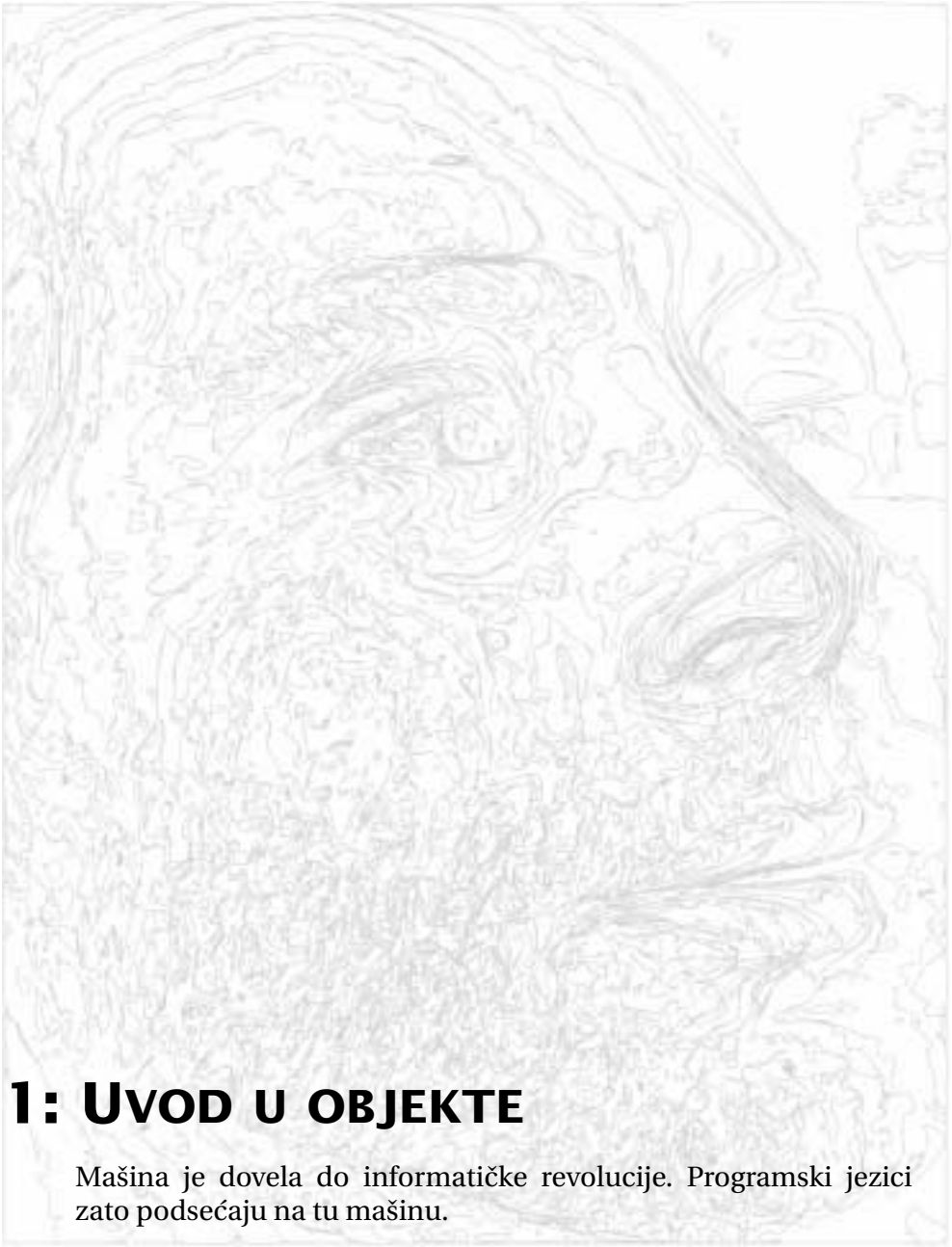


1: UVOD U OBJEKTE

Mašina je dovela do informatičke revolucije. Programski jezici zato podsećaju na tu mašinu.

Računari nisu toliko mašine, koliko su pojačala za misli („bicikli za um“, što bi rekao Steve Jobs) i sredstvo za posebnu vrstu izražavanja. Zbog toga računari sve manje predstavljaju mašine, a sve više deo našeg mišljenja, kao i drugi načini izražavanja poput pisanja, slikanja, vajanja, animacije i filma. Objektno orijentisano programiranje je deo ovog pomeranja ka korišćenju računara kao sredstva izražavanja.

Ovo poglavlje će vas uvesti u osnovne koncepte objektno orijentisanog programiranja (OOP), uključujući i pregled objektno orijentisanih razvojnih metoda. Pretpostavlja se da imate iskustva u radu s proceduralnim programskim jezicima, mada C nije obavezan. Ako vam treba dodatna obuka za programiranje i sintaksu jezika C pre nego što pristupite ovoj knjizi, pogledajte multimedijski seminar „Thinking in C: Foundations for C++ and Java“ na pratećem kompakt disku.

Ovo poglavlje predstavlja podlogu i dopunski materijal. Lakše ćete se otisnuti u vode objektno orijentisanog programiranja ako ga prvo sagledate kao celinu. Iz navedenih ideja steći ćete solidan pregled oblasti OOP-a. Međutim, mnogi ne mogu da sagledaju celinu dok ne vide njene mehanizme i obeshrabre se ako odmah ne počnu da programiraju. Ako pripadate drugoj grupi i ako jedva čekate da saznate karakteristike jezika, slobodno preskočite ovo poglavlje, to vas neće sprečiti da pišete programe, niti da naučite jezik. Možda ćete poželeti da se vratite i da dopunite znanje, kako biste shvatili značaj objekata i način projektovanja.

Razvoj apstrakcije

Svi programski jezici sadrže apstrakcije. Može se reći da je složenost problema koje rešavate direktno povezana s vrstom i kvalitetom apstrakcije. Pod „vrstom“ podrazumevam „Šta apstrahujete?“. Asemblerski jezik je mala apstrakcija mašine na kojoj radi. Mnogi imperativni jezici koji su kasnije nastali (kao što su Fortran, BASIC i C) bili su apstrakcije asemblera. Ovi jezici predstavljaju veliki napredak u odnosu na asembler, ali i dalje zahtevaju da razmišljate o strukturi računara, a ne o strukturi problema koji rešavate. Programer mora da uspostavi vezu između modela mašine (u „oblasti rešenja“ gde modelujete problem, a to je računar) i modela problema koji se rešava (u „oblasti problema“ gde on postoji). Napor neophodan za ovo povezivanje, kao i činjenica da ono nije ugrađeno u programski jezik, proizvodi programe koji se teško pišu i skupo održavaju. Sporedni efekat je nastanak čitave industrije „metoda programiranja“.

Alternativa modeliranju mašine je modeliranje problema koji nastojite da rešite. Prvi jezici, kao što su LISP i APL, zagovarali su specifične poglede na svet („Svi problemi se svode na konačne liste“ ili „Svi problemi su algoritamski“). PROLOG svodi sve probleme na lance odlučivanja. Napisani su jezici za programiranje zasnovano na ograničenjima i za programiranje isključivo grafičkim simbolima. (Dokazano je da su ovi poslednji previše restriktivni.) Svaki od ovih pristupa predstavlja dobro rešenje za konkretnu klasu problema koju treba da rešava, ali su teško upotrebljivi izvan svojih domena.

Objektno orijentisani pristup ide korak dalje, pruža programeru sredstva za predstavljanje elemenata iz oblasti problema. Predstavljanje je dovoljno uopšteno, pa programer nije ograničen na određeni tip problema. Elemente u oblasti problema i njihovo predstavljanje u oblasti rešenja nazivamo „objekti". (Biće vam svakako potrebni i drugi objekti koji ne potiču iz oblasti problema.) Ideja je da se, dodavanjem novih tipova objekata, program prilagođava specifičnom jeziku problema, tako da pri čitanju koda koji opisuje rešenje, čitate i tekst koji izražava problem. Ovo je fleksibilnija i moćnija jezička apstrakcija od prethodnih. OOP vam omogućava da opišete problem njegovim terminima, a ne terminologijom računara na kome će se program izvršavati. I dalje postoji veza sa računarom. Svaki objekat je kao mali računar: ima svoje stanje i operacije koje može izvršavati na zahtev. Analogija sa objektima u svetu oko nas nije tako loša: svi imaju osobine i ponašanja.

Neki projektanti jezika zaključuju da objektno orijentisano programiranje ne omogućava jednostavno rešavanje svih programerskih problema i zalažu se za kombinaciju različitih pristupa u programskim jezicima *sa više paradigmi*.[1]

Alan Kay je sažeto prikazao pet osnovnih osobina Smalltalka, prvog uspešnog objektno orijentisanog jezika i jednog od jezika na kojima je C++ zasnovan. Ove osobine predstavljaju potpuni pristup objektno orijentisanom programiranju:

1. **Sve je objekat.** Mislite o objektu kao o drugačijoj vrsti promenljive: on čuva podatke, ali mu možete i „slati zahteve", tražeći da sam izvrši operacije. Teorijski, možete uzeti bilo koju konceptualnu komponentu problema koji rešavate (psi, zgrade, usluge itd.) i predstaviti je u programu kao objekat.
2. **Program je kolekcija objekata koji, porukama, jedan drugom saopštavaju šta da rade.** Zahtev se objektu šalje kao poruka. Poruku shvatite kao poziv funkcije koja pripada određenom objektu.
3. **Svaki objekat ima sopstvenu memoriju sačinjenu od drugih objekata.** Drugim rečima, od postojećih objekata možete napraviti novu vrstu objekta. Na taj način povećavate složenost programa, ali je skrivate jednostavnošću objekata.
4. **Svaki objekat ima tip.** U stručnom žargonu, svaki objekat je *primerak ili instanca klase*, pri čemu je „klasa" sinonim za „tip". Najznačajnija odlika klase je „Koje poruke joj se mogu poslati".
5. **Svi objekti određenog tipa mogu primati iste poruke**. Ovo tvrđenje je neprecizno, kao što ćete videti kasnije. Pošto je svaki objekat tipa „krug" istovremeno i tipa „oblik", krug može da prima i poruke za oblik. To znači da možete napisati kôd koji se odnosi na oblik i automatski obrađuje sve što odgovara opisu nekog oblika. Ova *zamenljivost* je jedan od najmoćnijih koncepata OOP-a.

[1] Videti *Multiparadigm Programming in Leda*, Timothy Budd, Addison-Wesley, 1995.

Objekat ima interfejs

Aristotel je verovatno prvi započeo pažljivo proučavanje pojma *tip*: govorio je o „klasi riba i klasi ptica“. U prvom objektno orijentisanom programskom jeziku, Simula-67, čija osnovna rezervisana reč **class** (klasa) uvodi nov tip u program, prvi put je neposredno iskorišćena ideja da su svi objekti jedinstveni, ali su i delovi klase objekata sa zajedničkim osobinama i ponašanjem.

Simula je, kao što samo ime jezika govori, napisana radi razvoja simulacija poput klasičnog „bankarskog problema“.[2] U bankama imate mnoštvo bankarskih službenika, klijenata, računa, transakcija i novčanih jedinica – veliki broj „objekata“. Objekti koji su identični po svemu osim po unutrašnjem stanju tokom izvršavanja programa, grupišu se u „klase objekata“, i otuda potiče rezervisana reč **class**. Formiranje apstraktnih tipova podataka (klasa) je osnovna tehnika objektno orijentisanog programiranja. Apstraktni tipovi podataka funkcionišu skoro potpuno isto kao tipovi ugrađeni u jezik: možete napraviti promenljive određenog tipa (u objektno orijentisanom žargonu nazivaju se *objekti* ili *instance*) i raditi s njima (to je *slanje poruka* ili *zahteva*, pri čemu objekat otkriva šta da radi na osnovu poruke koju ste mu poslali). Članovi (elementi) svake klase imaju nešto zajedničko: svaki račun ima saldo, svaki službenik može primiti ulog itd. Istovremeno, svaki član ima sopstveno stanje, svaki račun ima drugačiji broj, svaki službenik ima ime. Zato se službenici, klijenti, računi, transakcije itd. mogu predstaviti kao jedinstveni entiteti u programu. Ovaj entitet je objekat, a svaki objekat pripada određenoj klasi koja definiše njegove osobine i ponašanja.

Iako u objektno orijentisanom programiranju pravimo nove tipove podataka, svi objektno orijentisani programski jezici koriste termin „klasa“. Kada vidite reč „tip“, pomislite na „klasu“, i obrnuto.[3]

Pošto klasa opisuje skup objekata koji imaju identične osobine (podatke) i ponašanja (funkcije), klasa je zapravo tip podataka, zato što, na primer, i broj u formatu pokretnog zareza poseduje skup osobina i ponašanja. Razlika je u tome što programer definiše klasu koja odgovara problemu i nije prinuđen da koristi postojeće tipove podataka, koji samo opisuju način beleženja informacija u računaru. Programski jezik proširujete dodavanjem tipova podataka prema svojim potrebama. Programski sistem prihvata nove klase i na isti način radi s njima, kao i sa ugrađenim tipovima.

Objektno orijentisani pristup nije ograničen na simulacije. Bez obzira na to da li se slažete da je svaki program simulacija sistema koji projektujete – korišćenjem OOP tehnika, veliki skup problema se lako uprošćava i rešava.

Kada definišete klasu, možete napraviti proizvoljno mnogo njenih objekata i raditi s njima kao da su elementi problema koji rešavate. Jedan od izazova objektno orijentisanog programiranja svakako je uspostavljanje jednoznačnog preslikavanja između elemenata u oblasti problema i objekata u oblasti rešenja.

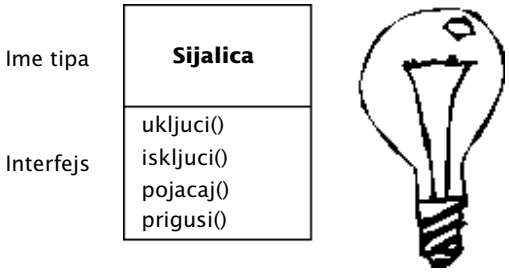
Kako da iskoristite objekat? Mora postojati način da objektu uputite zahtev da nešto uradi, na primer da izvrši transakciju, nacrta nešto ili uključi prekidač. Svaki objekat ispunjava samo određene zahteve. Zahtevi koje možete poslati

---

[2] Zanimljivo rešenje ovog problema naći ćete u tomu 2 ove knjige, koji se preuzima s adrese www.BruceEckel.com

[3] Neki prave razliku, tvrdeći da tip određuje interfejs, dok je klasa konkretna realizacija tog interfejsa.

objektu definisani su njegovim *interfejsom* (engl. *interface*) koji je određen tipom. Jednostavan primer predstavlja sijalica:




```
Sijalica lt;
lt.ukljuci();
```

Interfejs definiše zahteve koje možete uputiti određenom objektu. Međutim, negde mora postojati kôd koji odgovara na taj zahtev. Kôd, zajedno sa skrivenim podacima, čini *realizaciju* (engl. *implementation*). Sve ovo nije teško razumeti sa stanovišta proceduralnog programiranja. Tip poseduje funkciju pridruženu svakom od mogućih zahteva, i zahtevi se upućuju objektu pozivanjem tih funkcija. Ovaj postupak se ukratko može opisati kao „slanje poruke“ (upućivanje zahteva) objektu, koji određuje šta da učini s tom porukom (izvršava kôd).

Ime tipa/klase je **Sijalica**, ime konkretnog objekta je **lt**, a objektu klase **Sijalica** možete uputiti zahteve da se uključi, isključi, pojača ili priguši. Objekat klase **Sijalica** pravite deklarisanjem imena (**lt**) za taj objekat. Da biste poslali poruku objektu, navedite ime objekta i tačkom ga povežite s porukom. Posmatrano sa stanovišta korisnika unapred definisane klase, to je sve što se tiče programiranja objekata.

U prikazanom dijagramu koristi se zapis *objedinjenog jezika modeliranja* (engl. *Unified Modeling Language, UML*). Svaka klasa je predstavljena pravougaonikom, sa imenom tipa u gornjem delu, podacima članovima navedenim u središnjem delu i *funkcijama članicama* (funkcije koje pripadaju ovom objektu i primaju sve poruke koje mu šaljete) u donjem delu pravougaonika. U UML dijagramima se često prikazuju samo ime klase i javne funkcije članice, tako da se središnji deo izostavlja. Ako vas zanima samo ime klase, tada se ne mora prikazati ni donji deo.

Skrivena realizacija

Korisno je podeliti razvojni tim u *projektante klasa* (oni koji prave nove tipove podataka) i *programere klijente*[4] (korisnike klasa, koji koriste ove tipove podataka u svojim aplikacijama). Cilj programera klijenta je da prikupi kolekciju klasa radi brzog razvoja aplikacije. Cilj projektanta klasa je da napravi klasu koja prikazuje samo ono što je neophodno programeru klijentu, a sve ostalo skriva. Zašto? Zato što programer klijent ne može koristiti ono što je skriveno, što znači da projektant klase može da menja skriveni deo ne vodeći računa o uticaju na

[4] Ovaj termin je izmislio moj prijatelj Scott Meyers.

druge delove. Skriveni deo obično predstavlja osetljivi sadržaj objekta, koji programer klijent lako može da ošteti usled nepažnje ili neznanja, tako da skrivanje realizacije umanjuje broj grešaka u programu. Skrivanje realizacije je izuzetno važno.

U svakom odnosu je važno postaviti granice koje poštuju svi učesnici. Kada pravite biblioteku, uspostavljate odnos sa programerom klijentom koji pravi aplikaciju koristeći vašu biblioteku, ili možda pravi veću biblioteku.

Ako su svi članovi klase dostupni svima, tada programer klijent može učiniti bilo šta s klasom i ne postoji nikakav način da se prisili da poštuje pravila. Iako vam više odgovara da programer klijent ne rukuje neposredno nekim članovima klase, ne postoji način da to sprečite bez kontrole pristupa. Sve je dostupno svakome.

Pristup treba kontrolisati da bi se programeri klijenti držali dalje od delova koje ne bi trebalo da diraju – delova neophodnih za unutrašnje operacije nad tipom podataka, koji ne pripadaju interfejsu potrebnom za rešavanje konkretnih problema. To je istovremeno i usluga korisnicima, jer lako mogu uočiti šta je značajno, a šta se može zanemariti.

Kontrolom pristupa, projektantu biblioteke se omogućuje menjanje unutrašnjeg rada klase, bez obzira na uticaj promene na programera klijenta. Na primer, možete na jednostavan način realizovati neku klasu, a zatim otkriti da morate da je ubrzate. Ako su interfejs i realizacija jasno odvojeni i zaštićeni, ovo se može lako rešiti i zatim samo treba zahtevati od korisnika da ponovo poveže projekat.

C++ koristi tri rezervisane reči za definisanje granica klase: **public** (javni), **private** (privatni) i **protected** (zaštićeni). Njihova upotreba i značenje su sasvim jasni. Ovi *specifikatori pristupa* određuju ko može da koristi naredne definicije. Reč **public** znači da su naredne definicije svima dostupne. Rezervisana reč **private** označava da niko osim projektanta tipa ne može pristupati članovima, a i to je moguće samo unutar funkcije članice tog tipa. Reč **private** postavlja visoki zid između vas i programera klijenta. Ako neko pokuša da pristupi privatnom članu, dobiće grešku prilikom prevođenja. Reč **protected** se ponaša kao **private**, izuzev što izvedena klasa ima pristup zaštićenim, ali ne i privatnim članovima. Izvođenje klasa nasleđivanjem biće opisano ubrzo.

Ponovna upotreba realizacije

Kada se klasa jednom definiše i testira, trebalo bi da, u idealnom slučaju, predstavlja korisnu celinu koda. Ispostavlja se da ponovno korišćenje nije ni približno lako postići kako se mnogi nadaju – potrebno je iskustvo i znanje da bi se napravilo dobro rešenje i tada ono sâmo traži da bude ponovo iskorišćeno. Ponovna upotreba koda je jedna od najvećih prednosti objektno orijentisanih jezika.

Najjednostavnije ćete ponovo upotrebiti klasu kada neposredno koristite objekat te klase, a možete ga smestiti i u novu klasu. Ovo zovemo „definisanje objekta člana“. Nova klasa može sadržati proizvoljan broj drugih objekata, u bilo kojoj kombinaciji koja vam treba za postizanje potrebne funkcionalnosti. Pošto sastavljate novu klasu od postojećih, ovaj koncept se naziva *kompozicija* (ili,

opštije, *agregacija*). Za kompoziciju se kaže da je relacija „sadrži“, kao u primeru „Kola sadrže motor“.

(Popunjeni romb gornjem UML dijagramu ukazuje na kompoziciju, što omogućava da postoje samo jedna kola u relaciji. Uglavnom ću koristiti jednostavniji oblik: liniju bez romba, što označava asocijaciju.)[5]

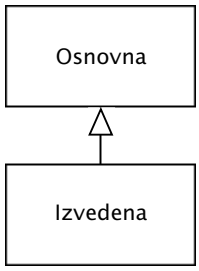
Kompozicija je veoma fleksibilna. Objekti članovi nove klase obično su privatni, što ih čini nedostupnim programerima klijentima koji koriste tu klasu. Možete ih promeniti bez narušavanja postojećeg klijentovog koda. Objekti članovi se mogu menjati i tokom izvršavanja, čime se dinamički menja ponašanje programa. Nasleđivanje, koje se opisuje u nastavku, nije toliko fleksibilno, pošto prevodilac mora postaviti ograničenja na izvedene klase.

Pošto je nasleđivanje tako značajno u objektno orijentisanom programiranju, često se posebno naglašava, i programer novajlija može pomisliti da nasleđivanje treba svuda koristiti. To dovodi do nespretnih i preterano komplikovanih rešenja. Pri definisanju novih klasa, prvo razmotrite kompoziciju koja je jednostavnija i fleksibilnija. Vaši projekti će biti čistiji ako prihvatite ovaj pristup. Kada steknete malo iskustva, biće vam sasvim očigledno kada treba primeniti nasleđivanje.

Nasleđivanje: ponovna upotreba interfejsa

Objekat je sam po sebi zgodna alatka. Omogućava vam da grupišete podatke i funkcije prema *konceptu*, tako da pri opisivanju oblasti problema ne morate koristiti idiome mašine. Ovi koncepti su osnovni elementi programskog jezika i definišu se pomoću rezervisane reči **class** (klasa).

Šteta je ako prođemo kroz sve nevolje definisanja jedne klase, da bismo zatim bili prinuđeni da napravimo potpuno novu klasu sa sličnom funkcionalnošću. Bilo bi bolje kada bismo uzeli postojeću klasu, klonirali je, a zatim dodavali i menjali elemente klona. To, u suštini, dobijate *nasleđivanjem* (engl. *inheritance*), samo što izmene originalne klase (nazvane *osnovna* klasa, *natklasa* ili *roditeljska*) utiču i na „klon“ (*izvedena* klasa, *potklasa* ili *potomak*).



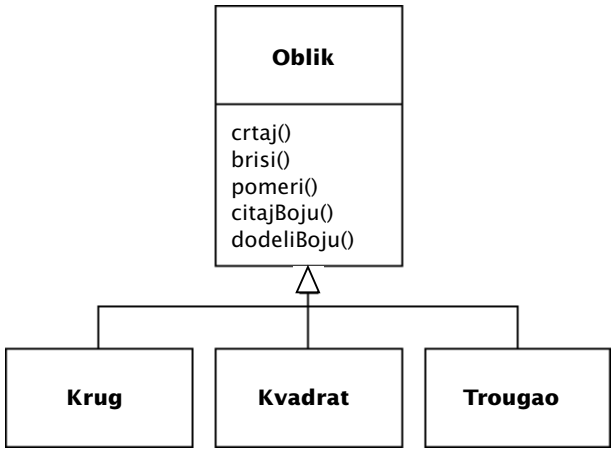
[5] Ovo je obično dovoljno detaljno za većinu dijagrama, pa ne morate precizirati da li koristite agregaciju ili kompoziciju.

(Strelica na prethodnom UML dijagramu usmerena je od izvedene klase ka osnovnoj. Kao što ćete videti, može postojati više izvedenih klasa.)

Tip je nešto više od opisa ograničenja definisanih za skup objekata – on ima relacije s drugim tipovima. Dva tipa mogu imati zajedničke osobine i ponašanja, ali jedan tip može imati više osobina i može obrađivati više poruka (ili obrađivati iste poruke na drugi način). Nasleđivanje izražava ovu sličnost između tipova korišćenjem osnovnih i izvedenih tipova. Osnovni tip sadrži sve osobine i ponašanja koje imaju i tipovi izvedeni iz njega. Osnovni tip treba da predstavlja jezgro vaših ideja o nekim objektima u sistemu. Iz osnovnog tipa izvodite ostale tipove koji izražavaju različite načine realizacije tog jezgra.

Na primer, aparat za recikliranje razvrstava delove otpada. Osnovni tip je „otpad“ i svaki komad otpada ima težinu, vrednost itd. i može se usitniti, otopiti ili razložiti. Iz ovoga mogu biti izvedeni posebni tipovi otpada s dodatnim karakteristikama (flaša ima boju) ili ponašanjima (aluminijum se može savijati, čelik se može namagnetisati). Osim toga, neka ponašanja mogu biti različita (vrednost hartije zavisi od vrste i stanja). Primenom nasleđivanja, možete izgraditi hijerarhiju tipova koja izražava problem.

Drugi klasični primer su oblici, korišćeni u sistemima za projektovanje pomoću računara ili za simulaciju igara. Osnovni tip je „oblik“, a svaki oblik ima veličinu, boju, položaj i slično. Svaki oblik se može nacrtati, izbrisati, pomeriti, obojiti itd. Iz ovoga se mogu izvesti posebni tipovi oblika: krug, kvadrat, trougao, i ostali, od kojih svaki može imati dodatne osobine i ponašanja. Određeni tipovi se mogu, na primer, osno preslikati. Neka ponašanja mogu se razlikovati, kao što je slučaj sa izračunavanjem površina oblika. Hijerarhija tipova obuhvata i sličnosti i razlike među oblicima.



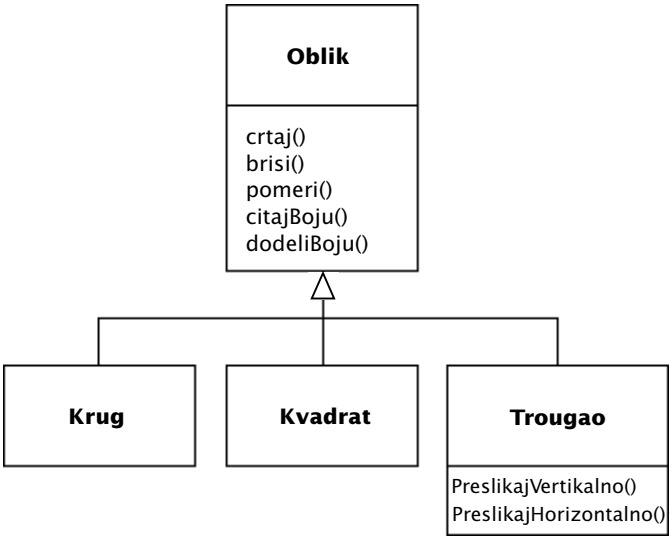
Korišćenje istog rečnika za formulisanje rešenja i problema naročito je zgodno zato što vam ne treba mnoštvo razvojnih modela da biste od opisa problema došli do opisa rešenja. Kada koristite objekte, hijerarhija tipova je primarni model, tako da od opisa sistema u stvarnom svetu neposredno stižete do opisa sistema u

kodu. Čudno zvuči, ali ljudi mogu imati problema s objektnim orijentisanjem zato što je jednostavno. Um, izvežban da traži složena rešenja, može zbuniti ovolika jednostavnost.

Nasleđivanjem postojećeg tipa, pravite nov tip. Ovaj novi tip sadrži sve članove postojećeg tipa (iako su privatni članovi skriveni i nedostupni), a što je još značajnije, sadrži i duplikat interfejsa osnovne klase. Znači da sve poruke koje možete poslati objektima osnovne klase takođe možete slati objektima izvedene klase. Pošto tip klase prepoznajemo po porukama koje joj možemo poslati, znači da je izvedena klasa *istog tipa kao osnovna klasa*. U prethodnom primeru, „krug je oblik". Ova ekvivalentnost tipova dobijenih nasleđivanjem, ključna je za razumevanje objektno orijentisanog programiranja.

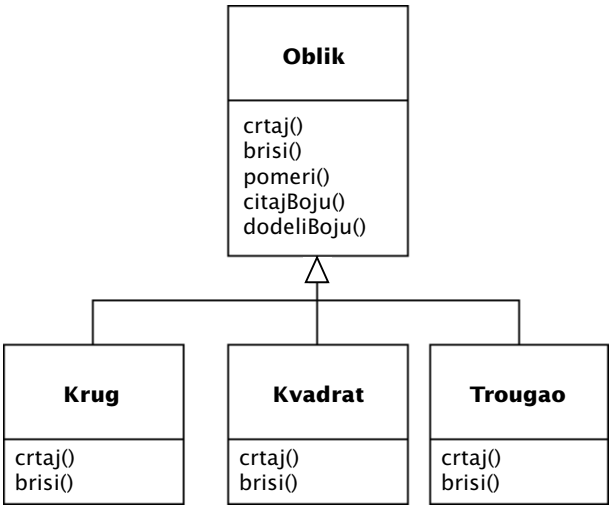
Kako i osnovna i izvedena klasa imaju isti interfejs, mora postojati realizacija uz taj interfejs. Drugim rečima, mora postojati kôd koji se izvršava kada objekat primi određenu poruku. Ako samo izvedete klasu i ne učinite ništa više od toga, metode iz interfejsa osnovne klase se prenose u izvedenu klasu. Znači da objekti izvedene klase imaju isti tip, ali i isto ponašanje, što i nije naročito zanimljivo.

Postoje dva načina da napravite razlike između nove izvedene klase i izvorne osnovne klase. Prvi je prilično jednostavan: dodajte nove funkcije izvedenoj klasi. Ove nove funkcije nisu deo interfejsa osnovne klase. Znači da osnovna klasa nije radila sve ono što ste želeli, pa ste dodali nove funkcije. Ova jednostavna upotreba nasleđivanja je ponekad savršeno rešenje problema. Pažljivo razmotrite da li su ove dodatne funkcije potrebne i osnovnoj klasi. Ovaj proces otkrivanja i iterativnog projektovanja uobičajen je za objektno orijentisano programiranje.



Nasleđivanje samo ponekad podrazumeva dodavanje novih funkcija interfejsu. Drugi i značajniji način da učinite novu klasu drugačijom jeste *promena*

ponašanja funkcije postojeće osnovne klase. To se naziva *redefinisanje* (engl. *overriding*) funkcije.



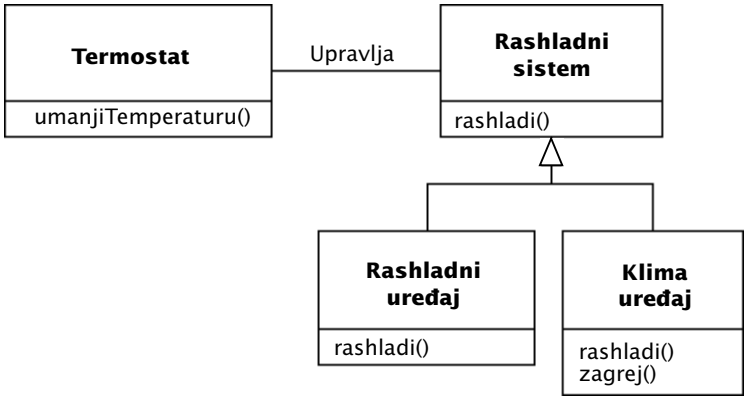
Da biste redefinisali funkciju, uvedite novu definiciju funkcije u izvedenu klasu. Kažite: „Koristim istu funkciju iz interfejsa, ali želim da uradi nešto drugo u novom tipu".

Relacije „je" i „je-kao"

Pri razmatranju nasleđivanja može nastati dilema: da li nasleđivanje treba *samo* da redefiniše funkcije osnovne klase (ne i da dodaje nove funkcije članice koje ne postoje u osnovnoj klasi)? To bi značilo da je izvedeni tip *potpuno* isti kao tip osnovne klase, pošto imaju isti interfejs. U tom slučaju, objekat izvedene klase možete zameniti objektom osnovne klase. Ovo se smatra *potpunom zamenom* (engl. *pure substitution*) i često se naziva *princip zamene* (engl. *substitution principle*). U izvesnom smislu, to je savršen način posmatranja nasleđivanja. Relaciju između osnovne klase i izvedenih klasa tada nazivamo relacijom *je* (engl. *is-a*), pošto možete reći „krug *je* oblik". Nasleđivanje ćete ispitati ako utvrdite smislenu relaciju „je" između klasa.

U nekim situacijama morate dodati nove elemente interfejsu izvedenog tipa: proširićete interfejs i tako stvoriti nov tip. Novi tip još uvek može biti zamenjen osnovnim tipom, ali ta zamena nije savršena pošto nove funkcije nisu dostupne iz osnovnog tipa. Ovaj slučaj se može opisati relacijom *je-kao* (engl. *is-like-a*) – novi tip ima interfejs starog tipa, ali sadrži i druge funkcije, tako da ne možete reći da su potpuno isti. Na primer, razmotrimo rashladni uređaj. Pretpostavimo da je vaša kuća opremljena svim elementima za hlađenje – znači, ima interfejs koji vam omogućava da upravljate hlađenjem. Zamislite da se rashladni uređaj pokvari i da ga zamenite klima-uređajem koji može i da greje i da hladi. Klima-uređaj *je-kao*

rashladni uređaj, ali ima veće mogućnosti. Kako je sistem za regulisanje temperature u vašoj kući projektovan da upravlja samo hlađenjem, on je ograničen na komunikaciju s rashladnim delom novog objekta. Interfejs novog objekta je proširen, a postojeći sistem i dalje prepoznaje samo prvobitni interfejs.



Naravno, kada pogledate ovo rešenje, postaje jasno da osnovna klasa „rashladni sistem“ nije dovoljno opšta i treba da bude preimenovana u „sistem za regulisanje temperature“, tako da može da uključi i grejanje. Tada se može primeniti i princip zamene. Gornji dijagram je primer onoga što se može desiti pri projektovanju i u stvarnom svetu.

Kada razmotrite princip zamene, pomislićete da se samo primenom ovog principa može obaviti posao i zaista *jeste* dobro ako imate takvo rešenje. Međutim, otkrićete situacije kada je jasno da morate dodati nove funkcije interfejsu izvedene klase. Oba slučaja su prihvatljiva.

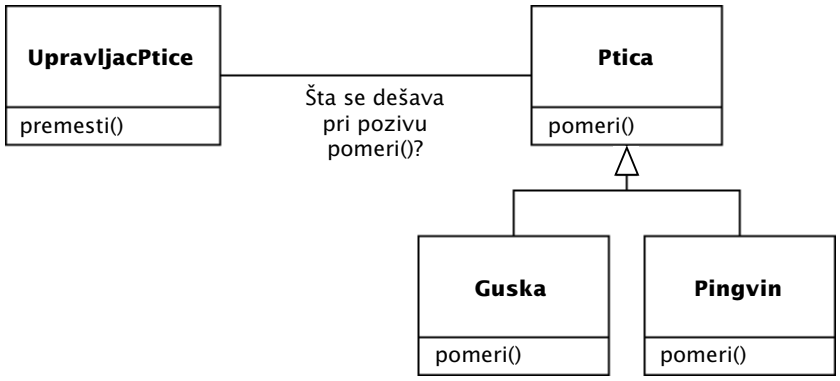
Zamenljivi objekti i polimorfizam

Pri radu s hijerarhijama tipova, često treba postupati s nekim tipom objekta kao sa njegovim osnovnim tipom. To vam omogućava da pišete kôd koji ne zavisi od konkretnih tipova. U primeru oblika, funkcije rade s opštim oblicima bilo da su krugovi, kvadrati, trouglovi ili nešto drugo. Svi oblici se mogu nacrtati, izbrisati, pomeriti, tako da ove funkcije jednostavno šalju poruku objektu oblika i ne vode računa kako će objekat izaći na kraj s tom porukom.

Dodavanje novih tipova ne utiče na takav kôd i to je najčešći način proširenja objektno orijentisanog programa radi obrade novih slučajeva. Na primer, možete izvesti nov podtip oblika, petougao, bez izmene funkcija koje rade samo sa opštim oblikom. Mogućnost jednostavnog proširenja programa izvođenjem novih podtipova je značajna, pošto se u velikoj meri poboljšavaju rešenja i pojevtinjuje održavanje softvera.

Problem nastaje kada pokušate da radite sa objektima izvedenih tipova kao sa opštim osnovnim tipovima (sa krugovima kao oblicima, biciklima kao vozilima, kormoranima kao pticama itd.). Ako neka funkcija zatraži da se opšti oblik nacrta

ili da se opštim vozilom upravlja ili da se opšta ptica pomeri, prevodilac ne zna tačno koji deo koda će se izvršavati. Suština je u tome što pri slanju poruke programer ne *želi* da zna koji deo koda će se izvršavati: funkcija za crtanje može se podjednako primeniti na krug, kvadrat ili trougao i objekat će izvršiti odgovarajući deo koda, u zavisnosti od tipa. Ako ne morate znati koji deo koda će se izvršavati, kôd u novom podtipu može biti drugačiji, pri čemu se ne menja poziv funkcije. Šta prevodilac radi kada ne zna koji deo koda se izvršava? Na primer, objekat **UpravljacPtice** radi samo sa opštim objektima klase **Ptica** i ne zna kog su tipa. Iz perspektive objekta **UpravljacPtice** to je pogodno, zato što se ne mora pisati poseban kôd za prepoznavanje konkretnog tipa **Ptice** s kojom se radi, niti njenog ponašanja. Kako je moguće da poziv funkcije **pomeri()** dovodi do ispravnog ponašanja iako nije poznato kog je tipa **Ptica** (**Guska** trči, leti ili pliva, a **Pingvin** trči ili pliva)?



Odgovor na ovo pitanje je prvi preokret u objektno orijentisanom programiranju: prevodilac ne može pozvati funkciju na tradicionalni način. Poziv funkcije koji pravi proceduralni prevodilac dovodi do *ranog povezivanja* (engl. *early binding*), o kome možda niste čuli zato što nikada niste ni razmišljali o alternativama. Znači da prevodilac poziva konkretnu funkciju po imenu, a povezivač razrešava poziv pomoću apsolutne adrese koda koji treba izvršiti. Objektni program ne može da odredi adresu koda do vremena izvršavanja, tako da treba drugim načinom poslati poruku opštem objektu.

Objektno orijentisani jezici rešavaju taj problem primenom *kasnog povezivanja* (engl. *late binding*). Kôd koji treba pozvati pri slanju poruke objektu, određuje se tek u vreme izvršavanja. Prevodilac obezbeđuje da ta funkcija postoji i proverava saglasnost tipova argumenata i povratne vrednosti (jezik u kome ovo ne važi je *slabo tipiziran*), ali ne zna tačno koji kôd će se izvršavati.

Da bi se ostvarilo kasno povezivanje, C++ prevodilac postavlja specijalni deo koda umesto apsolutnog poziva. Ovaj kôd izračunava adresu tela funkcije korišćenjem informacija smeštenih u objekat (postupak je detaljno opisan u poglavlju 15). Svaki objekat se ponaša različito, u zavisnosti od sadržaja ovog specijalnog dela koda. Kada pošaljete poruku objektu, on tada odlučuje šta će uraditi s porukom.

Rezervisana reč **virtual** označava da funkcija treba da ima fleksibilnost kasnog povezivanja. Ne morate razumeti kako taj mehanizam radi da biste ga koristili, ali ga morate koristiti da biste primenili objektno orijentisano programiranje u C++-u. Podrazumeva se da funkcije članice *nisu* dinamički povezane, pa to svojstvo morate izričito zahtevati navođenjem rezervisane reči **virtual**. Virtuelne funkcije vam omogućavaju da izrazite razlike u ponašanju klasa iste porodice. Ove razlike dovode do polimorfnog ponašanja.

Razmotrimo primer oblika. Porodica klasa (sve su zasnovane na istom interfejsu) prikazana je grafički ranije u ovom poglavlju. Da bismo prikazali polimorfizam, napisaćemo deo koda koji zanemaruje posebne detalje tipa i odnosi se samo na osnovnu klasu. Ovaj kôd je *odvojen* od informacija svojstvenih tipu i zato se jednostavnije piše i lakše razume. Kada se novi tip, na primer **Sestougao**, doda putem nasleđivanja, napisani kôd će raditi dobro s novim tipom, kao što je radio i s postojećim tipovima. To znači da je program *proširiv*.

Funkcija na C++-u (uskoro ćete saznati kako da je napišete):

```
void uradiNesto(Oblik& s) {
  s.brisi();
  //...
  s.crtaj();
}
```

obraća se bilo kom objektu tipa **Oblik** i ne zavisi od konkretnog tipa objekta koji se crta i briše (**'&'** znači „Uzmi adresu objekta koji se prosleđuje funkciji“, ali u ovom trenutku nije bitno da razumete detalje). U nekom drugom delu programa možemo da koristimo funkciju **uradiNesto()**:

```
Krug c;
Trougao t;
Linija l;
uradiNesto(c);
uradiNesto(t);
uradiNesto(l);
```

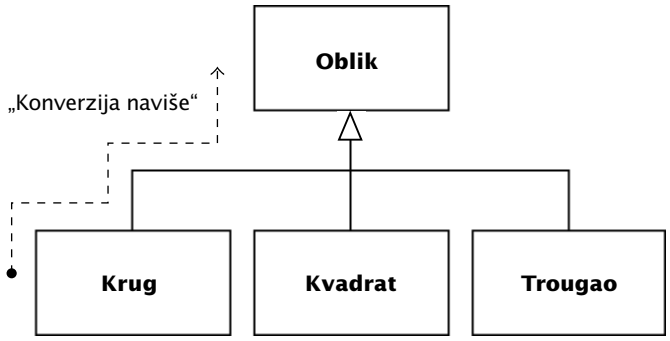
Funkcija **uradiNesto()** radi ispravno, bez obzira na konkretan tip objekta.

Ovo je stvarno začuđujući trik. Razmotrimo sledeći red:

```
uradiNesto(c);
```

Krug se prosleđuje funkciji koja očekuje objekat tipa **Oblik**. **Krug** *je* **Oblik**, pa ih funkcija **uradiNesto()** posmatra na isti način. **Krug** može da primi svaku poruku koju **uradiNesto()** može da pošalje objektu tipa **Oblik**. Znači, ovo je sasvim bezbedno i logično.

Kada sa objektom izvedenog tipa radimo kao da je osnovnog tipa, postupak nazivamo *konverzija naviše* (engl. *upcasting*). Termin *konverzija* (engl. *cast*) odnosi se na promenu oblika, a *naviše* (engl. *up*) na način uređenja dijagrama nasleđivanja, sa osnovnim tipom na vrhu i lepezom izvedenih tipova ispod njega. Konverzija u osnovni tip je pomeranje naviše u dijagramu nasleđivanja, tj. „konverzija naviše“.



Objektno orijentisani programeri obično koriste konverziju naviše, jer tada ne moraju znati s kojim konkretnim tipom rade. Pogledajmo kôd u **uradiNesto()**:

```
s.brisi();
//...
s.crtaj();
```

Zapazimo da u kodu ne postoji: „Ako je **Krug**, neka uradi ovo, ako je **Kvadrat**, neka uradi ono itd.“. Ako pišete kôd koji proverava sve moguće tipove kojima može pripadati **Oblik**, on postaje zamršen i morate ga menjati svaki put kada dodajete novu vrstu oblika. Ovde se samo kaže: „Objekat je oblik, znam da može da uradi **brisi()** i **crtaj()**, neka uradi to, i pobrine se da sve bude ispravno“.

U funkciji **uradiNesto()**, zanimljivo je to što se, na neki način, sve odvija kako treba. Poziv **crtaj()** za **Krug** izvršava drugačiji kôd nego kada se pozove za **Kvadrat** ili **Trougao**, ali kada se poruka **crtaj()** pošalje nepoznatom obliku, ponašanje je takođe ispravno, prema tipu oblika. Ovo je zapanjujuće zato što, kao što je ranije rečeno, C++ prevodilac ne zna tačno s kojim tipovima radi dok prevodi funkciju **uradiNesto()**. Očekivali biste da prevodilac pozove verziju funkcija **brisi()** i **crtaj()** za **Oblik**, a ne za određeni **Krug**, **Kvadrat** ili **Trougao**. Razlog ispravnog ponašanja je polimorfizam. Time se bave prevodilac i izvršni sistem. Bitno je da to znate i, što je još važnije, da poznajete način korišćenja ovih mogućnosti u projektovanju. Ako je funkcija članica deklarisana kao **virtual**, objekat će se ponašati pravilno kada mu pošaljete poruku, čak i ako je primenjena konverzija naviše.

Inicijalizacija i uništavanje objekata

U tehničkom smislu, OOP je oblast apstraktnih tipova podataka, nasleđivanja i polimorfizma, ali i drugi aspekti mogu biti podjednako značajni. Ovde se razmatraju ta druga važna pitanja.

Način inicijalizacije i uništavanja objekata posebno je značajan. Gde se nalaze podaci objekta i kako se određuje njegov životni vek? U različitim programskim jezicima primenjeni su različiti pristupi ovim pitanjima. C++ smatra da je efikasnost najvažnija, tako da izbor ostavlja programeru. Da bi se postigla maksimalna brzina izvršavanja, smeštaj i životni vek se mogu definisati tokom pisanja programa, postavljanjem objekata na stek ili u statičku memoriju. Stek je oblast

memorije koju mikroprocesor neposredno koristi za čuvanje podataka tokom izvršavanja programa. Promenljive na steku ponekad se nazivaju *automatske* (engl. *automatic*) ili *oblasne* (engl. *scoped*). Statička memorija je fiksni deo memorije, koji se rezerviše pre početka izvršavanja programa. Upotrebom steka ili statičke memorije daje se prednost brzini zauzimanja i oslobađanja prostora, što može biti značajno u nekim situacijama. Međutim, time se umanjuje fleksibilnost, zato što morate tačno znati količinu, životni vek i tip objekata *tokom* pisanja programa. Ako rešavate opštiji problem, kao što je projektovanje primenom računara, upravljanje skladištem ili vazdušnim saobraćajem, ovaj pristup je previše restriktivan.

Drugi pristup je pravljenje objekata u *dinamičkoj memoriji* (engl. *heap*). U ovom slučaju, do vremena izvršavanja se ne zna koliko je objekata potrebno, koliki je njihov životni vek, ni kog su tipa. Odluke se donose u trenutku izvršavanja. Kada vam zatreba objekat, pravite ga u dinamičkoj memoriji, pomoću rezervisane reči **new** (novi). Po završetku upotrebe memorije, ona se mora osloboditi korišćenjem rezervisane reči **delete** (briši).

Pošto se memorijom upravlja dinamički tokom izvršavanja, rezervisanje prostora traje značajno duže od zauzimanja prostora na steku. (Zauzimanje prostora na steku često se ostvaruje mikroprocesorskom instrukcijom koja pomera pokazivač steka naniže; druga instrukcija pomera pokazivač naviše.) Dinamički pristup polazi od uopštene logike da su objekti složeni, tako da dodatno vreme za rezervisanje i oslobađanje memorije nema značajnog uticaja na ukupno vreme potrebno za inicijalizaciju objekta. Neophodna je i veća fleksibilnost za rešavanje opštih problema u programiranju.

Sledeće pitanje je životni vek objekta. Ako inicijalizujete objekat na steku ili u statičkoj memoriji, prevodilac određuje koliko će dugo objekat postojati i može ga automatski uništiti. Međutim, ako inicijalizujete objekat u dinamičkoj memoriji, prevodilac ne zna koliki je životni vek objekta. Programer mora da odluči kada da uništi objekat i ovu operaciju izvršava upotrebom rezervisane reči **delete**. Postoji i drugo rešenje: *skupljač smeća* (engl. *garbage collector*). On automatski otkriva objekte koji se više ne koriste i uništava ih. Pisanje programa koji koriste skupljanje smeća je svakako mnogo pogodnije, ali zahteva da sve aplikacije tolerišu prisustvo skupljača i dodatno procesorsko vreme koje mu je potrebno. Ova mogućnost nije bila u skladu s osnovnim zahtevima C++-a i zato nije bila uključena direktno u jezik, iako za C++ postoje skupljači smeća nezavisnih proizvođača.

Obrada izuzetaka: rad s greškama

Obrada grešaka je jedan od najsloženijih postupaka pri programiranju, još od nastanka programskih jezika. Pošto je toliko teško napraviti dobru šemu za obradu grešaka, mnogi jezici zanemaruju ovaj zahtev. Problem se prenosi projektantima biblioteka, pa oni daju polovična rešenja primenljiva na mnoge situacije, koja se lako mogu zanemariti. Dok programer koristi šemu za obradu podataka, mora veoma pažljivo pratiti odgovarajuće konvencije koje nisu obavezne na nivou samog jezika. Ako programeri nisu pažljivi, što se često dešava u žurbi, lako mogu zaboraviti ove šeme.

Obrada izuzetaka (engl. *exception handling*) neposredno povezuje obradu grešaka sa programskim jezikom, nekada čak i sa operativnim sistemom. Izuzetak je objekat koji se „baca“ (engl. *throw*) s mesta gde je greška, a „hvata“ (engl. *catch*) ga odgovarajući *blok za obradu izuzetaka* (engl. *exception handler*). Obrada grešaka je drugačiji, paralelni pravac izvršavanja, kojim se kreće kada stvari pođu nizbrdo. Pošto se koristi posebni tok izvršavanja, nije potrebno da on utiče na ostatak koda. Takav kôd se jednostavnije piše, pošto ne morate neprekidno da proveravate greške. Osim toga, izbačeni izuzetak nije kôd greške koji vraća funkcija, niti indikator koji funkcija definiše kako bi ukazala na nastanak greške – oni se mogu i zanemariti. Izuzetak se ne može ignorisati, pa se garantuje da će na nekom mestu biti obrađen. Konačno, izuzeci obezbeđuju pouzdan oporavak od loših situacija. Umesto izlaska iz programa, često možete popraviti stvari i nastaviti izvršavanje programa, što daje mnogo robusnije sisteme.

Važno je primetiti da obrada izuzetaka nije karakteristika objektno orijentisanog programiranja, ali je uobičajeno da se u objektno orijentisanim jezicima izuzetak predstavlja objektom. Obrada izuzetaka je postojala i pre objektno orijentisanih jezika.

Obrada izuzetaka se samo površno uvodi i koristi u ovom tomu. Tom 2 u potpunosti pokriva obradu izuzetaka.

Analiza i projektovanje

Objektno orijentisani pristup je novi i drugačiji način razmišljanja o programiranju i mnogi teško savladavaju OOP projekte. Kada shvatite da se svaka stvar smatra objektom, i dok učite da sve više razmišljate na objektno orijentisani način, možete početi da pravite dobre projekte koji koriste sve prednosti koje OOP nudi.

Metodologija je skup postupaka i heuristika primenjenih da se razloži složenost programskog problema. Mnoge OOP metodologije su definisane od nastanka objektno orijentisanog programiranja. Ovo izlaganje će vam stvoriti osećaj za probleme koji se rešavaju primenom metodologija.

S metodologijama se puno eksperimentiše, naročito u OOP-u, tako da je važno razumeti koji problem metodologija pokušava da reši pre nego što odlučite koju ćete usvojiti. Ovo posebno važi za C++, programski jezik namenjen pojednostavljenju (u poređenju sa C-om) pisanja programa. Tako se i umanjuje potreba za još složenijim metodologijama. Široka klasa problema u C++-u obrađuje se jednostavnijim metodologijama nego u proceduralnim jezicima.

Važno je razumeti da je izraz „metodologija“ često pretežak i previše obećava. Metodologija je sve što radite dok projektujete i pišete neki program. To može biti vaša metodologija, čijih specifičnosti niste ni svesni, ali to je postupak kroz koji prolazite dok stvarate. Ako je postupak efikasan, možda se treba samo neznatno prilagoditi C++-u. U slučaju da niste zadovoljni svojom produktivnošću i konačnim rezultatom, možda ćete usvojiti neku formalnu metodologiju ili izabrane delove većeg broja formalnih metodologija.

Najvažniji zahtev dok prolazite kroz postupak razvoja jeste: nemojte se izgubiti. To se lako može dogoditi. Većina metodologija analize i projektovanja je

namenjena rešavanju najvećih problema. Zapamtite da najveći broj projekata ne spada u kategoriju velikih problema, tako da analizu i projektovanje možete uspešno izvesti primenom malog podskupa preporuka koje propisuje metodologija.[6] Bilo koji postupak, koliko god bio ograničen, izvodi vas na pravi put mnogo brže nego ako odmah počnete da pišete program bez projekta.

Lako je upasti u zamku, u „paralizu usled analize", kada osećate da ne možete napredovati pošto niste razrešili sve pojedinosti tekuće faze. Bez obzira na to koliko analizirate, neke stvari o sistemu nećete otkriti sve do faze projektovanja. Još više ćete saznati kad počnete da pišete program, ili tek kad se program aktivira. Zato je veoma važno proći prihvatljivom brzinom kroz analizu i projektovanje i testirati planirani sistem.

Pošto imamo iskustva s proceduralnim jezicima, naglašavamo: pohvalno je da tim pažljivo radi i shvati svaki detalj pre početka projektovanja i realizacije. Svakako je pri izradi sistema za upravljanje bazama podataka neophodno potpuno razumeti potrebe kupca. Ovaj sistem pripada kategoriji dobro postavljenih i dobro shvaćenih problema, a u mnogim takvim programima struktura baze podataka *jeste* problem nad kojim se treba zamisliti. U ovom poglavlju se razmatra klasa programerskih problema iz skupa „džokera" (moj izraz). Do rešenja ovih problema se ne dolazi preoblikovanjem nekog dobro poznatog rešenja. U problem se uključuje jedan ili više „nepoznatih činilaca" – elemenata za koje ne postoji prethodno dobro razrađeno rešenje. Neophodno je istraživanje.[7] Pokušaj da se džokerski problem sveobuhvatno analizira pre prelaska na projektovanje i realizaciju dovodi do paralize u analizi, pošto za rešavanje ovakve vrste problema ne stičete dovoljno informacija tokom faze analize. Rešavanje ovakvog problema zahteva iteraciju kroz ceo ciklus i prihvatanje rizika (ima smisla, pošto pokušavate da uradite nešto novo i eventualna nagrada je veća). Može izgledati da rizik potiče od „uletanja" u realizaciju. Upravo ranim otkrivanjem podobnosti konkretnog pristupa problemu smanjujete riskantnost džokerskog projekta. Razvoj programskog proizvoda uključuje i rizik od neuspeha.

Često se predlaže da „napravite jedan projekat da biste ga bacili". U OOP-u možete odbaciti neki deo, ali kako je kôd kapsuliran u klasama, tokom prve iteracije ćete neminovno napraviti neke korisne klase i doći do vrednih ideja o sistemu, koje ne bi trebalo odbaciti. Na taj način, prvi brzi prolazak kroz problem pruža značajne informacije za sledeću iteraciju analize, projektovanja i realizacije, i pravi osnovni kôd za tu iteraciju.

Ako tražite metodologiju koja obuhvata mnoštvo detalja i preporučuje mnogo koraka i dokumenata, svakako će vam biti teško da znate kada treba da se zaustavite. Imajte u vidu šta pokušavate da otkrijete:

1. Šta su objekti? (Kako delite projekat na sastavne komponente?)
2. Šta su njihovi interfejsi? (Koje poruke treba da šaljete svakom objektu?)

---

6 Odličan primer je *UML Distilled*, Martin Fowler, Addison-Wesley, 2000, koji redukuje ponekad preopširan UML proces.

7 Moje pravilo za procenu takvih projekata glasi: ako postoji više džokera, nemojte ni pokušavati da planirate koliko dugo će trajati projekat, niti koliko će koštati, sve dok ne napravite radni prototip. Postoji previše nivoa slobode.

Kada dobijete spisak objekata i njihovih interfejsa, možete napisati program. Verovatno će vam trebati više opisa i dokumentacije, ali je ovo sigurno neophodno.

Proces se može odvijati u pet faza, dok se u nultoj fazi samo uvodi određena struktura.

Faza 0: Pravimo plan

Na početku morate odlučiti koje korake ćete preduzeti u radu. Zvuči jednostavno (zapravo, *sve* ovo zvuči jednostavno), pa ipak ljudi često ne donesu ovu odluku pre početka programiranja. Ako je vaš pristup „hajde da odmah napišem program“, to je u redu. (Nekada je ovo prihvatljivo ako rešavate mali i ispravno shvaćen problem.) Priznajte da je i takav pristup neka vrsta plana.

U ovoj fazi možete odlučiti i da treba dodatno strukturirati proces, ali ne za ceo put koji treba preći. Razumljivo je da ima programera koji rade u „opuštenom režimu“, u kome se postupak razvoja ne planira unapred: „Biće gotovo kad bude gotovo“. Takav stav može biti privlačan u kraćem periodu. Otkrio sam da nekoliko kontrolnih tačaka na putu deluje stimulativno i usredsređuje napor na te manje ciljeve, pa niste okupirani jednim jedinim ciljem da „završite projekat“. Projekat je tako podeljen u manje celine i izgleda manje strašno (osim toga, manji ciljevi stvaraju više prilika za proslave).

Kada sam počeo da proučavam pisanje (znači da ću jednom napisati roman), imao sam otpor prema ideji o strukturi. Osećao sam da ono što pišem jednostavno sleće na stranicu. Kasnije sam shvatio da je struktura dovoljno jasna kada pišem o računarima, tako da o njoj i ne razmišljam mnogo. Ipak, i dalje strukturiram svoj rad, makar i samo polusvesno. Čak i ako mislite da je vaš plan da jednostavno započnete programiranje, vi ipak prolazite kroz sledeće faze dok postavljate izvesna pitanja i nalazite odgovore.

Definisanje zadatka

Svaki sistem koji gradite, koliko god da je komplikovan, ima osnovnu namenu, instituciju kojoj pripada, osnovne potrebe koje ispunjava. Ako pogledate dalje od korisničkog okruženja, detalja specifičnih za hardver ili sistem, programskih algoritama i problema efikasnosti, eventualno ćete otkriti jednostavnu i otvorenu suštinu njegovog postojanja. Slično takozvanoj *osnovnoj zamisli* holivudskih filmova, suština se može opisati u jednoj ili dve rečenice. Polazna tačka je čisti opis osnovne ideje.

Osnovna ideja je veoma važna zato što daje ton vašem projektu: to je definicija zadatka. Možda je nećete naći na samom početku (ponekad se do nje dolazi tek u kasnijoj fazi projekta), ali pokušavajte dok ne osetite da je ideja prava. Na primer, u sistemu za upravljanje vazdušnim saobraćajem, možete krenuti od osnovne ideje da se treba usredsrediti na sistem koji pravite: „Program kontrolnog tornja prati letelice“. Razmotrite šta se dešava kada suzite sistem na veoma

mali vazdušni prostor: možda postoji samo jedan kontrolor ili ga i nema. Korisniji model se ne odnosi na vaše konkretno rešenje, već opisuje problem: „Letelica sleće, iskrcava se, servisira se, ukrcava se i odleće“.

Faza 1: Šta pravimo?

U prethodnoj generaciji programskih projekata (pod nazivom *proceduralno projektovanje*), ova faza je nazvana *analiza zahteva* i *definicija sistema*. Na ovim mestima smo se mogli izgubiti: obeshrabrivali su nazivi dokumenta koji su, sami po sebi, mogli postati veliki projekti. Namera je, međutim, bila dobra. Analiza zahteva kaže: „Napravimo listu vodećih principa koje ćemo koristiti da bismo znali kada je posao završen, a kupac zadovoljan“. Definicija sistema kaže: „Ovde je opis onoga *šta* će program raditi (ne i *kako*) da bi ispunio zahteve“. Analiza zahteva je zapravo ugovor između vas i korisnika (čak i ako korisnik radi u vašem preduzeću ili je korisnik neki drugi objekat ili sistem). Definicija sistema je rezultat ispitivanja problema na najvišem nivou i otkrivanja da li se posao može uraditi i za koje vreme. Pošto će oba dokumenta zahtevati konsenzus među ljudima (i pošto će se dokumenti verovatno vremenom menjati), mislim da je najbolje, radi uštede vremena, da dokumentacija bude što jednostavnija, najbolje u vidu lista i osnovnih dijagrama. Druga ograničenja će možda zahtevati proširenja, ali ako je početni dokument sažet, može se napraviti u nekoliko sesija intenzivnog kolektivnog rada, sa rukovodiocem koji opisuje rešenja. Na taj način se od svakoga zahteva učešće, ali se i podstiče zalaganje na samom početku i usaglašavanje tima. Možda je najvažnije poletno započeti projekat.

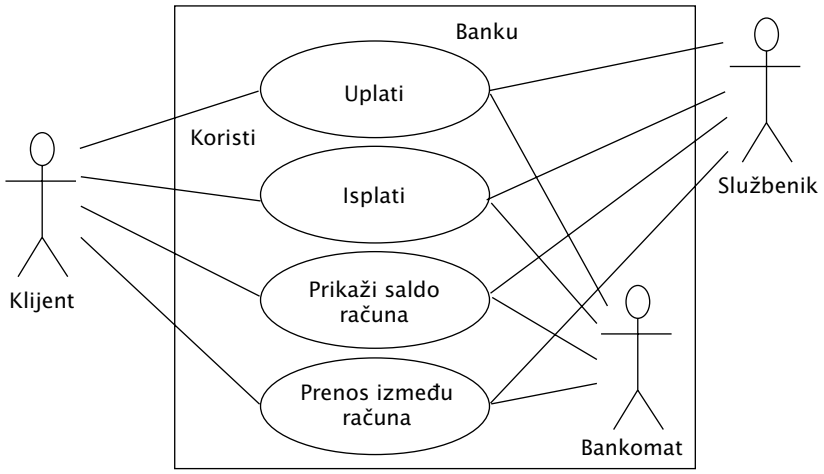
Neohodno je ostati usredsređen na ono što pokušavate da rešite u ovoj fazi, a to je ustanovljivanje šta sistem treba da radi. Najvrednije sredstvo u tome je skup *slučajeva korišćenja* (engl. *use cases*). Slučajevi korišćenja identifikuju ključne karakteristike sistema i otkrivaju neke osnovne klase koje ćete upotrebiti. To su, u suštini, opisni odgovori na pitanja poput sledećih:[8]

- „Ko će koristiti ovaj sistem?“
- „Šta mogu ovi učesnici učiniti sa sistemom?“
- „Kako će ovaj učesnik to uraditi ovim sistemom?“
- „Kako bi još ovo moglo raditi ako bi to radio neko drugi ili ako bi isti izvođač imao neki drugi cilj?“ (za otkrivanje varijacija)
- „Kakvi se problemi mogu pojaviti dok se to radi sa sistemom?“ (za otkrivanje izuzetaka)

Ako, na primer, projektujete bankomat, slučaj korišćenja konkretne funkcije sistema je u stanju da opiše šta bankomat radi u svakoj mogućoj situaciji. Svaka od ovih situacija naziva se *scenario*, a slučaj korišćenja je skup scenarija. Možete razmišljati o scenariju kao pitanju koje počinje sa: „Šta sistem radi ako...? “. Na primer: „Šta radi bankomat ako je klijent uložio ček u intervalu od 24 sata, a na računu nema dovoljno novca da bi se obezbedila tražena isplata?“

[8] Zahvaljujem na pomoći Jamesu H Jarretu.

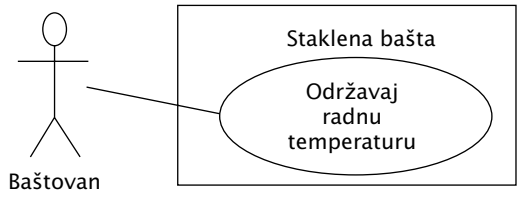
Dijagrami slučajeva korišćenja namerno su veoma jednostavni da bi vas sprečili da se prerano upuštate u detalje realizacije sistema:



Svaki simbol čoveka predstavlja „učesnika“ (engl. *actor*) koji je uglavnom osoba ili neka druga vrsta nezavisnog izvršioca. (Može biti čak i drugi računarski sistem, kao što je bankomat.) Pravougaonik predstavlja granicu vašeg sistema. Elipse predstavljaju slučajeve korišćenja, a to su opisi korisnog posla koji sistem može obaviti. Linije između učesnika i slučajeva korišćenja predstavljaju interakcije.

Nije bitno kako je sistem stvarno realizovan, sve dok ga korisnik ovako vidi.

Slučaj korišćenja ne mora biti užasno složen, čak i ako je sistem u njegovoj osnovi složen. Njegova jedina namena je da pokaže kako sistem izgleda korisniku. Na primer:



Slučajevi korišćenja proizvode definicije zahteva, određujući sve moguće interakcije između korisnika i sistema. Pokušajte da otkrijete potpuni skup slučajeva korišćenja svog sistema i imaćete jezgro onoga što sistem treba da radi. Dobra strana usredsređivanja na slučajeve korišćenja jeste što vas uvek vraća na suštinu i sprečava da zalutate u zahteve nebitne za završetak posla. Ako imate potpuni skup slučajeva korišćenja, možete opisati svoj sistem i preći na sledeću fazu. Verovatno neće sve biti savršeno izvedeno u prvom pokušaju, ali to je u redu. Sve će se razjasniti vremenom, a ako sada zahtevate savršenu definiciju sistema, upašćete u zamku.

Ako se nađete u zamci, ovu fazu možete prevazići primenom grube aproksimacije: opišite sistem u nekoliko pasusa, a zatim potražite imenice i glagole. Imenice ukazuju na učesnike, kontekst slučaja korišćenja (na primer, „predvorje“) ili tvorevine sa kojima se radi u slučaju korišćenja. Glagoli ukazuju na interakcije između učesnika i slučajeva korišćenja i definišu korake unutar slučaja korišćenja. Otkrićete takođe da imenice i glagoli proizvode objekte i poruke u fazi projektovanja (primetite da slučajevi korišćenja opisuju interakcije između podsistema, tako da tehnika „imenica i glagola“ može da se koristi samo kao pomoćno sredstvo pri razmišljanju, pošto se tako ne dobijaju slučajevi korišćenja).[9]

Granica između slučaja korišćenja i učesnika može ukazati na postojanje korisničkog okruženja, ali ona ne definiše to okruženje. Da biste se upoznali sa procesom definisanja i pravljenja korisničkih okruženja, pogledajte knjigu *Software for Use*, autora Larry Constantine i Lucy Lockwood, Addison-Wesley Longman, 1999. ili Web lokaciju *www.ForUse.com*.

U ovom trenutku je bitan osnovni raspored, mada to može delovati mistično. Imate pregled onoga što gradite, tako da ćete verovatno imati ideju koliko dugo će to trajati. U igri je veliki broj činilaca. Ako predvidite dug period, preduzeće može odlučiti da se sistem ne razvija (i da uloži resurse u nešto isplatljivije – to je *dobra* stvar). Rukovodstvo je možda već odlučilo koliko će trajati projekat i pokušaće da utiče na vašu procenu. Najbolje je odmah prikazati pošten raspored i čvrste odluke. Bilo je mnogo pokušaja da se lansiraju pouzdane tehnike raspoređivanja (poput tehnika predviđanja zbivanja na berzi), ali je verovatno najbolje da se pouzdate u svoje iskustvo i intuiciju. Pođite od svog osećaja koliko će proces zaista trajati, zatim udvostručite vreme i dodajte 10 procenata. Vaš unutrašnji osećaj je verovatno ispravan i vi *možete* doći do nekih rešenja tokom tog perioda. „Udvostručavanje“ će rad učiniti udobnijim, a 10 procenata ostaje za konačno doterivanje i detalje.[10] Kako god ga objasnili, i bez obzira na žalbe i manipulacije koje će uslediti kada obelodanite svoj raspored, izgleda da taj način vodi ka uspehu.

Faza 2: Kako ćemo izgraditi sistem?

U ovoj fazi morate doći do rešenja koje opisuje kako klase izgledaju i kako će sarađivati. Odlična tehnika za prepoznavanje klasa i interakcija jeste kartica *Klasa-Odgovornost-Saradnja* (engl. *Class-Responsibility-Collaboration*, *CRC*). Ovo sredstvo je značajno zbog svoje jednostavnosti: započinjete rad sa skupom kartica dimenzija 7×12 cm i pišete po njima. Svaka kartica predstavlja jednu klasu i na njoj ispisujete:

1. Ime klase. Bitno je da ovo ime obuhvata suštinu onoga što klasa radi, tako da na prvi pogled ima smisla.

---

[9] Više informacija o slučajevima korišćenja možete naći u knjizi *Applying Use Cases*, Schneider & Winters, Addison-Wesley, 1998. i *Use Case Driven Object Modeling with UML*, Rosenberg, Addison-Wesley, 1999.

[10] Moj pristup ovome se nedavno promenio. Dupliranje i dodavanje 10 procenata će vam dati prihvatljivo pouzdanu procenu (uz pretpostavku da nema previše džokerskih činilaca), ali ćete i dalje morati vredno da radite kako biste završili posao u roku. Ako vam je treba više vremena za elegantno rešenje, a pri tome hoćete da uživate u radu, verujem da je činilac kojim treba pomnožiti tri ili četiri.

2. „Odgovornost“ klase: šta ona treba da radi. Moguće je sažimanje nabrajanjem imena funkcija članica (pošto ova imena u dobrom projektu treba da budu opisna), ali se ne isključuju dodatne beleške. Ako hoćete da makar kako započnete proces, posmatrajte problem sa stanovišta lenjeg programera: koji objekti bi trebalo da se pojave da bi rešili vaš problem?
3. „Saradnja“ klasa: s kojim klasama ova klasa komunicira? Namerno je odabran širok pojam „komunikacija“, jer može značiti okupljanje ili samo postojanje neke druge klase koja obavlja usluge za objekte posmatrane klase. Saradnja treba da uzme u obzir i okolinu klase. Na primer, ako napravite klasu **Vatromet**, ko će je proučavati: **Hemičar** ili **Posmatrač**? Prvi će želeti da zna hemijski sastav, a drugi će reagovati na boje i oblike koji nastaju pri eksploziji.

Možda vam se čini da kartice treba da budu veće zbog svih informacija koje će obuhvatati, ali su one namerno malih dimenzija, ne samo zato da bi vaše klase bile male, nego i da se ne biste prerano upustili u detalje. Ako ne možete da smestite na karticu sve što treba da znate o klasi, klasa je previše složena (možda ste uneli previše detalja ili bi trebalo da napravite više od jedne klase). Savršena klasa treba da bude razumljiva na prvi pogled. CRC kartice pomažu da napravite grub projekat, tako da steknete globalnu sliku i potom doterujete svoje rešenje.

Komunikacija je jedna od velikih prednosti CRC kartica. Najbolje je raditi u grupi, neposredno, bez korišćenja računara. Svaki učesnik preuzima odgovornost za nekoliko klasa (one u početku nemaju imena, niti sadrže bilo kakve informacije). Tako obavljate simulaciju uživo, rešavajući jedan po jedan scenario, odlučujući koje se poruke šalju objektima da bi se scenario ostvario. Tokom ovog procesa otkrivate koje vam klase trebaju, njihove odgovornosti i saradnju, i pri tome popunjavate kartice. Kada prođete kroz sve slučajeve korišćenja, trebalo bi da imate prilično kompletan prvi nacrt projekta.

Pre nego što sam počeo da koristim CRC kartice, najuspešnije konsultantsko iskustvo sam stekao kada sam započeo projekat s timom koji nikada ranije nije koristio OOP i crtao objekte na tabli. Razgovarali smo o komunikaciji među objektima, neke od njih smo brisali i zamenjivali drugim objektima. Zapravo, radio sam sa „CRC karticama“ na tabli. Tim, koji je znao namenu projekta, došao je do rešenja – nije im dato rešenje, bilo je „njihovo sopstveno“. Ja sam pri vođenju projekta samo postavljao prava pitanja, testirao sam pretpostavke, primao povratne informacije od tima i menjao polazna stanovišta. Prava lepota procesa je bila u tome što tim nije učio objektno orijentisano projektovanje pregledajući apstraktne primere, nego su radili na svom rešenju, koje im je u tom trenutku bilo najzanimljivije.

Kada prikupite CRC kartice, možda ćete hteti da formalnije opišete svoje rešenje primenom UML-a.[11] UML ne morate upotrebljavati, mada može biti koristan, naročito ako želite da stavite dijagram na zid, da ga svi mogu proučavati, što je dobra ideja. Alternativa UML-u je tekstualni opis objekata i njihovih interfejsa ili neki pseudokod, što zavisi od korišćenog programskog jezika.[12]

UML obezbeđuje dodatnu grafičku notaciju za opis dinamičkog modela sistema. To je korisno kada su promene stanja sistema ili podsistema tako važne

[11] Početnicima preporučujem ranije spomenuti *UML Distilled*.
[12] Python (www.Python.org) često se koristi kao „izvršni pseudokod“.

da su potrebni posebni dijagrami (primer je sistem za upravljanje). Opis struktura podataka sistema ili podsistema može biti potreban ako su podaci najznačajniji faktor sistema (slučaj baza podataka).

Znaćete da ste završili fazu 2 kada budete opisali objekte i njihove interfejse, ili bar većinu, pošto neki uvek izmaknu i budu otkriveni tek u fazi 3. To je, u redu. Samo hoćete da pronađete sve objekte. Lepo je ako se otkriju na početku procesa, ali OOP obezbeđuje dovoljno materijala za rad, pa nije strašno ni ako ih otkrijete kasnije. Projektovanje objekata se odvija u pet nivoa, kroz proces razvoja programa.

Pet nivoa projektovanja objekata

Objekti se ne projektuju samo prilikim pisanja programa, već se projekat objekta postupno razvija. Ova perspektiva je korisna, pošto prestajete da očekujete trenutno savršenstvo i shvatate da ćete vremenom uvideti šta objekat radi i kako treba da izgleda. Ovaj pogled se može primeniti i na projektovanje različitih tipova programa: obrazac za projektovanje određenog tipa programa nazire se tek posle dužeg napora uloženog u rešavanje problema (*Obrasci projektovanja* su opisani u tomu 2). Za objekte takođe postoje obrasci koji nastaju kroz razumevanje, korišćenje i ponovnu upotrebu.

1. **Otkrivanje objekata.** Ovo je nivo početne analize programa. Objekti mogu biti otkriveni dok se traže spoljni činioci i granice, ponavljanja elemenata u sistemu i najmanje konceptualne celine. Neki objekti su očigledni ako već imate skup biblioteka klasa. Sličnost među klasama, koja ukazuje na osnovne klase i nasleđivanje, može se pojaviti na početku ili kasnije, u procesu projektovanja.
2. **Formiranje objekta.** Tokom pravljenja objekta, nastaće potreba za novim članovima koji se nisu pojavili pri otkrivanju. Unutrašnje potrebe objekta mogu zahtevati podršku drugih klasa.
3. **Izgradnja sistema.** U prethodnom nivou mogli su se pojaviti dodatni zahtevi. Objekte razvijate dok saznajete nove pojedinosti. Potreba za komunikacijom i povezivanjem s drugim objektima u sistemu, može izmeniti potrebe klasa i zahtevati nove klase. Na primer, možda će vam biti potrebne pomoćne klase, kao što je povezana lista, s malo ili bez ikakvih informacija o stanju, koje pomažu rad drugih klasa.
4. **Širenje sistema.** Dok dodajete nove osobine sistemu, možete otkriti da prethodno rešenje ne omogućava jednostavno proširivanje sistema. Uz nove informacije, možete restrukturirati delove sistema, i eventualno dodati nove klase ili hijerarhije klasa.
5. **Ponovna upotreba objekata.** Ovo je pravi test opterećenja za klasu. Ako neko pokuša da je ponovo upotrebi u sasvim novoj situaciji, verovatno će otkriti neke nedostatke. Dok menjate klasu da biste je prilagodili novim programima, opšte osobine klase se razjašnjavaju, sve dok ne dobijete tip koji zaista možete ponovo koristiti. Nemojte očekivati da ćete većinu objekata sistema ponovo upotrebljavati – sasvim je prihvatljivo da veliki broj objekata bude specifičan za određeni sistem. Ponovo upotrebljivi tipovi su ređi i koriste se za rešavanje opštijih problema.

Uputstva za razvoj objekata

Evo nekih preporuka za razvoj klasa:

1. Neka klasa nastane iz konkretnog problema, a zatim je razvijajte tokom rešavanja drugih problema.
2. Setite se da otkrivanje potrebnih klasa (i njihovih interfejsa) čini najveći deo razvoja sistema. Da ste već imali gotove klase, projekat bi bio jednostavan.
3. Nemojte se prisiljavati da shvatite sve na samom početku – učite tokom rada. U svakom slučaju, sve ćete saznati.
4. Započnite programiranje, dođite do nečeg što radi, tako da možete dokazati ili opovrgnuti svoje rešenje. Nemojte se plašiti da ćete završiti sa proceduralnim špagete kodom – klase raščlanjuju problem, lokalizujući problematične delove i nepoznate elemenete. Loše klase ne narušavaju dobre klase.
5. Zadržite jednostavnost. Mali i jasni objekti, od kojih imate očigledne koristi, bolji su od velikih složenih interfejsa. U času donošenja odluke koristite Occamov princip: razmotrite mogućnosti i uzmite najjednostavniju, pošto su jednostavne klase uvek najbolje. Počnite od malog i jednostavnog interfejsa klase, koji ćete širiti tek kada ga budete bolje razumeli, pošto je teško vremenom odstranjivati elemente iz klase.

Faza 3: Gradimo jezgro

Ovo je prvi prelaz od nacrta projekta u prevođenje i izvršavanje koda koji se može testirati, i koji će dokazati ili opovrgnuti ispravnost vaše arhitekture. Proces nije jednoprolazni, nego je početak niza koraka koji će iterativno izgraditi sistem, kao što ćete videti u fazi 4.

Cilj je da pronađete jezgro konstrukcije sistema, koje treba realizovati da bi se napravio izvršni sistem, bez obzira na nepotpunost u prvom prolazu. Pravite okvir koji možete nadograđivati u sledećim iteracijama. Osim toga, sprovodite prvu od mnogih faza integracija i testiranja sistema, i pružate investitorima povratnu informaciju o izgledu i napredovanju sistema. U najboljem slučaju, otkrivate i neke značajne rizike. Verovatno ćete otkriti šta treba promeniti u prvobitnoj arhitekturi i kako ćete je poboljšati – da niste realizovali sistem, ovo ne biste ni saznali.

Deo izgradnje sistema je provera podobnosti analize zahteva i definicije sistema (u kakvom god obliku postojali). Obezbedite da test proverava zahteve i slučajeve korišćenja. Po uspostavljanju stabilnog jezgra sistema, spremni ste da nastavite dodavanje novih mogućnosti.

Faza 4: Razvijamo slučajeve korišćenja

Kada se napravi izvršni okvir jezgra sistema, svaki skup osobina koji dodajete je jedan mali projekat. Skup osobina dodajete tokom *iteracije*, srazmerno kratkog perioda razvoja.

Koliko traje iteracija? U najboljem slučaju, iteracija traje od jedne do tri sedmice (može varirati u zavisnosti od jezika na kome se projekat realizuje). Na kraju

ovog perioda imate integrisani, testirani sistem, s većim brojem mogućnosti. Posebno je zanimljiva osnova iteracije: to je jedan slučaj korišćenja. Svaki slučaj korišćenja je paket međusobno povezanih funkcija koje odjednom ugrađujete u sistem, tokom jedne iteracije. Ne samo da se ovako dolazi do tačnijeg zaključka o opsegu slučaja korišćenja, nego se on i proverava, pošto se slučaj korišćenja ne odbacuje posle analize i projektovanja, već ostaje osnovna celina razvoja tokom celog procesa izgradnje softvera.

Prekidate sa iteracijama kada dostignete ciljnu funkcionalnost ili kada istekne rok, a kupac je zadovoljan trenutnom verzijom. (Prisetite se, razvoj softvera je plaćeni posao.) Kako je proces iterativan, za isporuku proizvoda imate mnogo mogućnosti, a ne samo krajnji termin. Projekti otvorenog izvornog koda razvijaju se isključivo u iterativnom okruženju, uz mnoštvo povratnih informacija, i upravo to ih čini uspešnim.

Iterativni proces razvoja je važan iz mnogo razloga. Rano možete otkriti i razrešiti bitne rizike, kupci imaju dovoljno mogućnosti da promene mišljenje, zadovoljstvo programera je veće i projektom se može preciznije upravljati. Još jedna značajna prednost je davanje povratne informacije investitorima, koji po trenutnom stanju proizvoda mogu tačno videti u šta ulažu. Tako se umanjuje ili eliminiše potreba za iscrpljujućim sastancima i povećava se poverenje i podrška investitora.

Faza 5: Evolucija

Ova tačka razvojnog ciklusa obično se naziva „održavanje“. Taj sveobuhvatni termin može značiti bilo šta, od „postizanja da radi na način kako je prvobitno zamišljeno“ do „dodavanja mogućnosti koje je kupac zaboravio da spomene“ ili, tradicionalnije, „otklanjanje otkrivenih grešaka“ i „dodavanje novih mogućnosti kada se ukaže potreba“. Mnoge zablude izaziva termin „održavanje“, jer poprima sumnjivo značenje, delimično zato što se nameće mišljenje da ste zapravo odavno napravili program i potrebno je samo da zamenite delove, podmažete ih i ne dozvolite da zarđaju. Možda postoji bolji izraz koji opisuje šta se tokom ove faze zaista dešava.

Koristiću termin *evolucija*.[13] Znači, „prvi put neće uspeti, zato dozvolite sebi da učite, a zatim se vratite i napravite izmene“. Možda će biti potrebno mnogo izmena nakon što dublje razumete osnovne zahteve. Elegantan kôd koji ćete napraviti tokom daljeg razvoja dok se ne dobije sasvim odgovarajući sistem, isplatiće se, i kratkoročno i dugoročno. Evolucija je proces u kome dobar program postaje sjajan i u kome se razjašnjavaju sporna pitanja iz prvog prolaza. U tom procesu se klase za upotrebu u jednom projektu razvijaju u ponovno upotrebljive klase.

„Odgovarajući sistem“ ne znači samo da program radi prema zahtevima i slučajevima korišćenja. To znači i da razumete unutrašnju strukturu programa, da se sve dobro uklapa, da nema nejasne sintakse, prevelikih objekata ili nespretno izloženih delova koda. Osim toga, morate imati i uvid da li će struktura programa preživeti promene kroz koje će neminovno prolaziti tokom svog životnog veka i da li se te promene mogu uneti lako i jasno. To nije mali podvig. Ne samo

[13] Bar jedan aspekt evolucije opisan je u knjizi *Refactoring: improving the design of existing code*, Martin Fowler, Addison-Wesley, 1999. Upozoravam vas da knjiga koristi isključivo primere na jeziku Java.

da morate razumeti šta pravite, nego i kako će se program dalje razvijati (ja to nazivam *vektor promene*[14]). Objektno orijentisani programski jezici su, na sreću, u stanju da podrže ovu vrstu neprekidnih promena – granice koje postavljaju objekti sprečavaju raspad strukture. Takođe vam omogućavaju da, bez izazivanja većih potresa u kodu, unosite izmene kakve bi delovale drastično u proceduralnom programu. Podrška evolutivnom razvoju je možda i najveća korist od OOP-a.

U evolutivnom razvoju pravite nešto bar približno onome što mislite da treba izgraditi, zatim to isprobavate upoređujući sa zahtevima i pronalazite nedostatke. Možete se vratiti i ponovo projektovati i realizovati delove programa koji nisu radili ispravno da biste otklonili nedostatke.[15] Možda ćete morati da rešavate problem, ili neki njegov aspekt, nekoliko puta pre nego što dođete do pravog rešenja. (Ovde može biti korisno proučavanje *obrazaca projektovanja*, opisanih u tomu 2.)

Evolucija se dešava i kada napravite sistem koji odgovara zahtevima, a zatim otkrijete da to nije ono što ste hteli. Kada vidite sistem u radu, otkrijete da ste želeli da rešite neki drugi problem. Ako očekujete ovakav tok razvoja, što pre napravite prvu verziju, da biste rano otkrili da li je rezultat ono što ste zaista nameravali.

Najvažnije je zapamtiti da će, po definiciji, ako menjate klasu, njene natklase i potklase i dalje raditi. Ne morate se bojati izmena (naročito ako imate ugrađeni skup testova za proveru njihove ispravnosti). Izmena ne mora obavezno narušiti ceo program, a rezultat bilo koje promene može biti ograničen na potklase i/ili pojedine saradnike promenjene klase.

Planiranje se isplati

Svakako ne biste zidali kuću bez mnoštva pažljivo iscrtanih planova. Ako pravite krov ili kućicu za psa, planovi neće biti toliko detaljni, ali ćete krenuti od nacrta koji će vas voditi. Razvoj softvera je otišao u krajnosti. Dugo ljudi nisu planski pristupali razvoju, pa su veliki projekti počeli da propadaju. Kao reakcija su nastajale metodologije s neverovatno složenom strukturom i obiljem detalja, uglavnom namenjene velikim projektima. Bilo je previše zastrašujuće koristiti te metodologije – izgledalo je kao da ćete sve vreme posvetiti pisanju dokumentacije i nimalo nećete programirati. (To je često i bio slučaj.) Nadam se da pristup koji sam vam ovde prikazao preporučuje srednji put – liniju napredovanja. Koristite pristup koji odgovara vašim potrebama (i vašim osobinama). Bez obzira na to što može biti i minimalan, *neki* plan će svakako puno poboljšati vaš projekat, za razliku od rada bez plana. Zapamtite da, po većini procena, preko 50 procenata projekata propada (neke procene idu do 70 procenata).

Radeći prema planu, po mogućnosti jednostavnom i kratkom, i projektovanjem strukture pre nego što počnete da programirate, otkrićete da se stvari

[14] Termin se koristi u poglavlju *Obrasci projektovanja* u tomu 2.

[15] Ovo izgleda kao „brza izrada prototipa“, u kojoj ste napravili brzu i grubu verziju, tako da možete upoznati sistem, posle čega odbacujete prototip i izgrađujete pravi sistem. Problem je što programeri nisu odbacivali brzi prototip, nego su ga nadograđivali. Kada se brzi prototip kombinuje s nedostatkom strukture u proceduralnom programiranju, često nastaju zamršeni sistemi čije je održavanje skupo.

uklapaju mnogo jednostavnije nego ako se odmah bacite na programiranje. Moje iskustvo govori da je elegantno rešenje izvor zadovoljstva na sasvim poseban način: bliže je umetnosti nego tehnologiji. Dostizanje elegancije se uvek isplati i to nije beznačajan posao. Dobijate program koji se jednostavnije piše i greške se lakše otklanjaju, program se lakše razume i održava, a tu se leži i finansijska dobit.

Ekstremno programiranje

Tehnike analize i projektovanja sam detaljno proučavao, još od studija. Koncept *ekstremnog programiranja* (engl. *Extreme Programming, XP*) je najdetaljniji i najbolji koji sam video. Hroniku možete pronaći u knjizi *Extreme Programming Explained*, autor je Kent Beck, izdavač Addison Wesley, 2000. i na Web adresi *www.xprogramming.com*.

XP je i način programiranja i skup uputstava. Neka od ovih uputstava odražavaju se i u drugim savremenim metodologijama, ali dva najvažnija posebna doprinosa su „prvo sastavite testove“ i „programiranje u paru“. Iako se čvrsto zalaže za proces u celini, Beck naglašava da ćete značajno povećati produktivnost i pouzdanost ako usvojite bar ova dva načela.

Prvo sastavite testove

Testiranje se tradicionalno ostavljalo za sam kraj projekta, pošto „sve radi, samo se treba uveriti“. Samo po sebi je imalo nizak prioritet, ljudi koji su se specijalizovali za testiranje nisu uživali naročit ugled i često su premeštani u suteren, daleko od „pravih programera“. Timovi za testiranje su odgovarali na svoj način, idući dotle da se oblače u crno, a veselo su se razmetali svaki put kada otkriju neku grešku (iskreno rečeno, i ja sam se tako osećao dok sam obarao C++ prevodioce).

XP u potpunosti preokreće koncept testiranja, dajući mu jednak (čak i viši) prioritet od pisanja programa. Testove pišete *pre* odgovarajućeg koda i oni ostaju zauvek uz kôd. Uspešno izvršavanje testova je obavezno posle svake integracije projekta (to se dešava često, nekada i više puta dnevno).

Pisanje testova ima dve posebno važne posledice.

Prvo, pisanje testova primorava da se jasno definiše interfejs klase. Često sam kao pomoćno sredstvo pri projektovanju sistema preporučivao ljudima da „zamisle savršenu klasu za rešavanje određenog problema“. XP strategija testiranja ide još dalje: tačno se definiše kako klasa mora izgledati njenom korisniku i kako se mora ponašati. Nije dozvoljena nikakva neizvesnost. Možete sastaviti kompletan tekst ili nacrtati sve dijagrame koji opisuju ponašanje i izgled klase, ali ništa nije tako stvarno kao skup testova. Tekst i dijagrami su lista želja, a testovi su ugovor koji podržavaju prevodilac i izvršni program. Teško je zamisliti realniji opis klase od testova.

Pri sastavljanju testova primorani ste da pažljivo pregledate klasu i često ćete otkriti neophodne funkcije koje mogu biti propuštene tokom eksperimenata sa UML dijagramima, CRC karticama, slučajevima korišćenja itd.

Druga važna posledica pisanja testova na početku jeste to što se testovi izvršavaju svaki put kada sklapate program. Izvršavanje testova se nadovezuje na provere koje sprovodi prevodilac. Ako iz ove perspektive pogledate evoluciju programskih jezika, videćete da je napredak tehnologije uvek pratilo uvođenje strožih testova. Asemblerski jezik je proveravao samo sintaksu, ali je C nametnuo neka semantička ograničenja koja sprečavaju određene tipove grešaka. OOP jezici donose dodatna semantička ograničenja koja su vidovi testiranja. „Da li je ovaj tip podataka pravilno upotrebljen? Da li je ispravan ovaj poziv funkcije?“ jesu testovi koje obavlja prevodilac ili izvršni sistem. Videli smo rezultate ugrađivanja ovih testova u jezik: ljudi prave složenije sisteme i puštaju ih u rad, i za to im treba mnogo manje vremena i napora. Pitao sam se šta je razlog, ali sam shvatio da se radi o testovima: učinite nešto pogrešno i sigurnosni sistem ugrađenih testova ukazuje na to da problem postoji i gde se nalazi.

Ali, testovi u samom jeziku nisu svemoćni. U nekom trenutku morate *vi* nastupiti i dopuniti skup testova (u saradnji s prevodiocem i izvršnim sistemom) koji proverava ceo program. Pošto shvatate koliko je korisno da vam prevodilac stalno gleda preko ramena, zar ne biste želeli da vam ovi testovi pomažu od samog početka? Zato ih i pišite na početku i automatski izvršavajte pri svakom sklapanju. Vaši testovi proširuju sigurnosni sistem programskog jezika.

Otkrio sam da me korišćenje sve moćnijih programskih jezika ohrabruje da eksperimentišem, pošto znam da mi taj jezik neće dozvoliti da gubim vreme loveći greške. XP šema testiranja radi istu stvar za ceo vaš projekat. Pošto znate da će testovi pronaći sve greške koje napravite (a vi redovno dodajete nove testove dok razmišljate o njima), možete po potrebi praviti velike promene, ne brinući da ćete izazvati potpunu zbrku u projektu. Ovo je neverovatno moćno.

Programiranje u paru

Programiranje u paru stoji nasuprot individualizmu kojim smo indoktrinirani od početka, kroz školovanje (uspećemo samostalno ili pasti, a rad sa susedima se smatrao „prevarom“) i medije, naročito holivudske filmove u kojima se često junak bori protiv bezumnog poretka.[16] Programeri se, takođe, smatraju uzorima individualizma – „usamljeni kauboji programeri“, što bi rekao Larry Constantine. XP, koji se bori protiv konvencionalnog načina razmišljanja, tvrdi da program treba da pišu dve osobe za jednom radnom stanicom. Osim toga, ovo treba raditi u prostoriji s većim brojem radnih stanica, bez paravana koje vole projektanti računarskih sala. Beck kaže da pri prelasku na XP prvo treba doneti klešta da bi se rasklopilo i bacilo sve što se nalazi na putu.[17] (Ovo podrazumeva da rukovodilac može da vas zaštiti od gneva osoblja za održavanje.)

Vrednost programiranja u paru je što jedna osoba piše program, a druga razmišlja o njemu. Onaj koji razmišlja ima u vidu globalnu sliku i uputstva XP-a, a ne samo trenutni problem. Ako radi dvoje, manje je verovatno, na primer, da će

[16] Iako je ovo pretežno američko stanovište, holivudski filmovi stižu svuda.

[17] Uključujući (posebno) razglas. Jednom sam radio u kompaniji koja je preko razglasa obaveštavala o telefonskim pozivima i to je neprekidno ometalo rad (direktori nisu mogli da zamisle da se ukine tako značajan sistem kao što je razglas). Konačno, kada me niko nije gledao, počeo sam da sečem kablove.

se jedno provući sa stavom: „Neću da pišem testove“. Ako programer upadne u problem, mogu zameniti mesta. Desi li se da oboje upadnu u problem, neko drugi u prostoriji čuće njihove nedoumice i priteći u pomoć. Rad u parovima održava tok i pravac posla. Možda je još važnije to što programiranje postaje humanije i zabavnije.

Počeo sam da primenjujem programiranje u parovima tokom vežbi na nekim seminarima i iskustva svih učesnika bila su znatno bolja.

Zašto C++ uspeva

C++ je toliko uspešan delimično zato što nije bio cilj samo da se C pretvori u OOP jezik (iako je tako počelo), nego i da se reše mnogi drugi problemi s kojima se danas suočavaju programeri, naročito oni koji su mnogo uložili u C. Tvorci OOP jezika tradicionalno su nastojali da nateraju korisnike da zaborave sve i krenu iz početka s novim skupom koncepata i novom sintaksom, uz obrazloženje da je, gledajući dugoročno, bolje ostaviti stari prtljag proceduralnih jezika. Na kraće staze, većina tog prtljaga je imala vrednost. Najvredniji element možda i nije postojeća programska osnova (koja se, uz odgovarajuće alatke, može prevesti), nego *osnova razmišljanja*. Ako ste aktivni C programer i morate napustiti sve što znate o C-u da biste usvojili nove jezike, smanjuje vam se produktivnost tokom više narednih meseci, sve dok se vaš način razmišljanja ne uklopi u novu paradigmu. Oslonite li se na svoje poznavanje C-a i proširite ga, bićete jednako produktivni dok se selite u svet objektno orijentisanog programiranja. Pošto svako ima svoj mentalni model programiranja, ova promena je dovoljno komplikovana i bez dodatnih troškova zbog započinjanja rada s novim jezikom od osnova. Razlog uspeha C++-a je u suštini ekonomski: mora se platiti cena prelaska na OOP, ali je cena za C++ možda niža.[18]

Cilj C++-a je povećanje produktivnosti. Produktivnost se postiže na mnogo načina, ali je jezik projektovan da vam što više pomogne, ne ometajući vas previše strogim pravilima ili zahtevima da koristite neki određeni skup mogućnosti. C++ je projektovan da bude praktičan: odluke pri projektovanju C++-a su bile zasnovane na pružanju što više korisnih mogućnosti programerima (bar sa stanovišta C-a).

Bolji C

Napredovaćete čak i ako nastavite da pišete kôd na C-u, pošto je C++ popunio mnoge praznine jezika C i uveo bolju proveru tipova i analizu pri prevođenju. Moraćete deklarisati funkcije, tako da prevodilac može da proveri njihovu upotrebu. Pretprocesor je postao skoro nepotreban, bar se više ne koristi za zamenu vrednosti i makroe, pa je uklonjen uzrok niza grešaka koje se teško otkrivaju. C++ ima svojstvo pod nazivom *referenca* (engl. *reference*) što omogućava pogodniju obradu adresa argumenata i povratnih vrednosti funkcija. Upotreba imena je poboljšana uvođenjem *preklapanja funkcija* (engl. *function overloading*) koja

[18] Kažem „možda“ pošto, s obzirom na složenost C++-a, može biti jeftinije preći na Javu. Odluka o izboru jezika ima mnogo činilaca, a u ovoj knjizi pretpostavljam da ste izabrali C++.

dozvoljava upotrebu istog imena za različite funkcije. *Imenski prostor* (engl. *namespace*) takođe unapređuje upravljanje imenima. Postoje i brojne osobine koje povećavaju bezbednost C-a.

Već ste na uzlaznoj liniji

Produktivnost je problem pri učenju novog jezika. Nijedno preduzeće ne može sebi priuštiti da iznenada izgubi produktivnog inženjera zato što uči nov jezik. C++ je proširenje C-a, koje nema potpuno novu sintaksu i model programiranja. To vam omogućava da nastavite pisanje korisnog koda i primenjujete postepeno nove mogućnosti dok ih učite i usvajate. Ovo može biti jedan od najvažnijih razloga uspeha C++-a.

Osim toga, svi vaši postojeći C programi i dalje su upotrebljivi u C++-u. Pošto je prevodilac C++-a usavršen, često ćete pronaći skrivene greške C koda dok ga prevodite u C++.

Efikasnost

Ponekad treba smanjiti brzinu izvršavanja zarad veće produktivnosti programera. Na primer, finansijski model može biti upotrebljiv samo u kratkom periodu, tako da je važniji brz razvoj modela od brzine izvršavanja programa. Međutim, većina aplikacija zahteva izvestan stepen efikasnosti, pa C++ često daje prednost efikasnosti u odnosu na lakoću programiranja. Pošto C programeri uglavnom veoma paze na efikasnost, ovo je argument protiv tvrđenja da je jezik previše glomazan i spor. Veliki broj mogućnosti C++-a je namenjen poboljšavanju performansi ako se ispostavi da izvršni program nije dovoljno efikasan.

Pored istih mogućnosti niskog nivoa za upravljanje kao u C-u (i mogućnosti da koristite asemblerski jezik unutar C++ programa), u praksi je dokazano da je brzina objektno orijentisanog C++ programa u intervalu ±10% u odnosu na brzinu C programa, a često je razlika u brzini veoma mala.[19]

Projekat napravljen za OOP program može biti mnogo efikasniji nego za takav program na C-u.

Sisteme je lakše opisati i razumeti

Klase, projektovane radi savladavanja problema, služe njegovom boljem izražavanju. Znači da prilikom pisanja programa opisujete rešenje u terminima oblasti problema („Stavite prsten na gomilu“), umesto terminologijom računara koji je prostor rešenja („Postavite vrednost bita u čipu tako da se relej zatvori“). Tako radite sa pojmovima višeg nivoa i mnogo više možete postići samo jednim programskim redom.

Druga dobit od ove jednostavnosti izražavanja potiče od održavanja na koje (ako je verovati izveštajima) odlazi veliki deo troškova tokom životnog veka programa. Lakše je održavati razumljiviji program. Snižava se i cena pisanja i održavanja dokumentacije.

[19] Podatke o značajnim istraživanjima performansi C++ biblioteke možete naći u časopisu *C/C++ User' Journal*, u kolumni čiji je autor Dan Saks.

Maksimalna dobit od biblioteka

Najbrže ćete napisati program ako koristite već napisan kôd – biblioteku. Značajni cilj C++-a jeste jednostavija upotreba biblioteka. Ovo je rešeno pretvaranjem biblioteka u nove tipove podataka (klase), tako da se uvođenjem biblioteke jeziku dodaju novi tipovi. Pošto C++ prevodilac vodi računa o upotrebi biblioteke, garantuje ispravnu inicijalizaciju i brisanje i obezbeđuje ispravnost poziva funkcija, možete se usredsrediti na ono što biblioteka treba da uradi a ne na to kako biblioteka radi.

Pošto imena funkcija mogu biti podeljena po delovima programa, pomoću imenskih prostora C++-a možete koristiti koliko god biblioteka hoćete, bez sukoba imena koji se dešavaju u C-u.

Ponovna upotreba izvornog koda uz šablone

Postoji značajna klasa tipova koji zahtevaju izmenu izvornog koda da bi se mogli efikasno ponovo koristiti. *Šablon* (engl. *template*) C++-a automatski menja izvorni program i moćno je sredstvo za ponovnu upotrebu biblioteka. Šablon koji radi s jednim tipom radiće bez problema s mnogim drugim tipovima. Šabloni su posebno moćni pošto od programera klijenta skrivaju složenost ovog načina ponovne upotrebe koda.

Obrada grešaka

Obrada grešaka u C-u je ozloglašena i često se ignoriše, pa se programeri često oslanjaju na sreću. Ako pravite velik, složen program, nema ničeg neugodnijeg od greške koja je negde skrivena, a nema nikakvog pokazatelja odakle potiče. *Obrada izuzetaka* C++-a (opisana u ovom tomu, a detaljno obrađena u tomu 2) garantuje da će se greška otkriti i da će biti obrađena.

Programiranje složenih sistema

Upotrebljivost mnogih tradicionalnih jezika ograničava veličina i složenost programa. Na primer, BASIC može biti sjajan za brzo rešavanje određenih kategorija problema, ali ako program preraste nekoliko strana ili izađe iz domena problema uobičajenih za taj jezik, dalji rad liči na pokušaj plivanja kroz gustu tečnost. I jezik C ima ovakva ograničenja. Na primer, kada program premaši otprilike 50000 redova koda, nailazi se na problem kolizije imena koji potiče od prekoračenja broja imena funkcija i promenljivih. Posebno težak problem su sitne pukotine C-a: greške skrivene u velikom programu se jako teško pronalaze.

Ne postoji jasna granica koja bi ukazala da je programski jezik ispod vaših očekivanja, a i da postoji, vi biste je ignorisali. Ne kažete: „Moj BASIC program je postao prevelik i moram ga ponovo napisati na C-u! “. Umesto toga, pokušavate da na silu umetnete još nekoliko redova kako biste dodali novu mogućnost. Tako vam se neosetno približavaju dodatni troškovi.

C++ je projektovan da podrži *velike projekte*, što znači da izbriše granice između malog i velikog programa pri postepenom povećavanju složenosti. Svakako ne morate koristiti OOP, šablone, imenske prostore i obradu izuzetaka kada pišete mali uslužni program u stilu „zdravo, svete“, ali ove mogućnosti su tu kada vam budu zatrebale. Osim toga, prevodilac detaljno traži greške, kako u malim, tako i u velikim programima.

Strategije unapređivanja

Ako vam se dopadne OOP, verovatno je vaše sledeće pitanje: „Kako da pridobijem šefa/kolege/odeljenje/susede da počnu da koriste objekte? “ Pomislite na to kako biste vi, jedan nezavisni programer, učili da koristite novi jezik i novu programsku paradigmu. Vi ste to već uradili. Na prvom mestu su obrazovanje i primeri, a zatim se radi probni projekat da biste naučili osnove, bez zbunjujućih detalja. Zatim stiže projekat iz „stvarnog sveta“ koji zaista radi nešto korisno. Tokom prvog projekta nastavljate učenje čitajući, postavljajući pitanja stručnjacima i razmenjujući saznanja s prijateljima. Ovaj pristup prelasku s C-a na C++ preporučuju mnogi iskusni programeri. Prelazak celog preduzeća na C++ sigurno je dinamičniji, ali je korisno prisetiti se kako bi to uradila jedna osoba.

Uputstva

Pri prelasku na OOP i C++ treba uzeti u obzir naredna uputstva:

1. Obuka

Prvi korak je obrazovanje. Prisetite se koliko je preduzeće uložilo u C programe i pokušajte da sprečite haos tokom sledećih šest do devet meseci, dok su svi zaokupljeni odgonetanjem kako radi višestruko nasleđivanje. Izaberite malu grupu za indoktrinaciju, po mogućnosti sastavljenu od radoznalih osoba koje dobro rade timski i mogu delovati kao mreža za podršku tokom učenja C++-a.

Alternativni pristup koji se ponekad preporučuje jeste istovremeno obučavanje na svim nivoima preduzeća, uključujući informativne kurseve za rukovodioce koji definišu strategiju, kao i kurseve projektovanja i programiranja za one koji se bave razvojem projekata. Ovo je posebno pogodno za manja preduzeća koja suštinski menjaju način rada ili za organizacione celine većih preduzeća. Pošto je cena viša, neki mogu odlučiti da započnu obučavanje na nivou projekta, naprave pilot projekat (eventualno uz pomoć spoljnog savetnika), posle čega projektni tim može podučavati ostale u preduzeću.

2. Projekat niskog rizika

Počnite od projekta niskog rizika i slobodno grešite. Kada steknete izvesno iskustvo, članovi prvog tima mogu započeti druge projekte ili postati tehnička podrška. Prvi projekat ne bi trebalo da bude značajan za preduzeće, pošto možda neće od početka raditi sasvim korektno. Trebalo bi da bude jednostavan, nezavisan i poučan, što znači da obuhvata stvaranje klasa koje će biti značajne drugim programerima u preduzeću kada dođe red na njih da uče C++.

3. Modelirajte na osnovu uspelih rešenja

Potražite primere dobrih objektno orijentisanih projekata pre nego što počnete od praznog lista hartije. Postoji velika verovatnoća da je neko već rešio vaš problem, a čak i ako nije rešen baš taj, verovatno možete apstrahovati i izmeniti postojeći projekat tako da odgovara vašim potrebama. Ovo je opšti koncept *obrazaca projektovanja*, opisanih u tomu 2.

4. Koristite postojeće biblioteke klasa

Osnovni ekonomski motiv za prelazak na OOP jeste jednostavnost upotrebe postojećeg koda u vidu biblioteka klasa (konkretno, standardnih C++ biblioteka koje su detaljno opisane u tomu 2 ove knjige). Najkraći ciklus razvoja aplikacije postižete ako ne napišete ništa osim funkcije **main**(), inicijalizujući i koristeći objekte iz gotovih biblioteka. Neki početnici ovo ne shvataju, nisu svesni postojećih biblioteka klasa ili, oduševljeni jezikom, žele da pišu klase koje već možda postoje. Najveći uspeh uz OOP i C++ postići ćete ako se potrudite da na početku prelaznog perioda pronađete i koristite kôd drugih autora.

5. Nemojte na C++-u ponovo pisati postojeći kôd

Iako *prevođenje* C programa C++ prevodiocem često daje (nekada ogromne) koristi jer se pronalaze problemi u starom kodu, ponovno pisanje ispravnih postojećih programa na C++-u često nije najbolji način da se iskoristi vreme. (Ako ga morate pretvoriti u objekte, možete „omotati“ C kôd C++ klasama.) Neka korist od toga postoji, naročito ako je kôd namenjen ponovnoj upotrebi. Možda nećete uočiti očekivano veliko povećanje produktivnosti, osim ako projekat nije potpuno nov. C++ i OOP se pokazuju u punom sjaju kada projekat radite od ideje do realizacije.

Teškoće upravljanja

Ako ste na rukovodećoj poziciji, vaša zaduženja su da nabavite resurse za tim, da prevaziđete prepreke na putu do uspeha i, uopšte, da pokušate da obezbedite najproduktivnije i najugodnije okruženje, tako da vaš tim može da stvori sva ona čuda koja se uvek traže od vas. Prelazak na C++ spada u sve tri navedene kategorije i bilo bi sjajno kada ne biste morali ništa da platite. Iako prelazak na C++ može biti jeftiniji, što zavisi od ograničenja,[20] nego druge OOP mogućnosti za tim C programera (i verovatno za programere na drugim proceduralnim jezicima), ovaj jezik nije besplatan i postoje teškoće kojih morate biti svesni pre nego što započnete prelazak na C++ unutar preduzeća i upustite se u troškove.

Početni troškovi

Troškovi prelaska na C++ prevazilaze cenu nabavke C++ prevodioca (GNU C++ prevodilac, jedan od najboljih, besplatan je). Srednjoročni i dugoročni troškovi će se smanjiti ako investirate u obuku (ako je moguće, angažujte i savetnika za prvi projekat) i ako izaberete i naručite biblioteke klasa koje rešavaju problem, kako te

[20] S obzirom na poboljšanja produktivnosti, trebalo bi razmotriti i jezik Java.

biblioteke ne biste pravili sami. Ovo su veliki troškovi, i moraju se uzeti u obzir pri planiranju. Postoje i skriveni troškovi pri učenju novog jezika, eventualno i novog programskog okruženja. Oni se svakako smanjuju kroz obuku i savetovanje, ali članovi tima moraju uložiti i sopstveni napor pri usvajanju nove tehnologije. Za to vreme tim će više grešiti (ovo je prirodno, pošto je uviđanje grešaka najbrži način učenja) i biće manje produktivan. Čak i tada, s nekim problemima u programiranju, uz prave klase i u odgovarajućem okruženju, moguće je ostvariti veću produktivnost (uzimajući u obzir i veći broj grešaka i manji broj programskih redova na dan) nego ako se zadržite na C-u.

Pitanja performansi

Uobičajeno pitanje je: „Zar OOP mnogo ne uvećava i ne usporava moje programe?“. Odgovor je: „Zavisi“. Većina tradicionalnih OOP jezika napravljena je tako da omogući eksperimentisanje i brzu izradu prototipova, a ne izradu efikasnih sistema. Zato je delovalo da će objektni programi biti veći i sporiji. C++ je namenjen produktivnom programiranju. Kada se usredsredite na brze prototipove, možete ubacivati komponente zanemarujući pitanja efikasnosti. Ako koristite gotove biblioteke, one su obično već optimizovane, pa vi to ne morate raditi dok razvijate prototip. Kada dođete do odgovarajućeg sistema koji je dovoljno mali i brz, posao je završen. U suprotnom započinjete podešavanje parametara, otkrivajući šta se sve može ubrzati jednostavnom primenom ugrađenih opcija C++-a. Ako to ne pomogne, ispitujte kako se može promeniti program, ne menjajući kôd koji koristi pojedinačne klase. U slučaju da ništa ne uspe, menjajte projekat. Činjenica da su performanse veoma važne u nekom delu projekta je pokazatelj da one moraju biti deo početnih kriterijuma. Korisno je da to rano otkrijete primenjujući brzi razvoj.

Ranije je rečeno da se razlika u brzini između C-a i C++-a najčešće nalazi u intervalu ±10%, a često je sasvim mala. Veliko skraćivanje izvornog koda i ubrzavanje programa primenom C++-a može poticati od razlike u projektima pravljenim za C i C++.

Podaci o poređenjima C-a i C++-a su anegdotski i verovatno će tako i ostati. Iako neki preporučuju da se isti projekat realizuje i na C-u i na C++-u, to sebi mogu priuštiti samo velika preduzeća, zainteresovana za ovakve istraživačke projekte. Čak i tada izgleda da je novac mogao biti bolje uložen. Najubedljiviji argument je da su skoro svi programeri koji su sa C-a (ili nekog drugog proceduralnog jezika) prešli na C++ (ili neki drugi OOP jezik) znatno poboljšali svoju produktivnost.

Uobičajene greške pri projektovanju

Pri uvođenju tima u OOP i C++, programeri će najčešće prolaziti kroz niz uobičajenih grešaka. Često je razlog nedostatak stručnih saveta tokom projektovanja i realizacije prvih projekata, pošto u preduzeću još nema stručnjaka za C++, a može postojati otpor prema angažovanju savetnika. Možete prerano steći utisak da razumete OOP i krenuti pogrešnim putem. Ono što je iskusnom programeru očigledno, može biti izvor velikih nedoumica za početnika. Mnoge nevolje se mogu izbeći ako se za obuku i savetovanje angažuje spoljni saradnik.

Sa druge strane, činjenica da je lako napraviti ove greške u projektovanju ukazuje na osnovni nedostatak C++-a: povratnu kompatibilnost sa C-om (to je, naravno, i njegova osnovna moć). Da bi se izveo podvig prevođenja C programa, jezik je morao da napravi neke kompromise, što je dovelo do nastanka „tamnih strana". Takvi problemi postoje i često se javljaju tokom učenja C++-a. U ovoj knjizi i narednom tomu (i u drugim knjigama – videti dodatak C) pokušavam da razotkrijem najveći broj zamki na koje ćete naići radeći u C++-u. Uvek morate imati na umu da u bezbednosnom sistemu postoje pukotine.

Sažetak

Namena ovog poglavlja je da vas uvede u široki skup osobina objektno orijentisanog programiranja i C++-a, uključujući različitosti OOP-a i posebno C++-a, ideje OOP metodologija i vrste zahteva na koje ćete u preduzeću nailaziti pri prelasku na OOP i C++.

OOP i C++ možda nisu dobri za sve situacije. Važno je proceniti sopstvene potrebe i da li ih C++ optimalno ispunjava ili bi bilo bolje nastaviti sa nekim drugim programskim sistemom (uključujući i onaj koji trenutno koristite). Ako znate da će vaše potrebe biti sasvim posebne u doglednoj budućnosti i imate jasna ograničenja koja C++ možda neće ispuniti, tada bi trebalo da proučite i druga rešenja.[21] Čak i ako izaberete C++, razumećete kakav izbor ste imali i imaćete jasnu sliku zašto ste izabrali taj put.

Znate kako izgledaju proceduralni programi: definicije podataka i pozivi funkcija. Da biste otkrili značenje takvog programa i stekli sliku modela, morate pratiti pozive funkcija i shvatiti osnovne ideje. Zato su nam potrebni pomoćni načini predstavljanja rešenja proceduralnog programa – sami po sebi, ovi programi zbunjuju pošto je izražavanje rešenja više orijentisano ka računaru nego ka problemu.

Pošto C++ dodaje mnoštvo novih svojstava jeziku C, prirodna pretpostavka je da će funkcija **main()** u C++ programu biti mnogo komplikovanija nego u ekvivalentnom C programu. Ovo je prijatno iznenađenje: dobro napisan C++ program, u opštem slučaju, mnogo je jednostavniji i razumljiviji od ekvivalentnog C programa. Videćete definicije objekata koji predstavljaju problem (a ne specifičnosti računara) i poruke poslate objektima koje predstavljaju aktivnosti. Jedna od pogodnosti objektno orijentisanog programiranja jeste lakoća razumevanja dobro projektovanog programa. Obično ima i mnogo manje koda, pošto će većina problema biti rešena ponovnom upotrebom postojećih biblioteka.

[21] Posebno preporučujem da pogledate Web lokacije jezika Java (http://java.sun.com) i Python (http://www.Python.org).